

CS442/542  
Shell Project Part II – Functional Bash  
Due: Friday, November 1st - 11:59pm

## Program

This is the second half of your shell project. You will start with the work that you already turned in and complete a simple, but useful shell. Input to *bash* consists of lines using the same grammar as the first part of the assignment, with additions as listed below.

In this part of the project *cmd* is either an absolute path name or just a command name, as in:

`ls`                    `/bin/ls`                    `./a.out`

If the command name, *cmd*, is absolute or starts with “./”, *bash* should run the program specified without further processing. If the command name is not an absolute path, *bash* must search the directories in the PATH environment variable to find a directory containing the command and run the appropriate command, if found. For example, if the PATH environment variable contains:

`/bin:/usr/bin:/etc`

then *bash* should interpret the command “ls” as referring to the program “/bin/ls” and the command “mount” as referring to the program “/etc/mount”. Note that the system call **access()** may be helpful for determining if an executable file exists with a given name.

*Bash* will redirect the input and output of the command to the files specified on the command line, if included. By default, of course, *bash* should leave the stdin, stdout, and stderr attached to the terminal. Error checking and reporting is particularly important here.

## Quotes

Our Bash program will support two forms of quoting:

- double quotes  
any characters between double quotes will need to have variables expanded (see below)
- single quotes  
any characters between single quotes are left uninterpreted

You’ll need to think carefully about how to handle so that you can do the variable expansion later!

## Builtin Commands

Our Bash will support the following built-in commands:

**echo optarg1 optarg2 ...**

print each of the arguments to standard output separated by a single space (just like /bin/echo)

**cd**

change directory to the value of the HOME envvariable

**cd dir**

change directory to **dir**

## Shell Variables

Bash assignments are of the form

```
varname=value
```

There can be no spaces either before or after the equals sign. Value may, however, be a quoted string of any of the two forms above. Legal variable names must start with an upper or lower case letter and may consist **only** of upper and lower case letters, digits, and the underscore. References to variables are of two forms as given below, and references to undefined variables must result in NULL strings.

```
STRA="this is a test of my"
STRB=program
echo $STRB
echo ${STRA} ${STRB}
echo "${STRA} ${STRB}"
echo '${STRA} ${STRB}'
```

to which the output would be

```
program
this is a test of my program
this is a test of my program
${STRA} ${STRB}
```

Three of the variables, PATH, PS1, and DEBUG, have meaning to your shell. Their initial values should be read from the *environment* of the shell and any changes should immediately go back into the process environment. Changing PATH MUST change the shell's search PATH immediately and changing PS1 must change the prompt that the shell uses and install PS1 into the environment.

```
PS1="Yes Boss> "
PATH="/bin:/usr/bin:/usr/local/bin"
```

## \$VAR or \${VAR}

As part of the variable expansion, you will need to perform expansions on WORDs (I found it easier to perform this in the scanner, but you can do it wherever you want).

Note that none of these expansions will occur if the original WORD appeared in single quotes, but they will occur if it was in double quotes.

You will definitely need a subroutine that takes a string (with any quotes still attached) and returns a new string with any embedded variables expanded.

Unlike the real bash program, our version will have **ONLY** environment variables. Any setting of a variable should cause the variable **varname** to become part of the environment of your shell. Any references to variables (as discussed above) should be satisfied by checking for them in the environment.

## Debugging

Shell-level debugging is enabled when the value of the envariable DEBUG is non-NULL, and disabled when it is NULL.

```
bash> DEBUG=yes
bash> echo "this is a test" > /tmp/junk
Debug:  program name: "/bin/echo"
Debug:  argv[0]: "echo"
Debug:  argv[1]: "this is a test"
Debug:  stdin: (undirected)
```

```

    Debug:  stderr: (undirected)
    Debug:  stdout: "/tmp/junk"
this is a test
bash> DEBUG='',
bash> echo "now debugging is off"
now debugging is off

```

Note that when debugging is enabled, you must **still** execute the commands in addition to printing information about them. You may add any other code to the debugging output that you wish. I only require that there be some debugging output present. The more use you make of this feature, the easier time you will have in debugging your program.

## Requirements

1. Your program must locate commands using the `PATH` environment variable and run them.
2. Your program must support the input and output redirection from the first part of the shell.
3. Your program should generate a prompt, as given by the `PS1` environment variable.
4. Your program must work for *any* reasonable number of arguments (but you must have a carefully checked limit) and must give meaningful error messages.
5. Your shell should terminate neatly and quietly when it reaches the end of file.
6. Your shell must issue a prompt before reading input.
7. Your shell must `wait()` for all of its children.

## Hints

You may find the following Unix system calls and library routines helpful:

`access()`, `chdir()`, `fork()`, `execv()`, `dup2()`, `getenv()`, `putenv()`, and `perror()`

I recommend that you implement the features in the following order (roughly the order of the test.? `files`):

1. pull out all of the “printing the command” code out of `doline()` into a new subroutine that you can use to debug commands
2. implement the builtin “echo” command as a subroutine. Then update `doline()` so that finding that the commandname is exactly “echo” is a special case that calls that routine
3. run commands with absolute path names as a second special case in `doline` (after it’s not “echo”).
4. look up the `PS1` environment variable and use it as the `PROMPT` to print before every command (do NOT print a newline). You will also need to do a `fflush(stdout)`; after every prompt to get them to show up correctly.
5. use the environment variable `PATH` to search for commands. Implement it as a subroutine that takes a command name and returns either `NULL` (if it’s not found) or a full path name if the command is in the `PATH`.
6. add `stdin/stdout/stderr` redirection (simple - use class examples)
7. implement variable assignment
8. implement variable substitution
9. implement “and lists” and “or lists” (this will require careful design)
10. finish any other required pieces from the test files

You should budget your time and priorities to allow a day or two to implement each of those tasks over the 1 month that you have to finish the project (and remember that there's a midterm during that time). If you start right away and allow an hour or two to work on the project on most days, it's quite feasible to finish the project in plenty of time.

**DO NOT MODIFY YOUR ORIGINAL PATH, MAKE A COPY OF IT!!!**

**ANY ASSIGNMENT SUBMITTED THAT USES AUTOMATIC PATH SEARCHING (like `execve()`, `execle()`, etc) WILL RECEIVE ZERO CREDIT**

**Check the return values for all system calls, it's always worth the extra effort!!!**

The syntax of the assignment statement is a little difficult to deal with, so you'll want to think carefully before starting.

Expanding variables in strings is challenging and neither Flex nor BISON will be much help. When you get to that part, you should write a good variable expansion function in C++ that takes a string and returns a new string with all variables expanded.

### Extra Credit Ideas

For those of you who want to further explore the project, and want to earn **at most** an additional 20% on the project, I suggest that you try to make your shell more fully-functional, perhaps using the following ideas:

#### Implement pipes

They aren't hard, but will take a while to debug. They also require that your `doline` command be carefully designed to support multiple commands. If you do NOT implement them, then any construct that uses them should generate an error in `doline()`.

#### Implement conditionals and loops

This will require more careful design.

#### history

Add history to *bash*. Use whatever syntax you want, but you might want to stick with *bash*'s simple mechanism. You might look into the GNU project's `readline` library here:

[https://web.mit.edu/gnu/doc/html/rlman\\_2.html](https://web.mit.edu/gnu/doc/html/rlman_2.html)

#### aliasing

Add aliasing to *bash* by using the command "`alias [match replacement]`", as in

```
alias ls "/bin/ls -F"
```

#### whatever you want

Think about what else the shell does for you and how you might make that work in *bash*. The command `man bash` will give you hundreds of ideas.