CS442/542
Shell Project Part II – The New BASH
Due: Wednesday, October 20th - 11:55pm

version 1.0 - Mon Oct 4, 2021

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!

LOOPS

Our Bash will support simple looping constructs just as in the regular shell:

```
while commandlist
  do
     commandlist (can be on several lines)
  done

for VAR in str1 str2 str3 ...
  do
     commandlist (can be on several lines)
  done
```

New Tokens

You will need to add the following reserved tokens to your scanner:

for while in do done All of these tokens are used for the looping constructs that we will implement and are now reserved words in our shell

semicolon A semicolon (;) works the same as an EOLN token (in the parser), but it doesn't advance the line number counter

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 three 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}
```

to which the output would be

```
program
this is a test of my program
```

Three of the variables, `PATH`, `PROMPT`, 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 `PROMPT` must change the prompt that the shell uses and install PROMPT into the environment.

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

Finally, the command

```
export varname
```

should cause the variable `varname` to become part of the environment of your shell. Any further changes to `varname` must also be propogated into the shell's environment, which is inherited by its children.

Builtin Commands

Our Bash will support the following built-in commands:

cd
: change directory to the value of the `HOME` envariable

cd directory
: change directory to `directory`

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 in 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.

Word Expansion

As part of the variable expansion, you will need to perform the following expansions on WORDs (in your main program, not in lex or yacc):

$VAR or ${VAR}

variable expansion

`${name#pattern}, ${name##pattern}`

remove the pattern from the beginning of the variable's value before returning it. These should work as in the regular shell except that our pattern can only be a fixed string (no wildcarding is needed).

`$name%pattern, $name%%pattern` same as the `$name#pattern` rule, but at the END of the string.

Note that none of these expansions will occur if the original WORD appeared in single quotes!

Requirements

1. Your program must locate commands using the `PATH` envariable 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 `PROMPT` envariable.
4. Your program must work for *any* reasonable number of arguments 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()`, `dup()`, `dup2()`, `getenv()`, `putenv()`, `perror()`, and `index()`

Implement the features in the following order:

1. run commands with absolute path names (trivial)
2. add stdin/stdout/stderr redirection (simple - use class examples)
3. use PATH to find commands
4. implement variable assignment
5. implement variable substitution
6. implement the looping constructs

You should budget you time and priorities to allow a day or two to implement each of those tasks over the 2 weeks that you have to finish the project. If you start right away and allow a couple of hours 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 difficult and neither Lex nor YACC will be much help. When you get to that part, you should write a good variable expansion function that takes a string and returns a new string with all variables expanded.

For the looping constructs, you'll probably need to make a new data structure that holds an entire construct, rather than just a single command. Then, for loops, yacc will return an entire SET of commands using that data structure.

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. If you do NOT implement them, then any construct that uses them should generate an error in doline(). **IMPLEMENTING PIPES IS REQUIRED FOR GRADUATE STUDENTS**

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`

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.