UVic CSC 360 Assignment 1 Part 2: *SEE*sh
A UNIX Shell
Due Jan 31 10pm

Objective
To learn about UNIX processes, and the command line interpreter.

Background
A UNIX **shell** is a program that makes the facilities of the operating system available to interactive users. There are several popular UNIX shells: *sh* (the Bourne shell), *csh* (the C shell), and *bash* (the Bourne Again shell) are just a few. In this assignment, you will build *SEE*sh.

Your Task
You need to create a C program in a file named SEEsh.c. SEEsh should be a minimal but realistic interactive UNIX shell.

Initialization and Termination
When first started, SEEsh should read and interpret lines from the file **.SEEshrc** in the user's **HOME** directory, provided that the file exists and is readable. Note the file name is **.SEEshrc** (with the leading ".", not SEEshrc), and that it resides in the user's HOME directory (where HOME is an environment variable that must be used!). Typically, the .SEEshrc file contains commands to specify the terminal type and environment.

To facilitate your debugging and our testing, SEEsh should print each line that it reads from .SEEshrc immediately after reading it. SEEsh should print a question mark and a space (?) before each line as a prompt to the user.

SEEsh should terminate when the user types 'Control-D' or 'exit'.

Interactive Operation
After startup processing, SEEsh should read lines from the terminal, prompting with a question mark and a space (?). Specifically, SEEsh repeatedly should perform these actions:

1. Read a line from standard input.
2. Lexically analyze the line to form an array of **tokens**.
3. Syntactically analyze (i.e. parse) the token array to form **command [options] [args]** typically fed to the command interpreter.
4. Execute the **command**.

Lexical Analysis
Informally, a **token** should be a word. More formally, a token should consist of a sequence of non-whitespace characters that is separated from other tokens by whitespace characters. SEEsh should assume that no line of standard input is longer than 512 characters. If a line of standard

input is longer than 512 characters, then SEEsh need not handle it properly; but it should not corrupt memory.

Execution
If the command is a SEEsh built-in, then SEEsh should execute it directly (i.e. without forking a child process). SEEsh should interpret six shell built-in commands: *cd, help, exit, pwd, set, unset.*

- **cd [dir]** SEEsh should change SEEsh's working directory to dir, or to the **HOME** directory if dir is omitted.
- **pwd** print the full filename of the current working directory.
- **help** display information about builtin commands.
- **exit** SEEsh should exit.
- **set var [value]** If environment variable var does not exist, then SEEsh should create it. SEEsh should set the value of var to value, or to the empty string if value is omitted. If both var and value are omitted, set should display all environment variables and values. *Note:* Initially, SEEsh inherits environment variables from its parent. SEEsh should be able to modify the value of an existing environment variable or create a new environment variable via the set command. SEEsh should be able to set the value of any environment variable; but the only environment variables that it explicitly uses are **HOME** and **PATH**.
- **unset var** SEEsh should destroy the environment variable var.

Note that those built-in commands should neither read from standard input nor write to standard output.

If the command is not an SEEsh built-in, then SEEsh should consider the command-name to be the name of a file that contains executable binary code. SEEsh should use the PATH environment variable to locate the binary, fork a child process and pass the filename, along with its arguments, to the *execvp* system call. If the attempt to execute the file fails, then SEEsh should print an error message indicating the reason for the failure.

SEEsh should print its prompt for the next standard input line only when a command has finished executing.

Process Control
All child processes forked by SEEsh should run in the foreground; SEEsh need not support background process control. However, the user must be able to terminate the current child processes using Control-C. Control-C should not terminate SEEsh itself.

Error Handling
SEEsh should handle an erroneous line gracefully by rejecting the line and writing a descriptive error message to standard error. SEEsh should handle **all** user errors; it should be impossible for the user's input to cause SEEsh to crash.

Memory Management
SEEsh should contain no memory leaks. For every call to malloc or calloc, there should eventually be a call to free.

History Mechanism (for extra credit)
SEEsh could also support a history mechanism that includes:

- A **history** built-in command. The history command should print a list of all previously issued commands. Note that the **history** command should write data to standard output.
- The ability to re-execute a previously issued command by typing a prefix of that command preceded by an exclamation point (**!*commandprefix***).

SEEsh need not support editing of the previously issued command.

Logistics
Make sure your code will run on our server, linux.csc.uvic.ca. The expectation is that you will devote substantial effort to creating and running test cases. Your shell will be tested with our own .SEEshrc file, so you do not have to upload yours.

You should submit:

- Your source code files for **both** your doubly linked list (Part 1 of the assignment) and your shell (Part 2 of the assignment).
- A makefile for your shell. The first dependency rule should build your entire program, compiling with the -Wall and - Werror options. The makefile should maintain object (.o) files to allow for partial builds. The executable must be called 'SEEsh'.
- A readme file.

Your readme file should contain:

- Your name.
- A description of whatever help (if any) you received from other resources while doing the assignment, and the names of any individuals with whom you collaborated.
- Optionally, any information that will help us to grade your work in the most favorable light. In particular you should describe any known bugs!

Submit your work electronically via CourseSpaces.

Grading
Your work will be graded on the basis of your code being complete, correct, and clear. To encourage good coding practices, we will take off points based on warning messages during compilation.

Complete: Are all of the features implemented? Are the test cases sufficient?	Poor, needs some work, good, excellent. (Where poor means mostly *missing*, and excellent means *outstanding*.)
Correct: Are the test cases working? Is all the functionality working?	Poor, needs some work, good, excellent.
Clear: Is the code well written and easy to understand? Is the documentation helpful?	Poor, needs some work, good, excellent.