KAIST
EE 209: Programming Structures for EE

Assignment 5: A Unix Shell - Supplementary Information

(This page is borrowed and slightly modified from Princeton COS 217)

The following are implementation requirements and recommendations for your shell program:

General

(Required) Your program should be *modular* at the function level. Define small functions, each of which does a single well-defined job.

(Required) Your program should be *modular* at the source code file level. Define interfaces and implementations, thus splitting your program into multiple files. Define stateless modules, abstract objects, or abstract data types as appropriate. Encapsulate data structures with functions.

(Recommended) Your program may use the `DynArray` ADT (from early precepts). The source code is available here: dynarray.h dynarray.c.

(Required) Your program should interpret commands from the `.ishrc` file when it is first launched. We will test your program by repeatedly copying command sequences to the `.ishrc` file and launching your program. If your program does not interpret commands from that file, then it will fail all of our tests. In that unfortunate circumstance, the grade would be penalized substantially.

(Required) Your program should look for the `.ishrc` file in the HOME directory, not in the working directory.

(Required) Your program should work properly if the `.ishrc` file does not exist or is not readable. It is **not** an error for the `.ishrc` file to not exist or to be unreadable.

(Required) Your program should print the commands from `.ishrc`. That is, immediately after your program reads a command from the `.ishrc` file, it should print its "% " prompt and that command to the standard output stream. In that manner your program should generate a transcript that shows each command of `.ishrc`, followed by the output that results from executing that command. If your program does not print the commands of `.ishrc` to the standard output stream, then it will be difficult to interpret the transcript. In that unfortunate circumstance, the grade would be penalized substantially.

(Required) Your program should not print the commands from the standard input stream. That is, when your program reads a command from the standard input stream, it should not print that command to the standard output stream.

Error Handling

(Required) Your program should detect and report each *programmer* error via a call of the `assert` macro. A programmer error is one that could not be caused by user input.

(Required) Your program should call the `assert` macro to validate the parameters of every function in your program, especially the non-static ones.

(Required) Your program should detect each *user* error via an `if` statement, and should report each user error via a descriptive error message. A user error is one that could be caused by user input.

(Required) Your program should check the result of each attempt to dynamically allocate memory. It should recover from dynamic memory allocation failures as gracefully as possible.

(Required) Your program should print error messages to the *standard error* stream, not to the *standard output* stream.

(Required) Your program's error messages should begin with *`programName`*`:`, where *`programName`* is your program's `argv[0]`. Note that the name of your program is not necessarily "ish"; the user might have renamed the executable binary file to something other than "ish".

(Required) After a failed call of a function that sets the `errno` variable, your program should call the `perror` or `strerror` function to print an appropriate error message.

Lexical Analysis

(Recommended) Your program should read each input line by calling the `fgets` function.

(Recommended) Implement your lexical analyzer as a deterministic finite state automaton.

(Required) Your program should call the `isspace` function to identify white-space characters.

(Recommended) Create some temporary code that prints the token list created by your lexical analyzer. Do not proceed to subsequent phases until you test your lexical analyzer thoroughly.

(Recommended) Test your lexical analyzer by making sure that it handles these example input lines properly:

INPUT LINE	TOKEN ARRAY
`one`	`one`
`123`	`123`
`one123`	`one123`
`123one`	`123one`
`one two`	`one` `two`
`one  two`	`one` `two`
`  one two`	`one` `two`
`one |`	`one` `|`
`one|`	`one` `|`
`|one`	`|` `one`
`"one"`	`one`
`"|"`	`|` (an ordinary token)
`"one two"`	`one two`

one"two"	onetwo
"one"two	onetwo
"one	Invalid: Unmatched quote
one"two	Invalid: Unmatched quote
'	Invalid: Unmatched quote

(Recommended) Your program's lexical analyzer should represent tokens so that the difference between quoted and unquoted special characters is adequately captured. For example, these two commands are very different, and that difference should be captured at the lexical analysis phase:

```
cat file1 | grep str
cat file1 "|" grep str
```

Syntactic Analysis

(Recommended) Your program should do as much validation of the command as possible at the syntactic analysis phase. The more error checking you do during the syntactic analysis phase, the less must be done during the (more complicated) execution phase.

(Recommended) Create some temporary code that prints the command created by your syntactic analyzer. Do not proceed to subsequent phases until you test your syntactic analyzer thoroughly.

(Recommended) Test your syntactic analyzer by making sure that it handles these valid and invalid example input lines properly:

INPUT LINE	VALID/INVALID
cat	Valid
cat file1	Valid
cat file1 | grep main	Valid
cat file1 | cat	Valid
cat file2 | grep test | wc -l	Valid
cat file1 |	Invalid: Missing command name
| file1	Invalid: Missing command name
cat file1 | | wc -l	Invalid: Missing command name

Execution

(Required) Your program should call `fflush(NULL)` before each call of `fork` to clear all I/O buffers.

(Required) Your program should call the `setenv` function to implement the `setenv` shell built-in command.

(Required) Your program should call the `unsetenv` function to implement the `unsetenv` shell built-in command.

(Required) Your program should call the `chdir` function to implement the `cd` shell built-in command.

(Required) Your program should call `exit(0)`to implement the `exit` shell built-in command. Or, implement the `exit` command by returning 0 from the program's `main` function.

Process Handling

(Required) Your program should run any commands that ends with & in the background.

(Required) Your program should call wait for every child that has been created.

(Required) Your program should notify creation and termination of background process to stdout with its pid.

(Required) Your program should bring the latest background process to the foreground when `fg` command is executed.

(Required) Your program should print out appropriate error messages to stderr for inappropriate usage or circumstances for & and `fg` commands.

(Recommended) Test your program's background processes functionality with sleep commands.

Signal Handling

(Required) Your program should call the `signal`function to install signal handlers.

(Required) Your program should call the `alarm` function to control handling of SIGQUIT signals.

(Required) Your program should use the `SIG_IGN` and `SIG_DFL` arguments to the `signal` function, as appropriate.

(Required) Your program should handle the `SIGCHLD` signal for reaping the 'zombie processes' (i.e., child processes that have finished but are not yet reaped by the parent process) when implementing background processes. You can call `wait() or waitpid()` to reap the finished child processes.

(Required) Your program should call the `sigprocmask` function near the beginning of the `main` function to make sure that `SIGINT`, `SIGQUIT`, and `SIGALRM` signals are not blocked.

Redirection

(Required) Your program should treat '<' and '>' as special characters in lexical analysis.

(Required) Your program should treat redirection of standard out in LHS of a pipe token as a syntax error.

(Required) Your program should treat redirection of standard in in RHS of a pipe token as a syntax error.

(Recommended) Test your syntactic analysis with these commands:

INPUT LINE	VALID/INVALID
`cat < file1`	`Valid`
`cat > file1`	`Valid`

`cat < file1 > file2`	Valid
`cat > file1 < file2`	Valid
`cat file1 > file2`	Valid
`cat > file2 file1`	Valid
`cat file1 | grep main > result`	Valid
`cat < file1 | grep main > result`	Valid
`cat < file1 | grep main > result`	Valid
`< file1`	Invalid: Missing command name
`cat file1 <`	Invalid: Standard input redirection without file name
`cat file1 < | grep str`	Invalid: Standard input redirection without file name
`cat file1 >`	Invalid: Standard output redirection without file name
`cat file1 > | grep str`	Invalid: Standard output redirection without file name
`cat file1 > file2 > file3`	Invalid: Multiple redirection of standard out
`cat < file1 < file2`	Invalid: Multiple redirection of standard input
`cat file1 | cat < file2`	Invalid: Multiple redirection of standard input
`cat file1 > out  | grep str`	Invalid: Multiple redirection of standard out

(Required) Your program should be able to redirect the standard in stream and standard out stream to a file.

(Required) Your program should be able to print out errors to standard error stream.

(Required) Your program should create a file if the command's output is redirected to a non-existing file.

(Required) Your program should truncate a file if the command's output is redirect to an existing file.

Background process

(Recommended) Your program should treat '&' as special character in lexical analysis.

(Required) Your program should treat a background operator without command as a syntax error.

(Required) Your program should treat a background operator that is not located at the end of command line as a syntax error.

(Required) Your program should notify creation and termination of a background process to stdout with its pid.

(Required) Your program should bring the latest background process to the foreground when `fg` command is executed.

(Required) Your program should print out appropriate error messages to stderr for inappropriate usage or circumstances for & and `fg` commands.

(Recommended) Test your program's background processes functionality with sleep commands.

(Recommended) Test your syntactic analysis with these commands:

INPUT LINE	VALID/INVALID
`cat &`	`Valid`
`cat file1 &`	`Valid`
`&`	`Invalid: Missing command name`
`cat & cat`	`Invalid: Invalid use of background`