COP4600 - Operating System Design
The Shell Project
Spring 2015
Professor Sumi Helal

Due: Sunday April 12: 11:59PM

1 Overview

This team project involves writing a command interpreter for a Korn shell-like command language in C using Lex and Yacc running under UNIX. Your shell will parse command lines and execute the appropriate command(s). The core of the shell consists of simple commands, pipes, I/O redirection, environment variables, aliases, pathname searching, and wild-carding. You will do well on the lab if these features work well in your shell. For extra credit you may implement jtilde expansion and automated command completion. Each requirement in the core section will be explained in the rest of this handout.

You are *required* to use Lex and Yacc for this project.[1] You should not hand-build tokenizers or parsers in your implementation.

It would be in your best interest to start working early on this project. Only teams of two will be accepted. **Teaming should be finalized by 11:59PM Friday February 27, 2015.**

We will be grading your projects by running them against a set of test files that will exercise various features of the shell which we think are important. Our tests will be in addition to your own testing procedure.

Your project will be graded on correctness, completeness, your level of testing, and the organization of your code.

Monday recitation classes will cover the following to prepare you to carry on this project:

- Monday March 9: Lex and Yacc
- Monday March 16: A primer on related Unix commands
- Monday March 23: More on Lex and Yacc

[1] Or use flex/bison, or lex++/yacc++.

- Monday March 30: Open support session
- Monday April 6: Open support session

Monday sessions for April 13 and 20 will be used for grading, in addtion to other grading time that will be scheduled by the TA's. Each project will be graded by the TAs through a face-to-face session with the project team.

2 The Shell

Your shell will accept commands from standard input (terminal or files) and execute them. The type of commands your shell will accept are detailed below.

2.1 Shell Types

word We will use *word* to refer to a sequence of characters which are treated as a logical unit, sometimes referred to as a *token*. *Words* are separated by white space, newlines, and metacharacters. Any other character is valid in a *word*. If you wish a *word* to contain white space, then you must put double quotes around it. For example, your scanner should interpret `echo test > foo` as 3 words and one metacharacter. However, the command `echo "test > foo"` would be interpreted by the scanner as 2 words, `echo` and `test > foo` (note that the "" have been removed from `"test > foo"`).

white space White space consists of the characters space and tab.

metacharacters Metacharacters are characters which have special meaning to the shell, and stand only for themselves. Metacharacters cannot be part of a *word* unless they are preceded by a \ or are inside quotes. The following are metacharacters:

```
< > | " \ &
```

2.2 Built-in Commands

setenv variable word This command sets the value of the variable *variable* to be *word*.

printenv This command prints out the values of all the environment variables, in the format `variable=value`, one entry per line.

unsetenv variable This command will remove the binding of *variable*. If the variable is unbound, the command is ignored.

cd word_directory_name This command changes the current directory to the one named. You must handle *cd* with no arguments, to take you back to the home directory, i.e. it should have the same effect as *cd ~* (see Tilde Expansion).

alias name word Adds a new alias to the shell. See the subsection on aliases for more information.

unalias name Remove the alias for `name`.

bye Gracefully quit the shell. The shell should also exit if it receives the end of file.

2.3 Other Commands

```
cmd [arg]* [|cmd [arg]*]* [< fn1] [ >[>] fn2 ] [ 2>fn3 || 2>&1 ] [&]
```

Any command of this form can be accepted along with its arguments, pipes, and I/O redirection if present. Note that the I/O redirection can only appear at the end of the line in your shell. The construct `2>file` redirects the standard error of the program to `file`, while `2>&1` connects the standard error of the program to its standard output. If *cmd* does not contain a /, the shell must check the directories on the path that is the value of the environment variable PATH for the commands. Only check those directories that are on the path, i.e., if the current directory is not in the path, do not look in the current directory. Only if the file exists and is executable, it should be run. You must also be able to handle piping and I/O redirection on builtin commands. If `&` exists at the end of the line, then the shell will execute this command in the backgound. If `&` doesn't exist, then the shell will wait for this command to finish.

2.4 Aliases

You must implement a simplified version of the ksh alias mechanism. Your version will consist only of simple string substitutions, and will not provide any rearrangement of the arguments. The commands are of the following types:

alias The `alias` command with no arguments lists all of the current aliases.

alias name word This `alias` command adds a new alias to the shell. An alias is essentially a shorthand form of a long command. For example, you may have an alias `alias lf "/bin/ls -F"` set up so that whenever you type `lf` from the

command line, the command that is executed is `/bin/ls -F`. Note that alias expansion is only performed on the *first* word of a command. However, aliases may be nested. Your shell has to detect when an infinite alias expansion occurs.

unalias name The `unalias` command is used to remove the alias for `name` from the alias list.

2.5 Environment Variable Expansion ${variable}

It is also possible to include environment variables as part of words inside command lines. The shell reads all the characters from *${* to the next } and assumes it is the name of a variable. The value, if any, of the variable is substituted.

2.6 Wildcard Matching

Many shells do filename generation with wildcarding. You will implement a subset of the functionality found in most shells. Before a command is executed, each command word should be scanned for the characters `*` and `?`. If one of these characters appears the word is regarded as a pattern. The word is replaced with alphabetically sorted filenames that match the pattern. If no filename is found that matches the pattern, the word is left unchanged.

A `*` matches any string, including the null string. A `?` matches any single character. The character '.' at the start of a filename or immediately following a '/', as well as the character '/' itself, must be matched explicitly.

2.7 Examples of commands

```
setenv PATH .:/usr/bin:/usr/local/bin:~ghi/bin:~/bin
setenv ARGPATH .:~ghi/bin:~/bin
cd ./bin
cd ~/bin
cd ~sjc/bin
cd ../misc/old
cd src/proj/first
ls project1
ls "~project1"
wc -l f1 f2 f3 | sort | page
command1 arg1 arg2 | command2 | command3 < file_in > file_out 2>&1 &
alias lo  logout
alias rot13 "tr a-zA-Z n-za-mN-ZA-M"
```

```
rot13 < foo > bar
ls *.c foo.?
alias lo jj
alias jj "ls -al"
lo
setenv this .
setenv lsthis "jj ${this}"
${lsthis}
bye
setenv LIB ~/bin
nm ${LIB}/libxc.a
```

A note about the value of PATH:

Your shell should interpret the value of PATH to be a list of colon-separated words. Your shell should reparse and do tilde expansion at the beginning (first character) of each of these words whenever the value of the variable is reset.

You must be able to support aliasing in combination with I/O redirection.

3 Helpful Hints

In order to parse the input line, you should use Lex and Yacc (documentation will be provided). A suggested scheme is for the parser to build a table which contains for each command: the command name, a count of its arguments, a pointer to a list of null terminated arguments, an input file name, and an output file name. (Note that you *may* not want to build this particular table if the command is a built-in command.) If a table is produced (i.e. the command is syntacticly correct), then the shell should set up the pipes and I/O redirection as indicated by the table, and fork and exec processes as needed, wait for children to exit, report any errors detected and repeat the process for the next command line. Built-in commands should be taken care of inside the shell program. Errors should be handled gracefully, and the offending line number should be printed out if the session is not interactive. In other words, the shell should never die, crash or exit unexpectedly. If an error is detected, a message should be printed, the current command purged, and the prompt displayed for the next command. You may use neither `execlp` nor `execvp` within your shell, and you should not fork a shell (any of ksh, csh, sh, etc.) nor use the `system` nor `popen` calls to run your commands.

You should implement only the features described in this assignment handout for your shell, not all the features of existing shells such as csh or ksh.

4 Extra Credit

4.1 Tilde Expansion

~name should be replaced with the home directory of user *name*.

~ when not followed by a user name should be replaced by the home directory of the current user.

You should only do tilde expansion at the beginning of a word. The rule for tilde expansion can be summarized as follows: find the substring starting with the character after the ~ and ending with either the end of the string or a /, whichever comes first. If this substring is null, use the value of the HOME environment variable, else look up the substring in /etc/passwd using `getpwnam()` and extract the user's home directory from the returned struct. You should not do tilde expansion inside quoted strings.

4.2 File Name Completion

When typing a command to be executed by your shell, your shell should complete a partially-typed filename or user name. If the word immediately preceding the cursor expands to an unambiguous filename (with the current directory providing the default context) your shell should do the expansion when the ESC character is typed. When the last (partial) word begins with a tilde, your shell should complete it with a user name instead. For example

cd ~russ ESC

expands to

cd /homes/russo

5 Helpful Unix Commands

```
csh
open
close
dup (and dup2)
access
fork
execl
```

```
execve
chdir
ioctl
pipe
strings
perror
malloc
exit (and _exit)
getenv
getpwnam
environ(5v)
getpgrp
setpgrp
termio
wait
kill
```

Do not procrastinate! This is a time-consuming project, and it will take the full time to complete.

6 Turning in Your Shell

You will submit this project through Sakai. You must prepare a Makefile that will generate an executable called "shell" when the command `make` is typed in the directory containing all your sources and the Makefile. You must also have a README file that *clearly* specifies the features that you have NOT implemented followed by a list of features that you have implemented. The first feature you must try to implement in your shell is the I/O redirection from and to files. This is because we will be testing your project by running it through a program that will invoke your shell with its standard input redirected to come from each of several test files. Should this feature not work well you will earn our wrath for making testing your shell a grading nightmare.