Programming Project: small shell

Introduction

A Unix shell is a tool that allows users to manage processes. Using a shell, a person can run programs, control input and output, and create pipelines.

Unix shells do more than launch programs. Most shells are comprehensive programming languages. The shell programming language includes variables (local and global), functions, and a full range of control flow syntax, including *if* statements and *while* loops. Programs written in a shell programming language are called shell scripts.

The sample shell in the text implements some of the features of a typical shell. It runs programs, it allows some control of environment variables, has a limited if..then structure, and it does not allow one to use the shell variables in statements.

For this assignment, you will build on ideas and code from the text to create a more sophisticated shell. By adding mechanisms to input and retrieve shell variables, by extending the 'if' statement, and by allowing the shell to read scripts in addition to standard input, you will give the shell enough structure to be programmed.

The Big Picture

In its simplest form, a shell executes programs on request. You type the name of a program, the shell creates a new process, and then the shell runs the requested program in the new process. When the program ends, the shell wakes up and awaits the next request.

The main loop for a shell is:

```
setup();
while ( (cmdline = next_cmd(prompt, stdin)) != NULL ){
        if ( (arglist = splitline(cmdline)) != NULL  ){
                process(arglist);
                freelist(arglist);
        }
        free(cmdline);
}
```

The Story So Far

In the shells presented in class (psh1, psh2, and smsh1), 'setup' consists of initializing the list of shell variables and setting signal handling. The 'next_cmd' function reads a command from standard input. 'splitline' splits the input line into white-space separated strings. The 'process' operation acts on the list of arguments. If the command is a built-in command, the shell calls an internal function to do the work. If the command is not built-in, then the shell tries to fork() and execvp() it. If the command is part of the 'if..then' control structure, the shell determines if it needs to execute the code at all.

Where to Go from Here

There are lots of ways to expand a shell. Just as there are several styles of word processing programs, there are several styles of shells, each with unique features and designs. You do not have to stick with the design provided; sometimes you have to rearrange components or change the basic elements. You may select any method that makes sense to you and meets the requirements of the assignment.

For this assignment, you will write a shell that includes the features from the example in class plus specific additions. The specific additions you will make are:

1) *Alternate Input* Smsh1 reads commands from standard input. This is the interactive mode: the user types a command, the shell executes it. The real shell can also read commands from a file. This is the scripted mode: the user prepares a file that contains commands and then passes the name of that 'script file' to the shell as a command line argument. For example: `smsh file.of.commands`

   Your program, smsh2, should check the command line. If there is an argument, your smsh2 should use that file as a shell script. When reading from a script, your shell should not print prompts before each line.

2) *The cd built-in* The shell executes the change-directory command (cd) directly; it does not call a program to do this. Be sure to understand why the shell cannot call a program to do this. Then add cd to smsh2.

3) *The exit built-in* The exit command causes the shell to exit. If exit is given a numeric argument, the shell passes the value of that argument to the exit() system call. The exit command is particularly useful in shell scripts. Add this feature to your shell.

4) *Input* Implement a new built-in command 'read' that works sort of like the Bourne shell read command. The read command takes one argument: the name of the variable into which the input is read. The shell reads one line from standard input (not from the script) and stores that line in the variable. It therefore similar to C's gets() function, Perl's <STDIN>, and BASIC's input command. The Unix shell allows the read command to accept several variables and assigns strings in the input to separate variables. You may add that operation if you like.

5) *Variable Substitution* Consider this fragment: `DIR=/usr/bin ; ls $DIR $DIR.old` it assigns a string to the variable DIR and then uses that variable in another command. The program 'ls' sees the arguments /usr/bin and /usr/bin.old, not the strings $DIR and $DIR.old. Add this feature to smsh2. Where does this substitution take place? During reading in the line? During parsing the line? The way you add variable substitution to your program affects the design of your input and parsing sections.

   Think through this carefully. When the shell sees a dollar sign, it reads the following characters as the name of a variable. How does it know where the variable name ends? The shell looks for upper and lower case letters, digits, and the underscore. Any sequence of those characters is a legal variable name, as long as the first character is not a digit. Thus, in `$DIR.old` the variable name is `DIR` .

   Your program, therefore, will consider the variable name to end when it encounters any character not in this set. Make sure you modify the assignment statement so that the user is not allowed to create variables with illegal characters.

6) *Quoted Characters* What if you want to include a real dollar sign in your command, and you do not want your shell to try to substitute a variable value? As soon as you introduce special characters, you need to create a mechanism for un-specialing them. The Unix tradition is to use the backslash in front of each special character that you want treated 'as is.' Add this feature to your shell. More sophisticated quoting (using " and ') is not required.

7) *Nested if commands* The hallmark of a programming language is the if command. The example in the text handles the if, then, and fi parts of the structure, but lacks the 'else' part and cannot do nested if's. The shell should accept:

```
echo Name to look up\?
read NAME
if grep $NAME $HOME/phonelist
then
      echo there is the name
else
      echo cannot locate $NAME
      if grep -i $NAME $HOME/phonelist
      then
          echo but a case-insensitive search succeeds
      fi
fi
```

The word *if* is followed by a command and its arguments. The shell runs the command normally and keeps track of the exit status of the command. If the command exits with status 0, the lines after the 'then' line are executed by the shell. If the command exits with non-zero status, then the lines after 'else' (if there be any) are executed. The if command is terminated by a line consisting of the word 'fi'. The other keywords 'then' and 'else' must appear on their own lines. The else clause is optional. The shell must support any depth of nested if's.

Getting Started

Various supporting documents and code are in ˜COURSE/hw/smsh . Copy what you want to play with to your account and test out the shell. Start with any version of smsh from the lecture samples. Add chdir() just to get started. It is not tricky.

Think about variable substitution. There are two popular approaches to this problem.

One method reads the entire input line into an array then calls a function to copy characters from that array to a new one, replacing each instance of $var with its value. This translation program also takes care of backslashes.

The other method performs variable substitution as part of a wrapper to getc(). Usually, getc() returns the next character from the current input stream. Note, to support item (1), your shell will not just read from stdin. It will read from stdin or from a script. This getc wrapper will read the next character. If it is a boring character, that is neither backslash nor dollar sign, the function will pass that character to its caller.

On the other hand, if the character is a backslash or dollar sign, the function will need to do something fancier in order to return the next character to its caller.

Regardless of how you handle variable substitution, you need to make sure you are splitting the input stream into separate arguments.

Plan out the 'if' command implementation. One approach is to read the line from 'if' to 'fi' into a tree-like data structure in memory and then traverse the tree deciding which branches to take by executing statements. Another approach is to extend the state model from the text with a stack of return values or recursive processing of input.

Your shell is a subset of the standard Bourne shell (sh), so test how things are supposed to work under sh.

What to Turn In

Full source, Makefile, a clean gcc -Wall, and a sample run. You *must* test the script provided in ˜COURSE/hw/smsh/test.smsh . Other samples of its execution are welcome.