CS 252 Lab3: Shell

CS252
Lab 3 - Implementing a Shell

FAQ | Part3Notes | Rubric | Code Review forms

NOTE: Text in green indicates extra credit.

Introduction

The goal of this project is to build a shell interpreter like `csh`. The project has been divided into parts. Some skeleton code has been provided, so you will not be starting from scratch.

Part 1: Lex and Yacc

To begin, you will write a scanner and parser for your shell.

Getting started

Login to a CS department machine (a lab machine or `data.cs.purdue.edu`), navigate to your preferred directory, and run

```
cd
cd cs252
git clone /homes/cs252/sourcecontrol/work/$USER/lab3-src.git
cd lab3-src
```

Build the shell by typing `make`, and start it by typing `./shell`. Type in some commands, for example

```
ls -al
ls -al aaa bbb > out
```

At this point, the shell does not have much implemented; notice what happens if you try to use some shell features that you used in Lab 2. For example, try redirecting input or editing a typo in a command.
Look through the skeleton code and try to understand how it works. First, read the `Makefile` to understand how the program is built; notice that it is mostly written in C++. The file `command.h` implements a data structure that represents a shell command. The struct `SimpleCommand` implements an argument list for a

simple command (i.e. a command of the form `mycmd arg1 arg2 arg3`). When pipes are used, a command will be composed of multiple `SimpleCommand`s. The struct `Command` represents a list of simple commands. Additionally, `Command` has fields for input, output, and error redirection filenames.
Since we are using C++, feel free to modify the skeleton code to use C++ types such as string, vector, map, etc. In fact, you may find that doing so eases the memory management difficulty of this lab.

Accepting more complex commands

You will use Lex and Yacc to implement the grammar of your shell. See here or here for a tutorial on Lex and Yacc.
The skeleton shell implements a very limited grammar:

```
cmd [arg]* [> filename]
```

The objective of Part 1 is to modify `shell.l` and `shell.y` to implement a more complex grammar:

```
cmd [arg]* [ | cmd [arg]* ]* [ [> filename] [< filename] [ >&
filename] [>> filename] [>>& filename] ]* [&]
```

Insert the necessary code in `shell.l` and `shell.y` to fill in the `Command` struct. Make sure that the `Command` struct is printed correctly.
Some example commands to test with are:

```
ls
ls -al
ls -al aaa bbb cc
ls -al aaa bbb cc > outfile
ls | cat | grep
ls | cat | grep > out < inp
ls aaaa | grep cccc | grep jjjj ssss dfdffdf
ls aaaa | grep cccc | grep jjjj ssss dfdffdf >& out < in
httpd &
ls aaaa | grep cccc | grep jjjj ssss dfdffdf >>& out < in
```

Submission

The deadline of Part 1 is Monday September 26th at 11:59 pm.
To turn in Part 1:

1. Login to a CS department machine
2. Navigate to your `lab3-src` directory
3. Run `make clean`

4. Run `make` to check that your shell builds correctly
5. Run `make clean`
6. Run `cd ..`
7. Run `turnin -c cs252 -p lab3-1 lab3-src`

Part 2: Executing commands

In Part 2, you will implement execution of the simple commands, file redirection, piping, and waiting or not waiting for commands to end.

2.1: Simple command process creation and execution

For each simple command, create a new process using `fork()` and call `execvp()` to execute the corresponding executable. If the `Command` is not set to execute in the background, then your shell will have to wait for the last simple command to finish using `waitpid()`. Refer to the `man` pages of these functions for information on their arguments and return values. Additionally, `cat_grep.cc` is a program that creates processes and performs redirection.
After you have completed Part 2.1, you should be able to execute commands such as:

```
ls -al
ls -al /etc &
```

2.2: File redirection

If the the `Command` has input, output, or error redirection files set, then create files as appropriate and use `dup2()` to redirect file descriptors 0, 1, or 2 (input, output, or error, respectively) to/from the files as appropriate. See the example redirection in `cat_grep.cc`.
After you have completed Part 2.2, you should be able to execute commands such as:

```
ls -al > out
cat out
ls /tttt >& err
cat err
cat < out
cat < out > out2
cat out2
ls /tt >>& out2
```

Note:

- `>&` file redirects both stdout and stderr to file
- `>>&` file appends both stdout and stderr to file
- `>>` file appends stdout to file

2.3: Pipes

Use `pipe()` to create pipes that will redirect the output of one simple command to the input of the next simple command. Use `dup2()` to do the redirection. See the example piping in `cat_grep.cc`.
After you have completed Part 2.2, you should be able to execute commands such as:

```
ls -al | grep command
ls -al | grep command | grep command.o
ls -al | grep command
ls -al | grep command | grep command.o > out
cat out
```

Testing

Much of your shell will be graded using automatic testing, so make sure that your shell passes the provided tests. Your grade for this lab will depend on the number of tests that pass. The tests are for Part 2 and Part 3 of the project.
See `lab3-src/test-shell/README` for an explanation of how to run the tests. The tests will also give you an estimated grade. This grade is just an approximation. Other tests not given to you will be used as well during grading.

Submission

The deadline of Part 2 is Monday October 3rd at 11:59 pm.
To turn in Part 2:

1. Login to a CS department machine
2. Navigate to your `lab3-src` directory
3. Run `make clean`
4. Run `make` to check that your shell builds correctly
5. Run `make clean`
6. Run `cd ..`
7. Run `turnin -c cs252 -p lab3-2 lab3-src`

Part 3: Ctrl-C, Wildcards, Zombie Elimination, etc.

In Part 3, you will add features to make your shell more useful and complete with respect to `csh`.

3.1: Ctrl-C

In `csh`, `bash`, etc., you can type Ctrl-C to stop a running command. In particular, a `SIGINT` signal is generated that kills the program. Additionally, if Ctrl-C is typed when no command is running, the current prompt is discarded and a fresh prompt is printed. As-is, your shell will simply exit when Ctrl-C is typed and no command is running. Make your shell behave as `csh` does with respect to Ctrl-C. See ctrl-c.cc for an example of ignoring SIGINT. Also see the `man` page for `sigaction`.

3.2: Exit

Implement a special command called `exit` which will exit the shell when run. Note that `exit` should not cause a new process to be created. Also, make your shell print a goodbye message like so:

```
myshell> exit

 Good bye!!

bash$
```

3.3: Builtins

Certain commands you can run in `csh` or `bash` do not actually correspond to executables. Implement the following builtin commands:

`printenv`	Prints the environment variables of the shell. The environment variables of a process are stored in the variable `char **environ;`, a null-terminated array of strings. Refer to the `man` page for `environ`.
`setenv A B`	Sets the environment variable `A` to value `B`. See article.
`unsetenv A`	Un-sets environment variable `A`
`cd A`	Changes the current directory to `A`. If no directory is specified, default to the home directory. See the `man` page for `chdir`.
`jobs`	Prints information about commands running in the background. See the `man` page.
`fg x`	Brings job with ID x into the foreground. See the `man` page.
`bg x`	Resumes a suspended background job. See the `man` page.

`source A`	Runs file `A` line-by-line, as though it were being typed into the shell by a user

You should be able to use the builtins like any other command, including with redirection and piping.

3.4: Wildcarding

In most shells, including `bash` and `csh`, you can use `*` and `?` as wildcard characters in file and directory names. You can try wildcarding in `csh` to see the results. The way you will implement wildcarding is the following. First do the wild carding only in the current directory. Before you insert a new argument in the current simple command, check if the argument has wild card (`*` or `?`). If it does, then insert the file names that match the wildcard including their absolute paths. Use `opendir` and `readdir` to get all the entries of the current directory (check the `man` pages). Use the functions `regcomp` and `regexec` to find the entries that match the wildcard. Check the example provided in `regular.cc` to see how to do this. Notice that the wildcards and the regular expressions used in the library are different, so you will have to convert from the wildcard to the regular expression. The "*" wildcard matches 0 or more non-blank characters, except "." if it is the first character in the file name. The "?" wildcard matches one non-blank character, except "." if it is the first character in the file name. Once that wildcarding works for the current directory, make it work for absolute paths.
IMPORTANT: Do not use the glob() call. You must use the functions discussed above.

3.5: Zombie Elimination

You will notice that in your shell the processes that are created in the background become *zombie* processes. That is, they no longer run, but wait for the parent to acknowledge that they have finished. Try doing in your shell:

```
ls &
ls &
ls &
ls &
/bin/ps -u <your-login> | grep defu
```

The zombie processes appear as "defu" in the output of the last command.
To cleanup these processes you will have to set up a signal handler, like the one you used for ctrl-c, to catch the SIGCHLD signals that are sent to the parent when a child process exits. The signal handler will then call `wait3()` to cleanup the zombie child. Check the man pages for `wait3` and `sigaction`. The shell should print the process ID of the child when a process in the background exits in the form "[PID] exited."

3.6: Quotes

Allow quotes in your shell. It should be possible to pass arguments with spaces if they are surrounded by quotes. For example:

```
myshell> ls "command.cc Makefile"
command.cc Makefile not found
```

`"command.cc Makefile"` is only one argument. Remove the quotes before inserting the argument. No wildcard expansion is expected inside quotes.

3.7: Escaping

Allow the escape character. Any character can be part of an argument if it comes immediately after `\`. For example:

```
myshell> echo \"Hello between quotes\"
"Hello between quotes"
myshell> echo this is an ampersand \&
this is an ampersand &
```

3.8: Environment variable expansion

You will implement environment variable expansion. When a string of the form `${var}` appears in an argument, it will be expanded to the value that corresponds to the variable `var` in the environment table. For example:

```
myshell> setenv A Hello
myshell> setenv B World
myshell> echo ${A} ${B}
Hello World
myshell> setenv C ap
myshell> setenv D les
myshell> echo I like ${C}p${D}
I like apples
```

Additionally, the following special expansions should be implemented:

`${$}`	The PID of the shell process
`${?}`	The return code of the last executed simple command (ignoring commands sent to the background)
`${!}`	PID of the last process run in the background
`${_}`	The command line of the last executed command

`${SHELL}`	The path of your shell executable (`argv[0]` in `main`)

3.9: Tilde expansion

When the character "~" appears itself or before "/" it will be expanded to the home directory. If "~" appears before a word, the characters after the "~" up to the first "/" will be expanded to the home directory of the user with that login. For example:

`ls ~`	-- List the ~~current~~ home directory
`ls ~george`	-- List george's ~~current~~ home directory
`ls ~george/dir`	-- List subdirectory "dir" in george's directory

3.10: isatty()

When your shell uses a file as standard input your shell should not print a prompt. This is important because your shell will be graded by redirecting small scripts into your shell and comparing the output. Use the function `isatty()` to find out if the input comes from a file or from a terminal.

3.11: Edit mode

Copy the file `/homes/cs252/Fall2016/lab3-shell-x86-new/lab3-src-kbd.tar.gz` that contains the code that you will need to change the terminal input from canonical to raw mode. In raw mode you will have more control of the terminal, passing the characters to the shell as they are typed. Copy the contents of the directory `lab3-src-kbd/*` to your `lab3-src/`. To build the examples type:

```
make -f Makefile.kbd
```

This will generate two executables: keyboard-example and read-line-example. Run `keyboard-example` and type letters from your keyboard. You will see the corresponding ascii code immediately printed on the screen.
The other program read-line-example is a simple line editor. Run this program and type ctrl-? to see the options of this program. The up-arrow is going to print the previous command in the history. The file tty-raw-mode.c switches the terminal from canonical to raw mode. The file read-line.c implements the simple line editor. Study the source code in these files.

To connect the line editor to your shell add the following code to `shell.l` after the `#include` lines:

```
    %{

    #include <string.h>
```

```
#include "y.tab.h"

////////////  Start added code ///////////

char * read_line();

int mygetc(FILE * f) {
static char *p;
char ch;

if (!isatty(0)) {
  // stdin is not a tty. Call real getc
  return getc(f);
}

// stdin is a tty. Call our read_line.

if (p==NULL || *p == 0) {
  char * s = read_line();
  p = s;
}

ch = *p;
p++;

return ch;
}

#undef getc
#define getc(f) mygetc(f)

/////////// End added code ///////////

%}

%%
```

Now modify your Makefile to compile your shell with the line editor. Copy the entries in `Makefile.kbd` that build `tty-raw-mode.o` `read-line.o` to your `Makefile` and add these object files to your shell.

Now modify `read-line.c` to add the following editor commands:

- left arrow key: Move the cursor to the left and allow insertion at that position. If the cursor is at the beginning of the line it does nothing.
- right arrow key: Move the cursor to the right and allow insertion at that position. If the cursor is at the end of the line it does nothing.
- delete key(ctrl-D): Removes the character at the cursor. The characters in the right side are shifted to the left.
- backspace (ctrl-H)key: Removes the character at the position before the cursor. The characters in

the right side are shifted to the left.

- Home key (or ctrl-A): The cursor moves to the beginning of the line
- End key (or ctrl-E): The cursor moves to the end of the line

3.12: History

With the line editor above also implement a history list. Currently the history is static. You need to update the history by creating your own history table. Implement the following editor commands:

- Up arrow key: Shows the previous command in the history list.
- Down arrow key: Shows the next command in the history list.

3.13: Subshells

Any argument of the form `` `command and args` `` will be executed and the output will be fed back into the shell. The character `` ` `` is called "backtick". For example:

- ``echo `expr 1 + 1` `` will be substituted by `echo 2`
- ``echo a b > dir; ls `cat dir` `` will list the contents of directories a and b

You will implement this feature by

1. Scanning in `shell.l` the command between backticks.
2. Calling your own shell in a child process passing this command a input. You will need two pipes to communicate with the child process. One to pass the command to the child, and one to read the output of the child.
3. Reading the output of the child process and putting the characters of the outptut back into the scanner buffer using the function yy_unput(int c) in reverse order. See the FAQ for more details.

3.14: Path completion

Implement path completion. When the <tab> key is typed, the editor will try to expand the current word to the matching files similar to what tcsh and bash do.

3.15: Variable prompt

Whenever your shell would print `myprompt>` , if there is an environment variable called `PROMPT`, it should print that instead. Additionally, if there is an environment variable called `ON_ERROR`, the shell should print its value whenever the last simple command in a command exits with a nonzero code.

3.16: .shellrc

When your shell starts, it should attempt to do the equivalent of running `source .shellrc`.

3.17: Ctrl-Z

Usually in a terminal, you can type Ctrl-Z to suspend the currently running process and send it to the background. Implement a signal handler as with Ctrl-C that intercepts this signal so that your shell will not be suspended.

IMPORTANT: There are no automatic tests for the line editor so it will be tested manually by the TAs. Make sure that you update the ctrl-? output correctly with the commands you have added. The items 3.11 and 3.12 will count for 10% of the total grade of the shell.

Submission

The deadline of Part 3 is Friday October 14th at 11:59 pm.

Add a `README` file to the lab3-src/ directory with the following:

1. Features specified in the handout that work
2. Features specified in the handout that do not work
3. Extra features implemented

To turn in Part 3:

1. Login to a CS department machine
2. Navigate to your `lab3-src` directory
3. Run `make clean`
4. Run `make` to check that your shell builds correctly
5. Run `make clean`
6. Run `cd ..`
7. Run `turnin -c cs252 -p lab3-3 lab3-src`