The Shell

The Frequently Asked Questions file is available here.

Something else that may be helpful for part3 is available here.

New tests test-shell-new.tar.Z Download them in your lab3-src and type "uncompress test-shell-new.tar.Z; tar -xvf test-shell-new.tar". Then type "cd test-shell-new; ./testall".

Introduction The goal of this project is to build a shell interpreter like csh. The project has been divided in several parts. Some sources are being provided so you don't need to start from scratch.

Using the Debugger

It is important that you learn how to use a debugger to debug your C and C++ programs. If you spend a few hours learning how to use gdb, it will save you a lot of hours of development in this and future labs.

To start gdb type "*gdb program*". For example, to debug your shell type:

```
csh> gdb shell
```

Then type

```
(gdb) break main
```

This will make the debugger stop your program before main is called. In general, to set a breakpoint in a given function type "*break <function-name>*"

To start running your program type:

```
(gdb) run
```

Your program will start running and then will stop at *main*.

Use "*step*"or "*next*" to execute the following line in your program. "*step*" will execute the following line and if it is a function, it will step into it. "*next*" will execute the following line and if it a function it will execute the function.

```
(gdb) next      - Executes following line. If it is a function it will execute the function and return.
or
(gdb) step      - Executes following line. If it is a function it will step into it.
```

An empty line in gdb will rerun the previous gdb command.

Other useful commands are:

```
print var    - Prints a variable
where        - Prints the stack trace
quit         - Exits gdb
```

For a more complete tutorial on gdb, google "gdb tutorial".

First part: Lex and Yacc

In this part you will build the scanner and parser for your shell.

- Download the tar file lab3-src.tar.gz, that contains all the files in lab3-src, to your home directory and untar it using the following command:

  ```
  gzip -dc lab3-src.tar.gz | tar xvf -
  ```
- Build the shell program by typing :

  ```
  make
  ```

  To run it type:

  ```
  shell
  ```

  Then type commands like

  ```
  ls -al
  ```

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

Check the output printed

- Try to understand how the program works. First read the Makefile to learn how the program is built. The file command.h implements the data structure that represents a shell command. The struct *SimpleCommand* implements the list of arguments of a simple command. Usually a shell command can be represented by only one *SimpleCommand*. However, when pipes are used, a command will consist of more than one *SimpleCommand*. The struct *Command* represents a list of *SimpleCommand* structs. Other fields that the *Command* struct has are *_outFile*, *_inputFile*, and *_errFile* that represent input, output, and error redirection.

- Currently the shell program implements a very simple grammar:

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

You will have to modify shell.y to implement a more complex grammar

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

- Insert the necessary actions in shell.y to fill in the *Command* struct. Make sure that the *Command* struct is printed correctly.

- Run your program against the following commands:

```
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
```

- The deadline of this part of the project is Thursday February 14th. 2013 at 11:59pm.

at 11:59pm. Follow these instructions to turnin your part one.

1. Login to lore.
2. cd to lab3-src and type "make clean"
3. Type "make" to make sure that your shell is build correctly.
4. Type "make clean" again.
5. cd one directory above lab3-src
6. Turnin your project by typing:

   turnin -c cs252 -p lab3-1 lab3-src

7. Verify that you have turned in your files correctly by typing:

   turnin -c cs252 -p lab3-1 -v

Here are some tutorials on Lex and Yacc .

Second part: Process Creation, Execution, File Redirection, Pipes, and Background

Starting from the command table produced in Part 1, in this part you will execute the simple commands, do the file redirection, piping and if necessary wait for the commands to end.

1. For every simple command create a new process using *fork()* and call *execvp()* to execute the corresponding executable. If the _bakground flag in the Command struct is not set then your shell has to wait for the last simple command to finish using *waitpid()*. Check the manual pages of *fork()*, *execvp()*, and *waitpid()*. Also there is an example file that executes processes and does redirection in cat_grep.cc. After this part is done you have to be able to execute commands like:

```
ls -al

ls -al /etc &
```

2. Now do the file redirection. If any of the input/output/error is different than 0 in the Command struct, then create the files, and use *dup2()* to

redirect file descriptors 0, 1, or 2 to the new files. See the example cat_grep.cc to see how to do redirection. After this part you have to be able to execute commands like:

```
ls -al > out

cat out

ls /tttt >& err

cat err

cat < out

cat < out > out2

cat out2

ls /tt >>& out2
```

">& file" redirects both stdout and stderr to file. ">>& file" append both stdout and stderr to file.

">> file" appends stdout to file.

3. Now do the pipes. Use the call *pipe()* to create pipes that will interconnect the output of one simple command to the input of the next simple command. use *dup2()* to do the redirection. See the example cat_grep.cc to see how to construct pipes and do redirection. After this part you have to be able to execute commands like:

```
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
```

4. The deadline of this part of the project is February 21st, 2013 at 11:59pm. You will use the following command from lore.

```
turnin -c cs252 -p lab3-2 lab3-src
```

Where lab3-src contains the sources of your shell. Type "make clean" before you turn in your files.

Testing your Shell:

Your shell will be graded using automatic testing, so make sure that the tests given to you run.

Your grade for this project will depend on the number of tests that pass. The tests given are for part 2 and 3 of the project.

See the file lab3-src/README for an explanation on 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.

Third part: Control-C, Wild Cards, Elimination of Zombie processes, etc.

This is the final part of your shell. You will include some more features to make your shell more useful.

1. Your shell has to ignore ctrl-c. When ctrl-c is typed, a signal SIGINT is generated that kills the program. There is an example program in ctrl-c.cc that tells you how to ignore SIGINT. Check the man pages for *sigset*.
2. You will also have to implement also an internal command called *exit* that will exit the shell when you type it. Remember that the *exit* command has to be executed by the shell itself without forking another process.

```
myshell> exit

  Good bye!!

csh>
```

3. You will implement an internal command *printenv* to print the environment variables of the shell. The environment variables of a process are stored in the variable *environ*. *Environ* is a null terminated array of strings.

```
char **environ;
```

Check the man pages of environ.

4. Now do the wildcarding. The wildcarding will work in the same way that it works in shells like csh. The "*" character matches 0 or more nonspace characters. The "?" character matches one nonspace character. The shell will expand the wildcards to the file names that match the wildcard where each matched file name will be an argument.

```
echo *          // Prints all the files in the current directory
```

```
echo *.cc       // Prints all the files in the current

                // director that end with cc

echo c*.cc

echo M*f*

echo /tmp/*     // Prints all the files in the tmp directory

echo /*t*/*

echo /dev/*
```

You can try this wildcards in csh to see the results. The way you will implement wild carding 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 *compile* and *advance* to find the entries that match the wildcard. Check the example file provided in regular.cc to see how to do this. Notice that the wildcards and the regular expressions used in the library are different and 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 "?" wildcar 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 have to use the functions opendir(), readir(), compile() and advance() to implement wildcards.

5. 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 them 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 "ps -u <your-login>" command.

To cleanup these processes you will have to setup 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 *sigset*. The shell should print the process ID of the child when a process in the background exits in the form "[PID] exited."

6. Implement the builtin command *setenv A B* this command sets the environment variable *A* to be *B*. Check man pages for function putenv().

7. Implement the builtin command *unsetenv A*. This command removes environment variable *A* from the environment.

8. Implement the *cd [dir]* command. This command changes the current directory to *dir*. When *dir* is not specified, the current directory is changed to the home directory. Check "man 2 chdir".
9. Extend lex to support any character in the arguments that is not a special character such as "&", ">", "<", "|" etc. Also, your shell should allow no spaces between "|", ">" etc. For example, "lslgrep a" without spaces after "ls" and before "grep" should work.

10. Allow quotes in your shell. It should be possible to pass arguments with spaces if they are sorrounded between quotes. E.g.

```
ls "command.cc Makefile"

command.cc Makefile not found
```

"command.cc Makefile" is only one argument.

Remove the quotes before inserting argument. No wild-card expansion is expected inside quotes.

11. Allow the escape character. Any character can be part of an argument if it comes immediately after \. E.g.

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

12. 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. E.g

```
setenv A hello
```

```
setenv B world
echo ${A} ${B}
Hello World

setenv C ap
setenv D le
echo I like ${C}p${D}
I like apple
```

13. 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 directory
ls ~george     -- List george's current directory
ls ~george/dir -- List subdirectory "dir" in george's directory
```

14. 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.
15. Edit mode. Download the file 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 copmmand 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 sourcecode 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

16. History: With the line editor above also implement the 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.

17. 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

or

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 ouptut back in to the scanner buffer using the function yy_unput(int c) in reverse order. See the FAQ for more details.
18. **Extra Credit**: 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.

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 15 and 16 will count for 20% of the total grade of the shell.

Deadline

The deadline of this part of the project is Thursday March 7th, 2013 at 11:59pm,

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

Make sure that your shell can be built by typing "make".

You will use the following command from lore.

```
turnin -c cs252 -p lab3-3 lab3-src
```

Where lab3-src contains the sources of your shell. Type "make clean" before you turn in your files.

Type the followiing command to verify the files that you turned in:

```
turnin -c cs252 -p lab3-3 -v
```