CS252

Lab 3 - Implementing a Shell

FAQ | Additional Notes | Grading Form P1 | Grading Form P2 | Final Grading Form

---

Updates

Any changes that need to be made to the handout / lab will be mentioned here.

NOTE: Text in green indicates extra credit features.

Introduction

The goal of this project is to build a shell interpreter which combines behavior from common shells including bash and csh. The project has been divided into parts. Some skeleton code has been provided, so you will not be starting from scratch.

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
tar -xvf /homes/cs252/Spring2024/lab3-shell-x86-Spring2024-if-while/lab3-test-2.tar
git clone /homes/cs252/sourcecontrol/work/$USER/lab3-src.git
cd lab3-src
```

Notice that the lab3-test-2/ and lab3-src/ are different directories.

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.

Part 1: Parsing and Executing Commands

To begin, you will write a scanner and parser for your shell using the open source versions of Lex and Yacc (Flex and Bison). 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.hh` 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 `SimpleCommands`. The struct `Command` represents a list of simple commands. Additionally, `Command` has fields which allow the user to specify files to use for input, output, and error redirection.

Much of the provided code uses C style data structure; however, you may find it easier to manage the code by making use of C++ features. Feel free to modify the skeleton code to make better use of C++ types such as string, vector, map, etc. In fact, you may find that doing so eases the memory management difficulty of this lab significantly.

Part 1A: Lex and Yacc - Accepting more complex commands

You will use Lex and Yacc to implement the grammar of your shell. See here and here for tutorials on Lex and Yacc. Here is an updated manual for Flex

The skeleton shell initially implements only a very limited grammar:

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

The first objective for Part 1 is to modify `shell.l` and `shell.y` to support a more complex grammar:

```
cmd [arg]* [| cmd [arg]* ]* [ [> filename] [< filename] [2> 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 included in the table below:

```
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.txt
httpd &
ls aaaa | grep cccc | grep jjjj ssss dfdffdf >>& out < in.txt
```

Part 1B: Executing commands

Now you will implement the execution of simple commands, IO redirection, piping, and allowing processes to run in the background.

1B.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, we have provided the file `cat_grep.cc` as an example, which is a program that creates processes and performs redirection.

After you have completed Part 1B.1, you should be able to execute commands such as:

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

1B.2: File redirection

If the `Command` specifies files for IO redirection (of input, output, or error), then create those files as necessary. To change the file descriptors to point to the specified files, you will need to use `dup2()`. Note that file descriptors 0, 1, and 2 correspond to input, output, and error respectively. See the example redirection in `cat_grep.cc`.

After you have completed Part 1B.2, you should be able to execute commands such as:

```
ls -al > out
cat -q cat 2> dog
ls
cat out
ls /tttt >& err
cat err
cat < out
cat < out > out2
cat out2
ls /tt >>& out2
```

Note:

- `2>` the command redirects stderr to the specified file
- `>&` the command redirects both stdout and stderr to the specified file
- `>>` the command appends stdout to the specified file
- `>>&` the command appends both stdout and stderr to the specified file

1B.3: Pipes

Pipes are an interface that allow for inter-process communication. They have two ends, one for reading and one for writing. Data which is written into the write end of the pipe is buffered until it is read from the read end by another process.

Use `pipe()` to create a pipe that will redirect the output of one simple command to the input of the next simple command. You will again need to use `dup2()` to handle the redirection. See the example piping in `cat_grep.cc`.

After you have completed Part 1B.3, 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
```

1B.4: 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.

Note: due to how the automated tests are built, you will need to complete this portion of part 1 before your shell will pass any of the automated tests.

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 partially depend on the number of tests that pass. The tests provided will be used for each part of the project, so don't worry if you are unable to pass all of the tests after finishing part 1.

See `~/cs252/lab3-test-2/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 which are not provided will be used as well during official grading; some points will also be awarded based on a demo of your shell.

Submission

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 `git tag -f part1`*
6. *Run `git push -f origin part1`*
7. *Run `git show part1`*
8. *The `show` command should show the diff from the most recent commit*

Part 2: Signal Handling, More Parsing, and Subshells

In Part 2, you will begin to add features that make your shell more useful and fully featured.

2.1: Ctrl-C

In `csh`, `bash`, and other common shells, you can type Ctrl-C to stop a running command; this can be especially helpful if a command you are running takes longer to finish than expected or if you are running a buggy program that falls into an infinite loop. This is accomplished by generating a `SIGINT` signal which is passed on to the program currently being run. 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 detecting and ignoring a `SIGINT` signal. Also see the `man` page for `sigaction()`.

2.2: Zombie Elimination

Try running the following set of commands in the shell you have written:

```
ls &
```

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

The last command shows all processes that show up as "defu" (for “defunct”). Such processes are called *zombie processes*: they no longer run, but wait for the parent to acknowledge that they have finished. Notice that each of the processes that are created in the background become *zombie processes*.

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 finishes. The signal handler will then call `waitpid()` to cleanup the zombie child. Check the man pages for the `waitpid()` and `sigaction()` system calls. The shell should print the process ID of the child when a process in the background exits in the form "`[PID] exited`."

2.3: 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; it should be picked up by your shell during parsing and cause your shell to `exit`. Also, make your shell print a goodbye message, like so:

```
myshell> exit

 Good bye!!

bash$
```

2.4: Quotes

Add support for 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
```

Here, "`command.cc Makefile`" is only one argument. You will need to remove the quotes before using the argument they contain. Note: wildcard expansion will not be expected inside quotes in the next part of the lab.

2.5: Escaping

Allow the escape character. Any character can be part of an argument if it comes immediately after `\`, including special characters such as quotation marks (“”) and an ampersand (&). For example:

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

2.6: Builtin Functions

Certain commands you can run in `csh` or `bash` do not actually correspond to executables; much like the exit command implemented for part 2.3, these commands are detected by the shell during parsing to carry out certain special functions. 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`
`source A`	Runs file `A` line-by-line, as though it were being typed into the shell by a user. See Multiple Input Buffers or look at Flex manual
`cd A`	Changes the current directory to `A`. If no directory is specified, default to the home directory. See the `man` page for `chdir()`.

You should be able to use builtins like any other commands (e.g. grep, cat, etc.), including with redirection and piping.

2.7: Environment variable expansion

You will implement environment variable expansion. Recall that in the previous part of the lab, you allowed users to set and retrieve environmental variables using builtin functions. 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 are required to 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
`${_}`	The last argument in the fully expanded previous command Note: this excludes redirects
`${SHELL}`	The path of your shell executable. Hint: `realpath()` can expand a relative path to an absolute path. You can obtain the relative path to the shell in `argv[0]`

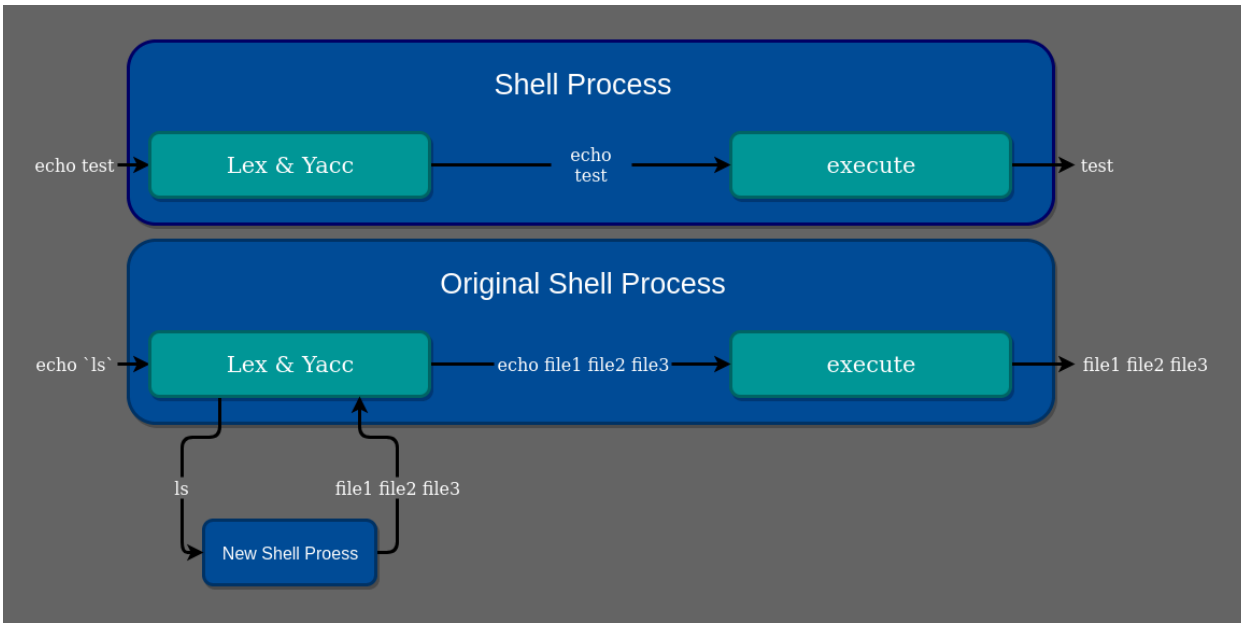
2.8: Creating a Default Source File: ".shellrc" (Extra credit)

When your shell starts, it should attempt to do the equivalent of running "`source .shellrc`". (This feature will be considered extra credit).

2.9: Subshells

Sometimes a user will need to run a complex command that uses the output from one shell command as the input of another. Any argument of the form `$(command and args)` will be processed by another shell (the subshell) which is executed as a child process and the output will be fed back into the original parent shell. For example:

- `echo $(expr 1 + 1)` will become `echo 2`
- `echo a b > dir; ls $(cat dir)` will list the contents of directories a and b



The example below further explains how your shell should interpret and processes commands with and without backticks:

```
myshell> echo test
```

Lex & Yacc parses the command and executes it normally

```
myshell> echo $(ls)
```

Lex & Yacc parses the command, but must evaluate the `ls` command before the echo command can be executed. Below is a step by step example of how a subshell command is processed.

```
myshell> echo $(ls) “and more”
file1 file2 file3 and more
```

1. A command containing a subshell command is passed to the shell

   ```
   Input buffer=echo $(ls) ”and more”
   Command Word=
      Arguments=
   ```

2. The shell parses the `echo` command normally.

   ```
   Input buffer=echo $(ls) ”and more”
   Command Word=echo
      Arguments=
   ```

3. The shell parses the subshell command `` `ls` ``

   ```
   Input buffer=echo $(ls) ”and more”
   Command Word=echo
      Arguments=
   ```

4. After executing the command in the subshell the input is injected at the head of the buffer

   ```
   Input buffer=echo $(ls) file1 file2 file3 ”and more”
   Command Word=echo
      Arguments=
   ```

5. Finally the shell parses file1, file2, file3, and “and more” as the arguments to echo.

   ```
   Input buffer=echo $(ls) file1 file2 file3 ”and more”
   Command Word=echo
      Arguments=file1, file2, file3, “and more”
   ```

You will implement this feature by

1. Scanning the command between backticks in `shell.l`
2. Calling your own shell as a child process and passing it the command as input. You will need two pipes to communicate with the child process; one to pass the command to the child, and the other to read the output from the child.

3. Reading the output from the child process and putting the characters of the output back into the scanner's buffer using the function `yy_unput(int c)` in reverse order. See the FAQ for more details.

Hint: It is common for students to redirect the current shell's stdin and stdout file descriptors to communicate with the subshell process, however this is not necessary. The current shell can communicate with the subshell by writing to the pipes directly.

IMPORTANT: Do not use the `popen()` call or a temporary file for the interprocess communication. You must use the method discussed above.

Submission

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 `git tag -f part2`*
6. *Run `git push -f origin part2`*
7. *Run `git show part2`*
8. *The `show` command should show the diff from the most recent commit*

Part 3: Expansions, Wildcards, and Line Editing

The final part of the lab involves adding a few major usability features to your shell. You will allow for your parser to expand a few types of input, handle wildcards, and implement a line editor that allows you to do things like fixing typos and traversing a history of previously submitted commands.

3.1: Tilde expansion

When the character "~" appears itself or before "/" it will be expanded to the home directory of the current user. 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 home directory
ls ~george        -- List george's home directory
ls ~george/dir    -- List subdirectory "dir" in george's directory
```

3.2: Wildcarding

In most shells, including `bash` and `csh`, you can use `*` and `?` as wildcard characters in file and directory names. 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. Try wildcarding in `csh` to see the results. You will implement wildcarding as follows:

1. First, handle wildcarding only within 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 wildcards to regular expressions.
2. Once your wildcarding implementation works for the current directory, make it work for any absolute path.

IMPORTANT: Do not use the `glob()` call. You must use the functions discussed above.

Reminder: you do not need to handle wildcard expansion between quotation marks!

3.3: Supporting if/while/for

The file shell.y already includes rules for matching if/while/for expressions. You will complete the implementation of these script constructions in your shell.

3.3.1 Implementing *if* statement

When the shell receives an input such as:

```
myshell> if [ -f Shell.o ]; then echo File Exists; fi
File Exists
```

Also, it can be used in the following way

```
myshell>if [ -f Shell.o ]; then
echo File Exists
fi
File Exists
```

Or in a shell script

```
vim testif.sh
#!./shell
if [ -f Shell.o ]; then
echo File Exists
fi
:x
chmod +x testif.sh
./testif.sh
File Exists
```

The arguments inside the brackets [-f Shell.o] will be executed by your shell in a child process using the UNIX command "test -f Shell.o" and if the exit value is 0 (success) then the list of commands inside the if statement (echo File Exists) will be executed.

You can run the UNIX "test" command as follows.

```
bash> test -f Shell.o
echo $?
0
```

Type "man test" to see other arguments for the test command.

3.3.2 Implementing ***while*** statement

When the shell receives an input such as:

```
myshell> setenv count 5; while[ $count -ne 0 ]; do echo $count; setenv
count `expr count - 1`; done
5
4
3
2
1
```

Also, it can be used in the following way

```
myshell>setenv count 5; while[ $count -ne 0 ]; do
echo $count; setenv count `expr count - 1`;
done
5
4
3
2
1
```

Or in a shell script

```
vim testwhile.sh
#!./shell
setenv count 5;
while[ $count -ne 0 ]; do
  echo $count;
  setenv count `expr count - 1`;
done
:x
chmod +x testwhile.sh
./testwhile.sh
5
4
3
2
1
```

The arguments inside the brackets [-f Shell.o] will be executed by your shell in a child process using the UNIX command "test -f Shell.o" and if the exit value is 0 (success) then the list of commands inside the while statement will be executed. After executing the list of commands, it will reevaluate the expression in brackets.

3.3.3 Implementing ***for*** statement

When the shell receives an input such as:

```
myshell> for t in a b c d; do echo $t $t.org; done
a a.org
b b.org
c c.org
d d.org
```

Also, it can be used in the following way

```
myshell>for t in a b c d; do
echo $t $t.org;
done
a a.org
b b.org
c c.org
d d.org
```

Or in a shell script

```
vim testfor.sh
#!./shell
for t in a b c d; do
   echo $t $t.org;
done
chmod +x testfor.sh
./testwhile.sh
5
4
3
2
1
```

The arguments inside the brackets [-f Shell.o] will be executed by your shell in a child process using the UNIX command "test -f Shell.o" and if the exit value is 0 (success) then the list of commands

inside the while statement will be executed. After executing the list of commands, it will reevaluate the expression in brackets.

3.3.4 Implementing Argument Environment Variables

To be able top interact with the shell script arguments, you will Add the following environment variables:

`${#}`	Number of arguments
`${0}`	The shell script name
`${1}, ${2},... ${n}`	Argument 1 to n of the script
`${*}`	Expands to all the arguments passed to the script.

3.4: Edit mode (Extra only after finishing required parts)

`tty-raw-mode.c` and `read-line.c` contains the sample code that you will need to change your terminal's input from canonical to raw mode. In raw mode you will have more control over the terminal, passing the characters to the shell as they are typed.

There are two example programs to look at: `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 `cread-line.ctrl-?` to see the options of this program. The up-arrow causes the program to print the previous command in its history.

The file `tty-raw-mode.c` contains sample code which switches the terminal from canonical to raw mode. The file `read-line.c` contains sample code which 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 ///////////

extern “C” 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. To do this just define`EDIT_MODE_ON` variable in the `Makefile` to be something for example `"yes"`.

```
EDIT_MODE_ON=yes
```

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 key (ctrl-H): Removes the character at the position before the cursor. The characters in the right side are shifted to the left.
- Home key (ctrl-A): The cursor moves to the beginning of the line
- End key (ctrl-E): The cursor moves to the end of the line

IMPORTANT: Do not use the readline library. You must implement your own line editor.

3.5: History (Extra only after finishing required parts)

In addition to the line editor above, also implement a history list. Currently the provided history is static. You need to update the history by creating your own history table. Every time the user runs a new command, a row will be added to the 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.

3.6: Path completion (Extra only after finishing required parts)

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 csh and bash do.

```
bash$ ls
cart.txt    card.txt
bash$ c<tab>
```

When tab is pressed, the line above becomes:

```
bash$ car
```

With the line indicator after c.

3.7: Variable prompt (Extra only after finishing required parts)

The shell has a default prompt indicator: `myprompt>`. If there is an environment variable called `PROMPT`, your shell should print the value of that variable as the prompt 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.

```
myshell> setenv PROMPT --cs252--
--cs252-- gcc
gcc: fatal error: no input files
compilation terminated
--cs252-- setenv ON_ERROR oops
--cs252-- gcc
gcc: fatal error: no input files
compilation terminated
oops
--cs252--
```

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. Manual testing will count for 10% of the total grade of the shell.

Submission

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 you have 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 `git tag -f part3`*
6. *Run `git push -f origin part3`*
7. *Run `git show part3`*
8. *The `show` command should show the diff from the most recent commit*

Grading

10% Milestone 1 (./testall p1 in lab)
10% Milestone 2 (./testall p2 in lab)
70% Final Testall
10% Manual Grading of readline and Ctrl+C
-5% For Memory Leaks
-5% For File Descriptor Leaks

Resources

Lex and Yacc Primer

Lab3 part1 slides (parsing)
Lab3 part1 slides (executing)
Lab3 part2 slides
Lab3 part3 slides