

# 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.

---

#### [Introduction](#)

##### [Getting Started](#)

#### [Part 1: Parsing and Executing Commands](#)

##### [Part 1A: Lex and Yacc - Accepting more complex commands](#)

##### [Part 1B: Executing commands](#)

###### [1B.1: Simple command process creation and execution](#)

###### [1B.2: File redirection](#)

###### [1B.3: Pipes](#)

###### [1B.4: isatty\(\)](#)

##### [Testing](#)

##### [Submission](#)

#### [Part 2: Signal Handling, More Parsing, and Subshells](#)

##### [2.1: Ctrl-C](#)

##### [2.2: Zombie Elimination](#)

##### [2.3: Exit](#)

##### [2.4: Quotes](#)

##### [2.5: Escaping](#)

##### [2.6: Builtin Functions](#)

##### [2.7: Creating a Default Source File: ".shellrc"](#)

##### [2.8: Subshells](#)

##### [2.9: Process Substitution](#)

##### [Submission](#)

#### [Part 3: Expansions, Wildcards, and Line Editing](#)

[3.1: Environment variable expansion](#)

[3.2: Tilde expansion](#)

[3.3: Wildcarding](#)

[3.4: Edit mode](#)

[3.5: History](#)

[3.6: Path completion](#)

[3.7: Variable prompt](#)

[Submission](#)

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

```
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.

---

# 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++*.

代码文件

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 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**.

pipe

文件

*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.*

C++

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

Lex, Yacc, Flex 文档

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]
```

代码文件

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

更复杂语法

```
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 dfdfdf
ls aaaa | grep cccc | grep jjjj ssss dfdfdf >& out < in
httpd &
ls aaaa | grep cccc | grep jjjj ssss dfdfdf >>& out < in
```

## 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. 运行示例

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 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.

代码

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.

代码

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 `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 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 中断

In `csch`, `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 `csch` does with respect to Ctrl-C. See [ctrl-c.cc](http://ctrl-c.cc) for an example of detecting and ignoring a `SIGINT` signal. Also see the `man` page for `sigaction()`. 示例

### 2.2: Zombie Elimination 处理僵尸

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

等待 exit 被接收

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."

### 3 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$
```

### 4 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.

### 5 2.5: Escaping



5  
✓

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 &
```

6

## 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 2.2, these commands are detected by the shell during parsing to carry out certain special functions. Implement the following builtin commands:

✓	<code>printenv</code>	Prints the <b>environment variables</b> of the shell. The environment variables of a process are stored in the <b>variable</b> <code>char **environ</code> ; a <b>null-terminated array of strings</b> . Refer to the <code>man</code> page for <code>environ</code> .
✓	<code>setenv A B</code>	<b>Sets the environment variable</b> A to value B. See <a href="#">article</a> .
✓	<code>unsetenv A</code>	<b>Un-sets environment variable</b> A
	<code>source A</code>	<b>Runs file A line-by-line</b> , as though it were being typed into the shell by a user. See <a href="#">Multiple Input Buffers</a> or look at <a href="#">Flex</a> manual
✓	<code>cd A</code>	<b>Changes the current directory to A</b> . If no directory is specified, default to the <b>home directory</b> . See the <code>man</code> page for <code>chdir()</code> .

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

7

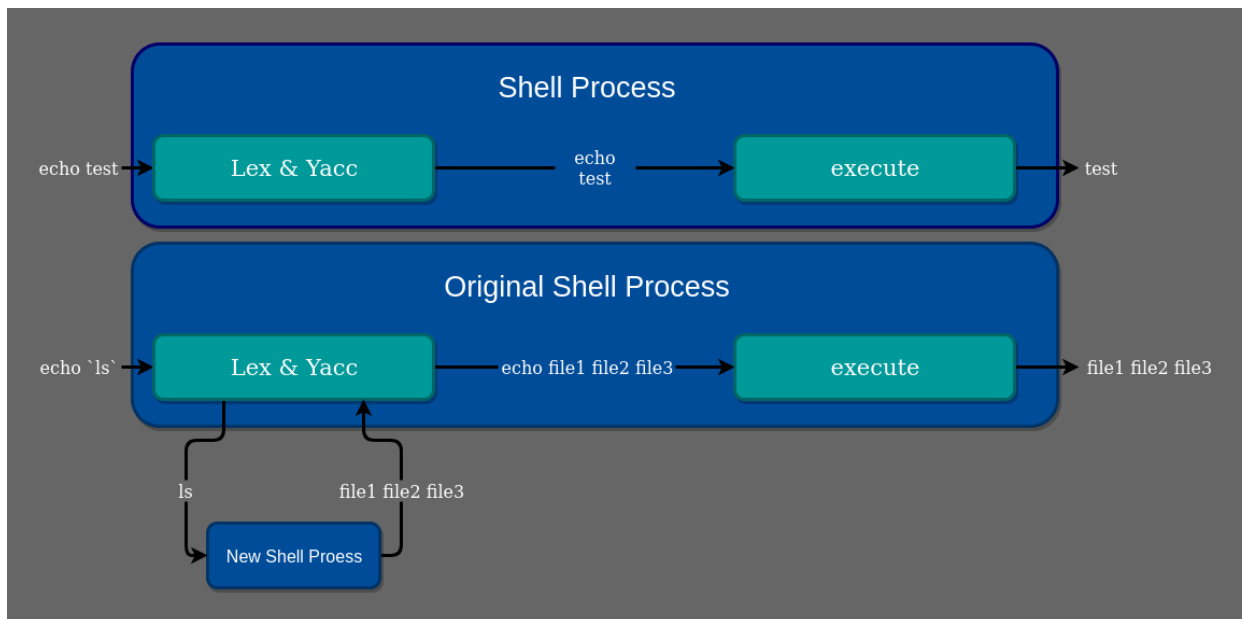
## 2.7: Creating a Default Source File: ".shellrc"

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

## 2.8: 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)
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.1`
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.

## 2.9: Process Substitution (Extra credit)

Subshells allow you to replace a command with its output as command line arguments and pipes allow you to replace stdin with the output of another process. These are both very powerful tools, but they have some limitations.

```
myshell> vim $(echo hello)
myshell> echo hello | vim

myshell> vim <(echo hello)

myshell> mkfifo fifo
myshell> echo hello > fifo &
myshell> vim fifo
myshell> rm fifo
```

In the example above we want to edit the output of a command in vim. The problem is that vim does not read its input from stdin and it expects filenames for its arguments. Therefore we cannot use pipes or subshells. To do this we can use process substitution which allows us to pass the output of a command to another command as if the output was the contents of some file. To achieve this the shell creates a named pipe to communicate between the two processes (echo and vim in the example above). A named pipe, unlike the pipes we have been using so far, are created at some specified location in the file system. This allows us to pass a command to a named pipe as a file path and have another process write its output into the named pipe.

To implement this feature you should use `mkdtemp()` to generate a temporary directory for the named pipe. Within the temp directory create a named pipe with `mkfifo()`. In your shell connect the command where you are doing the process substitutions output to the named pipe and pass the name of the pipe to the other command. Make sure to clean up any resources (fifo and directory) after the command has been executed. Look into the `unlink()` and `rmdir()` functions.

## 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
- 

## ctrl-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: 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:

<code>\${\$}</code>	The <b>PID</b> of the <b>shell process</b>
<code>\${?}</code>	The <b>return code</b> of the <b>last executed simple command</b> (ignoring commands sent to the <b>background</b> ). ?
<code>\${!}</code>	<b>PID</b> of the last process <b>run</b> in the background
<code>\${_}</code>	The <b>last argument</b> in the fully expanded previous command Note: this <b>excludes redirects</b>
<code>\${SHELL}</code>	The <b>path</b> of your <b>shell executable</b> . Hint: <u>realpath()</u> can expand a relative path to an absolute path. You can obtain the <b>relative path</b> to the shell in <code>argv[0]</code>



## 家目录 3.2: 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
```

## \* 3.3: Wildcarding 自动匹配

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). **所有文件**
- Use **opendir** and **readdir** to get **all the entries of the current directory** (check the man pages). **例子 匹配**
- Use the functions **regcomp** and **regexexec** 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**. **WC→RE**

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. **glob()X**

Reminder: you do **not** need to handle wildcard expansion **between quotation marks!** **バックslash 不考慮**

### 3.4: Edit mode

**文本编辑** `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. **示例2**

The other program, **read-line-example**, is a **simple line editor**. Run this program and type `read-line.ctrl-?` to see the options of this program. The up-arrow causes the program to print the previous command in its history. **示例3**

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.

修改

To connect the line editor to your shell, add the following code to `shell.1` 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 //////////  
  
%}  
  
%%
```



修改

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
```

修改

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 <sup>X</sup> use the `readline` library. You must implement your own line editor.

### 命令历史 3.5: History

✓ 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

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 `csch` 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

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](#)