Assignment 6:

Programming a UNIX Shell
Recitation for ECES 338
March 22, 1999

Overview of this assignment

The purpose of this assignment is to write a simplistic (yet decently-featured) UNIX shell. The assignment itself is very detailed, and maps out the functionality that you need to provide. Furthermore, the assignment contains helpful advice on how to achieve this functionality.

In essence, a shell is a program that takes a specified input (a *command*), and then uses fork() and exec() in order to execute this command. Upon completion, it will prompt the user for another command. Additionally, the shell allows for file redirection, and for pipes to be setup between commands. You will have to implement these sorts of functions into your shell.

- **About "parser.c"**
  This C file contains all of the functionality necessary in order to parse a command given to your shell. It is available from the following URLs:

  - `ftp://erciyes.ces.cwru.edu/338/parser.c`
  - `http://home.cwru.edu/~ajr9/eces338/source/parser.c`

  The use of this file is fairly well documented in the assignment – simply `#include` it in your source code, and make a call to the `parsecmd()` function, as specified. You should not need to modify any of the code in this file, but the assignment does not explicitly state that you are to leave it alone. Therefore, if you **do** modify this file, there had better be a **flaming** note of this fact accompanying your assignment, so that the TA knows what is going on…

- **More about piping and redirection**
  For redirection, all you have to do is `fopen()` the given filename, and then reconnect it to `stdin` (or `stdout`) as necessary. This reconnection process is done via the `dup2()` system call, which is documented on pages 126 – 129 of the *Monkey Book*. In order to handle pipes, you may want to use the `popen()` system call. This call is documented (with a nice example) on pages 131 – 132 in the *Monkey Book*.

The `execvp()` system call

Pages 62 and 63 in the *Monkey Book* document this system call fairly decently. There is also an example presented on the bottom of the first page of the assignment, which explicitly lays out how to use this function. Basically, the first argument is the

name of the file to execute, and the second argument is a list of parameters to send to this executable. The parser that is provided handles the details of constructing this second parameter, so that all you have to do is pass the proper data to `execvp()`, and watch it go to work.

The `struct cmd` structure

The parser code includes a rather beefy structure, upon which it places the guts of the string that it dissects. I am going to give a brief overview of this structure, since it is vital to the understanding of this assignment.

```
struct cmd {
    char *cmdname;    /* ptr to '\0' terminated string containing command   */
                      /* NULL if there is no command                        */
    char *argv[10];   /* ptr's to up to 10 arguments for this command       */
                      /* if there are i args, argv[i] is NULL               */
    char *infile;     /* NULL if no redirected input, else ptr to filename  */
    char *outfile;    /* NULL if no redirected output, else ptr to filename */
    int  pipe;        /* ==YES means there IS a pipe after this command.    */
    }  commands[MAXCMDS];  /* If there are i commands on the input command  */
                      /* line then command[j].cmdname==NULL for all j>=i    */
```

- The `cmdname` variable is simply the name of the file to be executed.
- For the `argv` member, it is important to note that the last element in the list will be a `NULL` character. This is important because `execvp()` will not work without it.
- If `infile` is not `NULL`, then `fopen()` it in read-only mode, and make it take over the `stdin` resource of child process.
- If `outfile` is not NULL, the `fopen()` it in write-only mode, and make it take over the `stdout` resource of the child process.
- The pipe variable will be set to either `YES` or `NO`, depending on if a pipe exists in the given command. Based upon this value, your program can decide if it needs to setup a pipe between the commands returned by `parsecmd()`.

The `sort` command

Don't worry to much about the sort command. It has a lengthy man page, and if you are interested, you can feel free to exercise it for all that it is worth. However, I recommend that you just stick to the example presented in the assignment.

Testing your shell

Don't limit your testing to the commands listed in the assignment. Those commands are only there as a guide. Your shell should be able handle decently complex input strings, so long as it doesn't violate the capabilities of the `parsecmd()` function.