# A Simple Shell[†]

Prof. Darrell Long

Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

# Due: *3 February at 1700h*

## Purpose

The primary goal of this assignment is to warm up your programming skills that may have atrophied over the inter-quarter break, and to gain some familiarity with the system call interface. A secondary goal is to use some of the programming tools provided in the FreeBSD environment. In this assignment you are to implement a FreeBSD shell program. A shell is simply a program that conveniently allows you to run other programs.

## Basics

You are provided with the files `lex.l` and `myshell.c` that contain some code that calls `getline()`, a function provided by `lex.l` to read and parse a line of input. The `getline()` function returns an array of pointers to character strings. Each string is either a word containing the letters, numbers, period (`.`), and forward slash (`/`), or a character string containing one of the special characters: `(`, `)`, `<`, `>`, `|`, `&`, or `;` (which have syntactical meaning to the shell).

To compile `lex.l`, you have to use the `flex` command. This will produce a file called `lex.yy.c`. You must them compile and link `lex.yy.c` and `myshell.c` in order to get a running program. In the link step you also have to use `-lfl` to get everything to work properly. Use `cc` for the compilation and linking.

## Details

Your shell must support the following:

1. The internal shell command `exit` which terminates the shell.
    a. *Concepts*: shell commands, exiting the shell
    b. *System calls*: `exit()`
2. A command with no arguments.

---

[†] Although the idea of writing a shell as the first assignment in the operating systems course is an ancient one, dating at least to the venerable Prof. Jehan-François Pâris, I have borrowed liberally from the assignment designed by Prof. Scott Brandt. Even so, it differs in key aspects which will be observed by the discerning student.

    a. *Example*: `ls`
    b. *Details*: Your shell must block until the command completes and, if the return code is abnormal, print out a message detailing the error. This holds for *all* command strings in this assignment.
    c. *Concepts*: Forking a child process, waiting for it to complete, synchronous execution.
    d. *System calls*: `fork()`, `execvp()`, `exit()`, `wait()`
3. A command with arguments.
    a. *Example*: `ls -l`
    b. *Details*: Argument *zero* is the name of the command other arguments follow in sequence.
    c. *Concepts*: Command-line parameters.
4. A command, with or without arguments, whose output is redirected to a file.
    a. *Example*: `ls -l > file`
    b. *Details*: This takes the output of the command and put it in the named file.
    c. *Concepts*: File operations, output redirection.
    d. *System calls*: `close()` and some variety of `dup()`
5. A command, with or without arguments, whose input is redirected from a file.
    a. *Example*: `sort < scores`
    b. *Details*: This takes the named file as input to the command.
    c. *Concepts*: Input redirection, file operations.
    d. *System calls*: `close()` and some variety of `dup()`
6. A command, with or without arguments, whose output is piped to the input of another command.
    a. *Example*: `ls -l | more`
    b. Details: This takes the output of the first command and makes it the input to the second command.
    c. Concepts: Pipes, synchronous operation
    d. System calls: `pipe()`, `close()` and some variety of `dup()`
7. A command played into the background with `&`
    a. *Example*: `awk "{print $1}" < pwds | sort -u > unique &`
    b. Typing `jobs` will provide a list of jobs.
    c. Each job will list the process IDs (PIDs) of the processes in that job.
    d. Instead of a single wait, the command shell must check a list of jobs to see if any have completed.

You must check and correctly handle all return values. *This means that you need to read the manual pages for each function and system call to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.*

## Deliverables

Push your project directory to `git`, including your `README`, design document (`DESIGN`) and `Makefile`. *Do not* push any binary files. Include a README file to explain anything unusual to

the teaching assistant. Your code and other associated files must be in a single directory so they will build properly in the submit directory.

Do not submit object files, assembler files, or executables. Every file in the submit directory that could be generated automatically by the compiler, assembler, loader, or `flex` will result in a 5-point deduction from your programming assignment grade.

Your design document should be called `DESIGN.txt` (if in plain text), or `DESIGN.pdf` (if in Adobe PDF) and should reside in the project directory with the rest of your code. Formats other than plain text or PDF are not acceptable; please convert other formats (Word, LaTeX, HTML, …) to PDF. Your design should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

Note on the attachment: `lex` cares about spaces, just like `make` does. As usual, *read the documentation*.

## Appendix I: lex.l

```
%{
# DEFINE    YY_DECL     char **yylex()
int   _numargs = 10;
char  *_args[10];
int   _argcount = 0;
%}

WORD        [a-zA-Z0-9\/\.-]+
SPECIAL     [()><|&;*]

%%
      _argcount = 0; _args[0] = NULL;

{WORD}|{SPECIAL} {
        if(_argcount < _numargs) {
          _args[_argcount++] = (char *)strdup(yytext);
          _args[_argcount] = NULL;
        }
      }

\n    return _args;

[ \t]+

.

%%

char **getline() { return yylex(); }
```

## Appendix II: myshell.c

```
#include    <stdio.h>
#include    <unistd.h>
#include    <sys/types.h>
#include    <errno.h>

extern char **getline();

int main() {
  int i;
  char **args;

  while(1) {
    args = getline();
    for(i = 0; args[i] != NULL; i++) {
      printf("Argument %d: %s\n", i, args[i]);
    }
  }
}
```