Assignment 5

cpe 357 Fall 2020

```
As soon as we started programming, we found to our surprise that it wasn't
as easy to get programs right as we had thought.  Debugging had to be
discovered.  I can remember the exact instant when I realized that a large
part of my life from then on was going to be spent in finding mistakes in
my own programs.
                -- Maurice Wilkes, designer of EDSAC, on programming, 1949
```

— /usr/games/fortune

Due by 11:59:59pm, Friday, November 16th.
For this assignment you may work with a partner[1]. Be sure both names appear in the README.

Program: `parseline`

Shell command-line parsing:

This assignment is to do the command-line parsing necessary for the Minimally Useful SHell (`mush`) that will be the subject of Asgn 6.

The Minimally Useful SHell (`mush`) has nowhere near the functionality of a full-blown shell like `/bin/sh` or `/bin/csh`, but it does support both file redirection and pipes. `parseline` is a subset of the shell that prompts for and reads a single `mush` command-line and parses it into a list of commands showing the inputs, outputs, and arguments of each.

Details

The grammar of `mush` is fairly simple:

- A command (pipeline stage) consists of a command name followed by its arguments, separated by whitespace.
- A command's standard input and standard out can be redirected from or into files via the use of `<` (standard in) and `>` (standard out). The filename for the redirection is the single word following the redirection symbol. A missing name names constitutes an error.

  The redirection symbol and filename are not considered part of the command name or argument list and are not included in the count of arguments.
- A pipe (|) connects the standard output of one command to the standard input of the following one. For example, "`ls | sort`" makes the output of `ls` the input of `sort`. A series of commands separated by pipes is a pipeline.
- You can assume that '`<`', '`>`', and '`|`' will appear as words by themselves with space around them. That is, you don't have to deal with "`ls b<a|more`". In addition, the characters '`<`', '`>`', and '`|`' are not valid filenames, so "`a.out > < a`" is an error, not the creation of a file called "`<`."

  Note, however, that redirections will not necessarily appear at the end of the command line. That is, "`ls > out a b`" would be a legitimate command to list the files "`a`," and "`b`" into the file "`out`".

[1] This will have to be a partner for both this assignment and Asgn 6.

In order to make the process easier, you may apply certain limits to the command line structure. These limits must be documented in your README file, and command-lines that are rejected for exceeding limits must be reported as errors.

Command line length:	at least 512 bytes
Commands in a pipeline:	at least 10
Arguments to a command:	at least 10

The fact that these maxima[2] exist will allow you to avoid the use of dynamic data structures. My initial version of this program did not have a single call to `malloc()` in it. (The current version does.)

Error Handling

`parseline` must identify and report malformed commands. This includes:

- malformed redirects. For example, "`a.out <` " doesn't specify the name of the file for redirect while "`a.out < a < b`" has two input redirects.
- ambiguous inputs or outputs. For example, in the pipeline "`ls | sort < foo`", the input to `sort` is specified to be two different things.
- command-lines that exceed any limits imposed (above).

Output

The purpose of parsing a command line is to identify the various components of each command so that each can be launched appropriately. On a Unix system, one needs to know where the input will come from, where the output will go, the number of arguments on the command line (not including any redirection commands) and the values of those arguments.

In order for this program to be efficiently graded, it is important to adhere to the output format specified below.

Error Cases When there is an error in one of the commands in the pipeline, `parseline` prints an error message an exits with nonzero exit status. If there are multiple errors on a line, it prints the the first one it encounters. Possible errors are shown in Table 1.

Valid Cases For syntactically correct pipelines, `parseline` should print out a description of each stage of the pipeline in the form given below. For the sake of `parseline`, the first stage of a pipeline will be stage 0. The required form of the output:

1. a header that identifies the stage number and shows the portion of the command line that corresponds to that stage, in quotes. This header should follow a blank line and have the following form:

   ```
   --------
   Stage n:  "<command line>"
   --------
   ```

   Example:

   ```
   --------
   Stage 0: "ls a  b   c "
   --------
   ```

[2]The phrasing here is a little awkward. What it means is that you may apply a maximum limit for each of these properties, but the value of your maximum must be at least as big as that given in the table above. That is, you can't define the maximum command line length to be 0 and be done.

Cause	Message
command line length limit exceeded	`command too long`
pipeline has too many elements	`pipeline too deep`
an individual command has exceeded the limit on arguments	*cmd:* `too many arguments`
a pipeline has an empty stage, e.g., "`ls | | more`"	`invalid null command`
a command either has more then one input redirection character ('<'), or the input filename is missing	*cmd:* `bad input redirection`
a command either has more then one output redirection character ('>'), or the input filename is missing	*cmd:* `bad output redirection`
a stage has both an input redirect and a pipe in	*cmd:* `ambiguous input`
a stage has both an output redirect and a pipe out	*cmd:* `ambiguous output`

Table 1: Possible errors that `mush` will encounter

2. a specification of the input for the stage which will be one of:

   a filename, if redirected
   "`original standard input`"
   "`pipe from stage` n"

   Example:

   ```
   input: original stdin
   ```

3. a specification of the output for the stage which will be one of:

   a filename, if redirected
   "`original standard output`"
   "`pipe to stage` n"

   Example:

   ```
   output: pipe to stage 1
   ```

4. the argument count (`argc`) for the stage.

   Example:

   ```
   argc: 4
   ```

5. the arguments strings for the stage with extra whitespace trimmed off, in quotes, comma-separated.

   Example:

   ```
   argv: "ls","a","b","c"
   ```

Tricks and Tools

Remember, parsing is always harder than it looks. Be sure to give some serious thought to your approaches and data structures before diving in. There are many library routines and system calls that may help with implementing `parseline`. Some of them are listed in Figure 1.

`sscanf()`
`strchr()`
`index()`
`strtok()`
`strpbrk()`
etc.
	The string functions, defined in `string.h` and `strings.h`, are helpful for parsing strings (man `string(3)` for more)

`isspace()` etc.	One of many functions defined in `ctype.h` for text processing. Very useful.

Figure 1: Some potentially useful system calls and library functions

A few thoughts:

- You can find out the number of stages in the pipeline by counting number of times the pipe character (`|`) appears and adding 1.
- You only need to report the first error you find, so you can abort processing once you find one.
- Be far-sighted about this. Read the specification for the full version of `mush` before starting so you will know where you want to end up. You want to make design decisions you can live with next week.
- Remember that although parsing looks easy it it harder than it looks. There are a lot of edge cases.

Coding Standards and Make

See the pages on coding standards and make on the cpe 357 class web page.

What to turn in

Submit via `handin` to the `asgn5` directory of the `pn-cs357` account:

- your well-documented source files.
- A makefile (called `Makefile`) that will build your program with the command "`make parseline`".
- A README file that contains:
  - Your name(s). In addition to your names, please include your Cal Poly login names with it, in parentheses. E.g. `(pnico)`
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

The README file should be **plain text,** and should be named "README", all capitals with no extension.

Sample Runs

Below are some sample runs of `parseline`. I will also place an executable version on the CSL in `~pn-cs357/demos` so you can run it yourself.

```
% parseline
line: ls

--------
Stage 0: "ls"
--------
     input: original stdin
    output: original stdout
      argc: 1
      argv: "ls"
% parseline
line: ls < one > two three four

--------
Stage 0: "ls < one > two three four"
--------
     input: one
    output: two
      argc: 3
      argv: "ls","three","four"
% parseline
line: ls < one | more | sort

--------
Stage 0: "ls < one "
--------
     input: one
    output: pipe to stage 1
      argc: 1
      argv: "ls"

--------
Stage 1: " more "
--------
     input: pipe from stage 0
    output: pipe to stage 2
      argc: 1
      argv: "more"

--------
Stage 2: " sort"
--------
```

```
     input: pipe from stage 1
    output: original stdout
      argc: 1
      argv: "sort"
% parseline
line: ls | | more
invalid null command
% parseline
line: ls < a < b | more
ls: bad input redirection
% parseline
line: ls < a | more < file
more: ambiguous input
% parseline
line: ls < a < b > c > d
ls: bad input redirection
%
```