EC440: Project 0 – Simple Shell Parser

Project Goals:

A number of students have asked us to suggest work that they can do before the class starts to help them prepare. As a result, we have teased out a portion of the first project, and are making it available as an optional Project 0 that you can start on before the class starts. This project should help you:

- Develop a parser that can later be used to implement a simple shell.
- Review C programming concepts:
    - Functions
    - String processing
    - Structs
    - Pointers
- Gain experience with Linux and tools you will use in the course.
- Update your experience with github.
- Gain experience developing simple unit tests.

We will provide you github repo with sample headers, and will over the next week make available a linux container environment where all development should be done. You should commit your solution frequently to your repository. You will be expected to be able to explain your github history, and how you came to your existing solution.

Collaboration Policy:

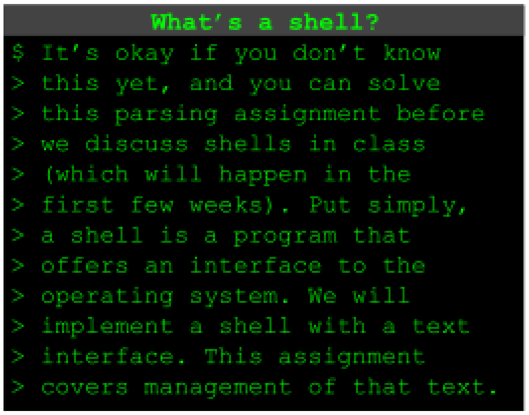
You are encouraged to discuss this project with your classmates and instructors. You **must** develop your own solution. While you **must not** share your parser code with other students, you **are encouraged** to share testcase code that you developed to test your solution for this assignment.

Deadline:

Project 0 is an **optional** assignment, but solving it will reduce the difficulty of solving project 1.

Project Description:

The goal of this project is to implement a command parser for a simple shell. In the *next assignment*, you will implement a simple shell. The code you create for this assignment will continue to be useful since you can use it as part of your shell implementation.



```
What’s a shell?
$ It’s okay if you don’t know
> this yet, and you can solve
> this parsing assignment before
> we discuss shells in class
> (which will happen in the
> first few weeks). Put simply,
> a shell is a program that
> offers an interface to the
> operating system. We will
> implement a shell with a text
> interface. This assignment
> covers management of that text.
```

This assignment is designed to give you a chance to warm up your C programming skills without requiring an understanding of Operating Systems topics that we have yet to discuss in class.

Parsing Shell Commands

You will write a program that receives a line of text as an input, and produces the same data in a structured form as output.

A typical approach to solving a parsing problem is to separate it into two parts: lexing and parsing. When lexing, you group parts of your string into a series of *tokens*. When parsing, you search for patterns of tokens that match an expected structure of *symbols*. You aren't required to strictly follow that separation of concerns in your implementation, but it may help provide a structured approach to solving this challenge.

For this assignment, the amount of separating whitespace between tokens has no meaning. That is, two tokens separated by a single space shall be treated the same as two words separated by 10 spaces.

Tokens:

- Ampersand (&) is used to indicate a background pipeline
- Pipe (|) is used to join commands in a pipeline
- The less-than symbol (<) is used to assign a redirect-in to a command.
- The greater-than symbol (>) is used to assign a redirect-out to a command.
- A *word* is used for arguments and redirect paths. It is a sequence of characters that is none of the above tokens, and is not whitespace (i.e., not space, tab '\t', or newline '\n').

Symbols:

- The provided line of text may contain zero or 1 *pipeline*. A *pipeline* is a sequence of one or more *command* symbols, joined by a pipe token.
- A *command* is a series of one or more *argument* symbols.
- An *argument* is a series of one or more *word* tokens.
- Each command may optionally have a *redirect-in* and/or a *redirect-out.* If both are present, they may be specified in either order.
  - A redirect-in consists of a less-than token followed by a word token. The word is the redirect source.
  - A redirect-out consists of a greater-than token followed by a word token. The word is the redirect destination.
  - If you are already familiar with how redirection and pipes work, you may note that there some combinations can't really work. You don't need to worry about those for this parsing assignment. You are free to ignore such cases at this time.
- A pipeline may optionally end with an *ampersand* token, indicating that the pipeline should run in the background.

Examples of Valid Inputs

`“/bin/cat my_file\n”, “a& \n”, “pipefrom| pipeto\n”, “cat< a_file\n”, “ls > txt\n”` are all valid inputs.

A detailed parsing example of a single input is included in the data structures section of this document.

Functions You Will Implement

You are expected to provide an implementation for two functions, which are declared in a header file that we provide:

```
struct pipeline *pipeline_build(const char *command_line);
```

This function creates a pipeline structure that represents the given command line. You may or may not choose to use malloc() while implementing this function. If you choose to use malloc(), any allocated memory must be freed by `pipeline_free()`.

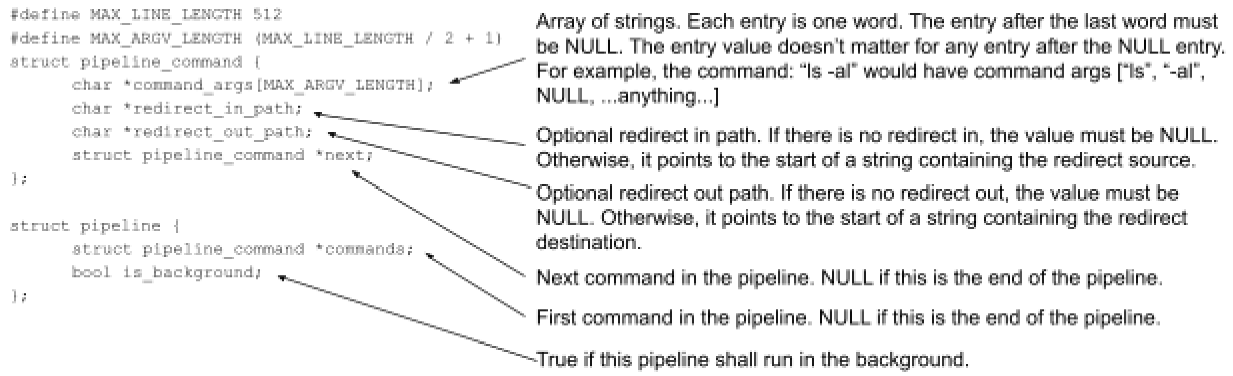
```
void pipeline_free(struct pipeline *pipeline);
```

Frees a given `pipeline` that was created with pipeline_build(). If your pipeline_build() function performs any dynamic memory allocation, this is where the corresponding deallocation must occur for `pipeline` and for any nested data structures it contains.

You will **not implement** a main function in your submitted code (that will be part of the next assignment). However, you are encouraged to develop your own unit tests which may have their own main functions.

Data Structures We Provide

Don’t type these out. We will provide a file that contains these data structures and the function declarations.



```
#define MAX_LINE_LENGTH 512
#define MAX_ARGV_LENGTH (MAX_LINE_LENGTH / 2 + 1)
struct pipeline_command {
        char *command_args[MAX_ARGV_LENGTH];
        char *redirect_in_path;
        char *redirect_out_path;
        struct pipeline_command *next;
};

struct pipeline {
        struct pipeline_command *commands;
        bool is_background;
};
```

Array of strings. Each entry is one word. The entry after the last word must be NULL. The entry value doesn't matter for any entry after the NULL entry. For example, the command: “ls -al” would have command args [“ls”, “-al”, NULL, ...anything...]

Optional redirect in path. If there is no redirect in, the value must be NULL. Otherwise, it points to the start of a string containing the redirect source.

Optional redirect out path. If there is no redirect out, the value must be NULL. Otherwise, it points to the start of a string containing the redirect destination.

Next command in the pipeline. NULL if this is the end of the pipeline.

First command in the pipeline. NULL if this is the end of the pipeline.

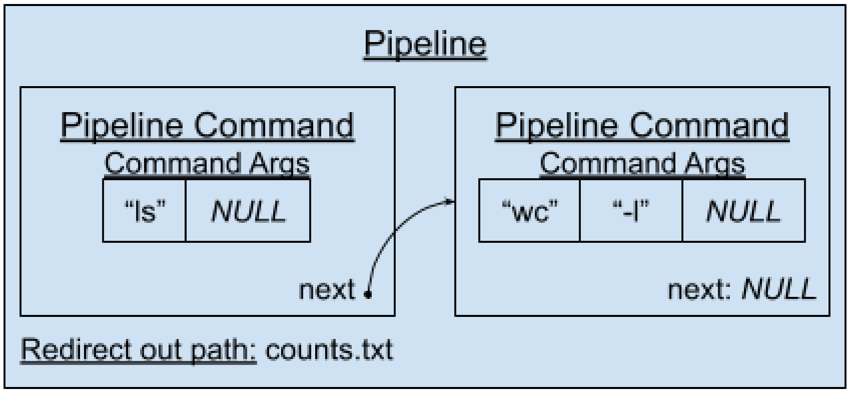
True if this pipeline shall run in the background.

Detailed breakdown of a single input:

Given `pipeline_build("ls|wc -l >counts.txt\n")`, expect the following parser output:

- This is a valid input, so a non-NULL pipeline pointer is returned.
- Pipeline commands:
  - First command:
    - command_args are ["ls", NULL]
    - redirect_in_path is NULL
    - redirect_out_path is NULL
  - Next command
    - command_args are ["wc", "-l", NULL]
    - redirect_in_path is NULL
    - redirect_out_path is "counts.txt"
- Pipeline is_background is false (because no ampersand was present)

The same output, in visual form is included below:



Given Code:

We will make this code available in your git repo as soon as we get that set up.

Some Hints:

- Solve the lexing/tokenizing problem first, then worry about parsing after that. Write a test that sends the provided valid inputs to your lexing function, and verify that it returns the expected series of tokens.
- Develop your own test cases. While the autograder has some descriptive explanations of what it is trying to do, it will not show you how or why your submitted code is failing a test. A good test suite for this assignment has many test cases. A good test case has some kind of expectations about how a program will react to different inputs and system state changes, and will fail when those expectations are not met.
  - The detailed breakdown in the previous section provides the basis of one test case. Write that in C code so you don't need to manually very every time you change your parser.

- Let your software development tools help you:
  - Don’t ignore compiler warnings. If the compiler tells you that something looks suspicious, it is because a compiler engineer determined that the suspicious code pattern is likely to lead to a bug or some other undesirable behavior.
  - Instead of repeatedly recompiling with new printf() statements to figure out what is happening in your program, use a debugger such as *gdb* to inspect and modify your program while it is running.
  - Use memory-aware tools, such as *valgrind*, to detect memory leaks or bad memory accesses.
  - Use sanitizers when you are building your tests (e.g., ASAN, UBSAN). This is easy if you use the given makefile. For example, to use the address sanitizer, build your tests by running `make checkprogs CFLAGS=-fsanitize=address LDFLAGS=-fsanitize=address` (you may need to run make clean first, if you already built without sanitizers).

Submission Guidelines:

Your parser must be written in C and run on Linux. Submit your *shell_parser.c* file to Gradescope. You will receive your submission’s grade shortly after submitting the solution.

Although **no grade is applied to this assignment**, it is in your best interest to ensure that all test cases pass. If you can get all test cases to pass for this assignment, there will be fewer bugs to fix in your final solution for project 1.

Your final submission for the next assignment must also include a README file, so we recommend that you start working on the README as part of this assignment. If you relied on any external resources to help complete this assignment, note the resource and the challenge it helped resolve. Use this file to provide any additional notes you would like to share with instructors about your submission.

Additional files can be present in the submission, but they will not be used by the autograder.