Parsing

Where to Start?

- Looking at this naively, you'll need to
  - Setup
  - Print prompt
  - Read in user input
  - Transform it to commands, files, and symbols
  - Match to a pattern
  - Execute command
  - Print results
  - Cleanup
- You'll need to do this for each line of input

Where to Start?

```
int main() {
    while (1) {
        //Setup
        //Print prompt
        //Read input
        //Transform input
        //Match against patterns
        //Execute command
        //Print results
        //Cleanup
    }
    Return 0;
}
```

- This is a REPL
  - Read-eval-print-loop
- Used within most text-based interactive processes
  - Lisp
  - Scripting languages
  - Shells
  - Query languages
  - Text-based games

Where to Start?

```
int main() {
    char *line;
    char **cmd;

    while (1) {
        my_setup();
        my_prompt();
        line = my_read();
        cmd = my_parse(line);
        my_execute(cmd);
        my_clean();
    }

    return 0;
}
```

```
void my_setup() {}
void my_prompt() {}

char *my_read () {
    return NULL;
}
char **my_parse (char *line) {
    return NULL;
}

void my_execute (char ** cmd) {
    //Match against patterns
    //Execute based on pattern
    //Print results
}

void my_clean () {}
```

Where to Start?

```
int main() {
    char *line;
    char **cmd;

    while (1) {
        my_setup();
        my_prompt();
        line = my_read();
        cmd = my_parse(line);
        my_execute(cmd);
        my_clean();
    }

    return 0;
}
```

```
void my_setup() {}
void my_prompt() {}

char *my_read () {
    return NULL;
}
char **my_parse (char *line) {
    return NULL;
}

void my_execute (char ** cmd) {
    //Match against patterns
    //Execute based on pattern
    //Print results
}

void my_clean () {}
```

my_read()

- Get line of data from stdin
  - Can use fgets
  - Make sure to check return value
- Return it as a c-string
- If dynamically created,
  - Make sure to free it later (cleanup)
  - Otherwise, you'll introduce a memory leak each iteration

my_parse()

- Takes in a c-string of input
  - Line from read
- Returns an array of c-strings
  - Each of the command arguments in a separate cell
- Parsing is necessary because
  - You need to strip excess whitespace
  - You need to split up the arguments
  - You need to expand environmental variables
  - You need to resolve pathnames

my_parse()

```
char **my_parse(char *line) {
    char **args;

    line = parse_whitespace(line);
    args = parse_arguments(parsed);
    args = expand_variables(args);
    args = resolve_paths(args);

    return cmd;
}
```

```
char **parse_whitespace(char *line) {
    return NULL;
}

char **parse_arguments(char *line) {
    return NULL;
}

char **resolve_paths(char **args) {
    return NULL;
}

char **expand_variables(char **args) {
    return NULL;
}
```

parse_whitespace()

- Takes in a c-string
- Returns the same c-string after adding/removing whitespace
- Transforms c-string such that there is exactly one space between each argument
  - To later parse out the arguments into an array

parse_whitespace()

- Use cases to consider
  - Leading white space
    - Remove until the first argument is at the 0^{th} slot of the c-string
  - Trailing white space
    - Remove until the last character of the last argument is the $n-1^{th}$ character of the c-string
      - nth character is '\0'
  - Extra white space between arguments
    - When there is multiple spaces, you'll need to remove all but one

parse_whitespace()

- Use cases to consider
  - No white space between arguments
    - Obviously can't handle cases like *file1file2*
      - Instead you should just assume it is one argument
      - Then when it (likely) doesn't exist, you'd return an error
      - Alternatively, you can detect these cases if there are not enough arguments to the command
    - But for cases involving special characters...
      - <*file*, *cmd*|, *cmd*|>*file*, etc
      - You'll need to add a space between the special character and the other argument
      - Special characters include: |, <, >, &, $, ~
      - Do **not** do this for: ., /

parse_whitespace()

- Use cases to consider
  - 'Extra' characters
    - The writeup specifies that you do not have to handle
      - Escaping characters
      - Regular expressions
      - Quoted strings
    - However, you should not remove these if they are in the original input
      - While rare, filenames can contain these characters
      - Typically you'd have to escape them, but you don't have to worry about that in this simple shell
    - You **do** need to worry about accidentally **adding** special characters
      - For example, fgets will place the newline separator in the input string
      - This will cause those arguments to be wrong

parse_arguments()

- Takes in a c-string
  - Represents the input command with augmented whitespace
- Returns an array of c-strings
  - Represents the arguments of the command in separate cells
- Here you will traverse the input string placing the characters into different buckets of the output array
  - The bucket to place the characters in is determined by the number of spaces encountered
  - You can copy each character one by one or you can scan and then do a strncpy using a offset
- The difficulties here will be
  - C-string semantics
  - Knowing how much space to allocate for each bucket
  - Remembering to free later (cleanup)