Normandy Shell

By Norman Li

Features

- Basic Shell Functionality
- Parse Multiple Commands with `;`
- Simple Redirection
  - `stdin (<)`
  - `stdout (>)`
  - `stdout [append] (>>)`
  - `pipe (|)`
- Arbitrarily Large User Input
- Ignores Redundant Spaces
- Prints prompt in *bash* format:
  - `<user>@<hostname>:<current working directory>$`
    - If the user's "home directory substring" is part of "current working directory", it is replaced with a ~.

Bugs/Missing Features

- No tab completion
- No command history
- There can only be one redirection operator at a time.
- Files created in `/tmp` directory are not cleared

Function Headers & Files

NOTE:

- Header files are located in `include/`
- C files are located in `src/`
- Object files are located in `build/`

prompt.c/h

```
// Get Username Function
/* Explanation:
   Obtains the username running the shell.
*/
/* Arguements: None
   Returns (char *): The username of the process owner
*/
char *get_username()
```

```
// Get Hostname Function
/* Explanation:
   Obtains the hostname of the machine running the shell.
*/
/* Arguements: None
   Returns (char *): The Hostname of the Machine
*/
```
char *get_hostname()

```
// Get Current Working Directory Function
/* Explanation:
   Outputs the formatted current working directory.
   If path contains the home directory, it is replaced with
   "~/".
*/
/* Arguements: None
   Returns (char *): The Current Working Directory (Formated)
*/
```
char *get_cwd()

```
// Print Prompt Function
/* Explanation:
   Prints the prompt for the shell.
   Format:
   <username>@<hostname>:<cwd>$
*/
/* Arguements: None
   Returns (void)
*/
```
void print_prompt()

---

cd.c/h

```
// Change Directory Function
/* Explanation:
   Changes the current directory to the one specified
   in the arguement (char *) "path". Will print an error
   if an error occurs."
*/
```

```
/* Arguements:
   -(char *): Pathname of directory to change to
   Returns (void)
*/
```
`void cd(char *path)`

---

control.c/h
```
// Control Flow Function
/* Explanation:
   The "Control Center" of the whole shell.
   This function controls the flow of the program
*/
/* Arguements: None
   Returns (void)
*/
```
`void control_flow()`

---

error.c/h
```
// Print Error Function
/* Explanation:
   Prints a formatted error string describing the error,
   given (char *) "command" and (int) "error".
*/
/* Arguements:
   -(char *): Command that gave the error
   -(int): Errno
*/
```
`void print_error(char *command, int error)`

---

execute.c/h
```
// Run Commands Function
/* Explanation:
   Given a linked list of commands, this function
   will run them sequentially.
*/
/* Arguements:
   -(struct node*): The List of Commands
   Returns (void)
*/
```

```
void run_commands(struct node* commands)

// Execute Command Function
/* Explnation:
   Given (char **) "arguements", the parent (shell) will fork
   off a child and the child will execute the command w/
arguements.
   An error will be displayed if an error occurs.
*/
/* Arguements:
   -(char **): Array of Arguements
   Returns (void)
*/
static void execute_command(char **arguements)

// Redirect Stdout Function
/* Explanation:
   This function redirects stdout to a flie (truncate).
   array[0] = command
   array[1] = file
*/
/* Arguements:
   -(char **): Array of elements
   Returns (void)
*/
static void redirect_stdout(char **array)

// Redirect stdin Function
/* Explanation:
   This function redirects stdin to a flie.
   array[0] = command
   array[1] = file
*/
/* Arguements:
   -(char **): Array of elements
   Returns (void)
*/
static void redirect_stdin(char **array)
```

```
// Redirect Append Function
/* Explanation:
   This function redirects stdout to a flie (append).
   array[0] = command
   array[1] = file
*/
/* Arguements:
   -(char **): Array of elements
   Returns (void)
*/
```
static void redirect_append(char **array)

```
// Redirect Pipe Function
/* Explanation:
   This function redirects cmmand0's stdout
   to command1's stdin.
*/
/* Arguement:
   -(struct node*): List of Commands to pipe
   Returns (void)
*/
```
static void redirect_pipe(struct node* list)

---

input.c/h

```
// Get Input Function
/* Explanation:
   This function grabs the input from the terminal/console
   character by character until it reaches EOF (End of File).
*/
/* Arguements: None
   Returns (void)
*/
```
char *get_input()

---

list.c/h

```
// Insert Node Function
/* Explanation:
   Given a (char *) string, this function inserts
   a node with the string at the end of the linked
   list.
*/
/* Arguements:
   -(struct node*): The head node of the list
   -(char *): The string to be added
   Returns (void)
*/
```
`struct node* insert_node(struct node* head, char *data)`

```
// Free List Function
/* Explanation:
   Given the head node, this function will free the entire list,
   include the malloc'ed "data" strings.
*/
/* Arguements:
   -(struct node*): Head Node of List
   Returns (void)
*/
```
`void free_list(struct node* head)`

```
// Return String Array
/* Explanation:
   Given the head node, this function will return the array of
   character arrays of the linked list
*/
/* Arguements:
   -(struct node*): The Head Node
   Returns (char **): The array of character arrays
*/
```
`char **return_string_array(struct node* head)`

```
// Free String Array
/* Explanation:
   Given a "string array" generated from the function "(char **)
return_string_array",
   this function will free all elements of the array, including
the malloc'ed strings
   themselves.
*/
```

```
/* Arguements:
   -(char **): The array to be freed
   Returns (void)
*/
```
`void free_string_array(char **string_array)`

---

parse.c/h
```
// Remove Trailing Whitespace Function
/* Explanation:
   This function removes trailing whitespace (front/back).
*/
/* Arguements:
   -(char *): The input string
   Returns (char *): The modified string
*/
```
`char *remove_trailing_whitespace(char *input)`

```
// Remove Extra Function
/* Explanation:
   Given (char *) "input", this function returns a string (char
*)
   without extra whitespace and without newlines.
   Example:
   remove_extra_whitespace("chicken   have  heads .\n" ->
"chicken have heads ."
*/
/* Arguements:
   -(char *): String to be modified
   Returns (char *): Modified String
*/
```
`char *remove_extra(char *input)`

```
// Split Array Function
/* Explanation:
   Given a string and a delimiter, this function
   will split the string on the delimter and return
   an array of strings.
*/
/* Arguements:
   -(char *): String to split
   -(char *): Delimiter
   Returns (char **): String Array
*/
static char **split_array(char *string, char *delim)

// Split List Function
/* Explanation:
   Given a string and a delimiter, this function
   will split the string on the delimter and return
   an linked list.
*/
/* Arguements:
   -(char *): String to split
   -(char *): Delimiter
   Returns (struct node*): Linked List
*/
static struct node* split_list(char *string, char *delim)

// Arguementify Function
/* Explanation:
   Given (char *) "command", which is the command given to the
shell,
   this function will return an array of strings of arguements,
   if any.
*/
/* Arguements:
   -(char *): The command to be parsed.
   Returns: (char **): The arguements
*/
char **arguementify(char *command)

// Split on Semicolon Function
/* Explanation:
   Given the shell input, this function will parse
   the commands semicolon by semicolon, if any.
*/
/* Arguements:
   -(char *): Shell input
   Returns (struct node*): List of commands
*/
struct node* split_on_semicolon(char *input)

// Split on stdout Function
/* Explanation:
   Given the command, this function splits it into the
   "command" portion and the "file" portion.
*/
/* Arguements:
   -(char *): The command
   Returns (char **): The "command' and "file"
*/
char **split_on_stdout(char *command)

// Split on stdin Function
/* Explanation:
   Given the command, this function splits it into the
   "command" portion and the "file" portion.
*/
/* Arguements:
   -(char *): The command
   Returns (char **): The "command' and "file"
*/
char **split_on_stdin(char *command)

// Split on Append Function
/* Explanation:
   Given the command, this function splits it into the
   "command" portion and the "file" portion.
*/
/* Arguements:
   -(char *): The command
   Returns (char **): The "command' and "file"
*/
char **split_on_append(char *command)

// Split on Pipe Function
/* Explanation:
   Given the command, this function splits it into
   "command0" and "command1".
*/
/* Arguements:
   -(char *): The command
   Returns (char **): The "command0' and "command1"
*/
```
`struct node* split_on_pipe(char *command)`