MYSH

Technical Documentation

Dawson Geist

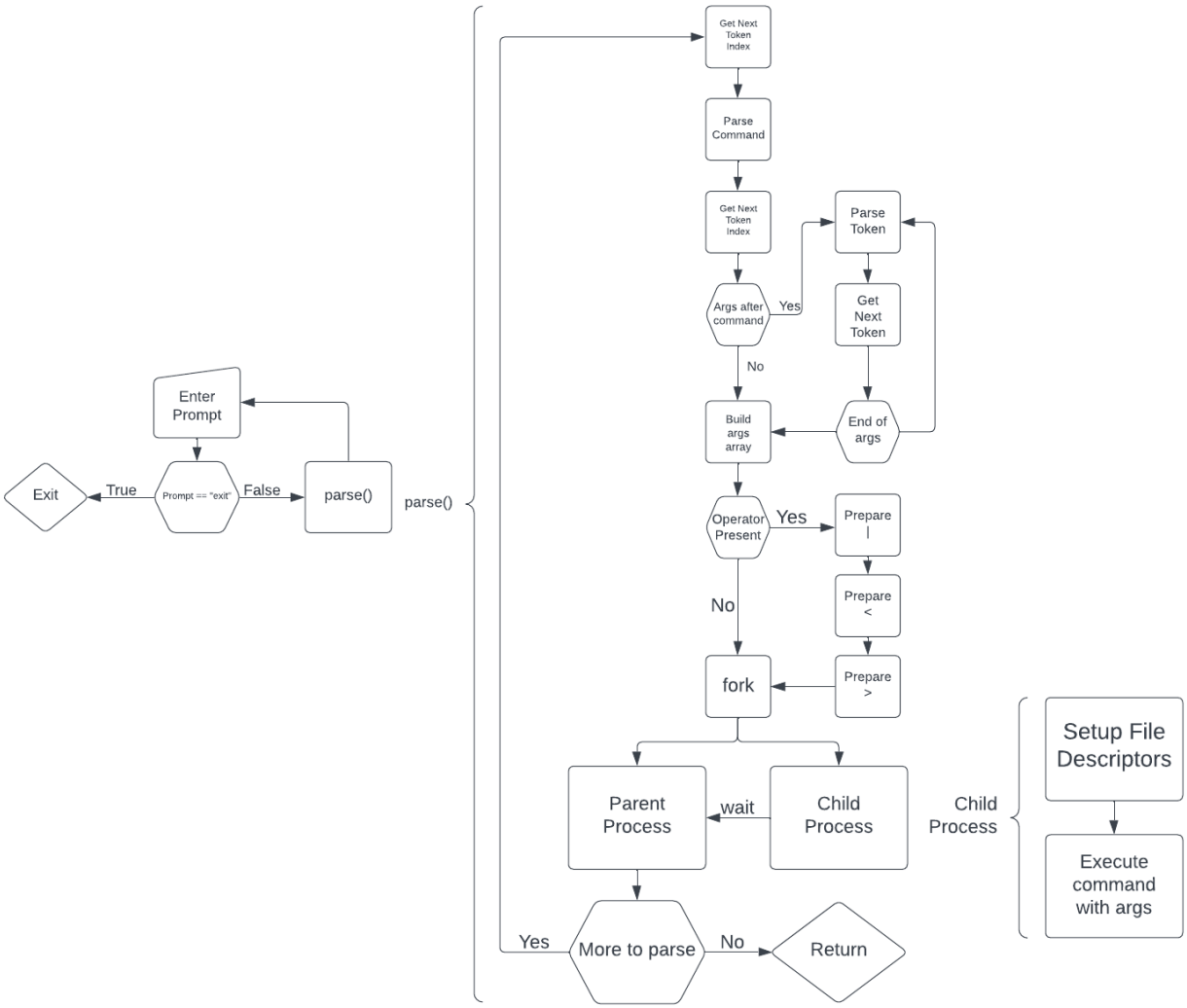
SUNY University at Albany

3/20/2021

Table of Contents

System Documentation

Data Flow Diagram



Functions

- int main():
  - Entry point for the program. Holds the run-loop that will keep the shell alive until the exit command is entered.
- int run():
  - Responsible for printing the mysh> command prefix as well as grabbing the user-entered command prompt from stdin.
- char * parse(char* prompt, int prompt_index, int in_fd):
  - Responsible for recurssively parsing and executing the command prompt in chunks based off of the operators |, <, >.
- char * parseToken(char* prompt, int * promptIndex):
  - Helper Function. Given the full prompt and the starting index of the word/keyword of interest, add non whitespace characters to a character array and return the array. The address of promptIndex is passed into the function so that promptIndex can be dereferenced and parsing progress can be tracked across functions.
- int getNextTokenIndex(char* prompt, int * tokenIndex):
  - Helper Function. Given the full prompt and the starting index of the whitespace seperator of interest, increment the token index everytime you see a whitespace character at the tokenIndex of prompt. This function allows keywords in the command prompt to be seperated by 1 or more whitespace characters.
- void testParse():
  - Test Function. Store a number of prompts used for debugging and testing the Parse function. Successful calls to parse should execute the command prompt and receive the exact results returned by using the native shell.
- void runTests():
  - Test Function. Calls test functions.

Implementation Details

The entire program was implemented under the idea that it is impossible to know what the user will enter (with respect to the command) but whatever they enter it will be correct. Because of this the function works recursively so that 1 -> N commands can be chained together with pipes. These commands and their parameters can be spaced by variable amounts of whitespace and there will be no issue. This is done by using a global integer as a pointer to the current character in the command prompt. By dereferencing and incrementing this pointer whenever we traverse over whitespace or command prompt tokens the program always ends at the start of a new token or at the end of the command. Since we pass this index value into the function we can recursively call the function and know the program will not execute a previously parsed section of the command prompt.

Test Documentation

How it was Tested

The majority of the testing for this program came with running of the test prompts in the testParse() function. These command prompts tested a wide range of functionality that was implemented in this shell. Other sections of the code were debugged using print statements to print the values of variables before and after changes. This was a fundamental process throughout the development of this shell.

Test Set

prompt = " cat country.txt city.txt | egrep 'g' | sort | more > countryCitygSorted.txt";

Output:

(in countryCitygSorted.txt)

angola

beijing

chongqing

germany

hong kong

nicaragua

shanghai

prompt = "cat country.txt city.txt | more";

Output:

zimbabwe

russia

australia

brazil

china

denmark

germany

france

angola

italy

japan

korea

poland

mexico

nicaragua

miami

shanghai

albany

chongqing

tokyo

beijing

detroit

new york

hong kong

macau

prompt = "ls | sort";

Output:

city.txt

country.txt

Documentation.txt

mysh

mysh.c

test2.txt

test3.txt

test4.txt

test.txt

prompt = "cat country.txt city.txt | sort | sort | more";

Output:

albany

angola

australia

beijing

brazil

china

chongqing

denmark

detroit

france

germany

hong kong

italy

japan

korea

macau

mexico

miami

new york

nicaragua

poland

russia

shanghai

tokyo

zimbabwe

prompt = "cat country.txt city.txt | egrep g | more";

Output:

germany

angola

nicaragua

shanghai

chongqing

beijing

hong kong

prompt = "cat country.txt city.txt | egrep g | sort | more";

Output:

angola

beijing

chongqing

germany

hong kong

nicaragua

shanghai

prompt = "egrep g | more";

Output: * Working egrep Command *

prompt = "cat country.txt city.txt | sort | more > test.txt";

Output:

(in test.txt)

albany

angola

australia

beijing

brazil

china

chongqing

denmark

detroit

france

germany

hong kong

italy

japan

korea

macau

mexico

miami

new york

nicaragua

poland

russia

shanghai

tokyo

zimbabwe

prompt = "sort < city.txt > test2.txt";

Output:

(in test2.txt)

albany

beijing

chongqing

detroit

hong kong

macau

miami

new york

shanghai

tokyo

prompt = "egrep g test.txt > test3.txt";

Output:

(in test3.txt)

angola

beijing

chongqing

germany

hong kong

nicaragua

shanghai

prompt = "cat country.txt city.txt | tail | sort -r | wc > test4.txt";

Output:

(in test4.txt)

```
    10      11      74
```

Prompt = “cat country.txt city.txt | egrep ‘g’ | sort | wc -l > countryCitygCount.txt”;

Output:

(in countryCitygCount.txt)

7

User Documentation

How to Run

After downloading the source code file the object file needs to be created.

1. Navigate to the directory where the source code is located.
2. Enter the following command
    a. gcc -o mysh mysh.c
3. After the previous command executes you can run the program by entering the following command
    a. ./mysh

You are then free to type any native Linux command into the mysh shell. When you want to exit:

1. Enter "exit" into the shell and press enter. The string has to be "exit" exactly (excluding quotation marks)
    a. NOTE: IF EXECV FAILS YOU HAVE TO ENTER "EXIT" MULTIPLE (number of times execv failed + 1) TIMES BEFORE YOU CAN EXIT

NOTE: egrep works by looking for the search parameter exactly. Ex egrep “g” will look for “g” in your data. Egrep g will look for g in your data. This is an issue when looking for a sentence that contains whitespaces such as “Operating Systems are fun” will look for “Operating.

Parameters

Mysh.c does not accept any optional parameters.

Files

City.txt

miami

shanghai

albany

chongqing

tokyo

beijing

detroit

new york

hong kong

macau

Country.txt

zimbabwe

russia

australia

brazil

china

denmark

germany

france

angola

italy

japan

korea

poland

mexico

nicaragua

countryCitygSorted.txt

angola

beijing

chongqing

germany

hong kong

nicaragua

shanghai

countryCitygCount.txt

7

Source Code

mysh.c

```
#include <stdio.h> // for IO
#include <stdlib.h>// for memory allocation
#include <string.h> // for string functions
#include <fcntl.h> // for open
#include <unistd.h> // for close
#include <sys/types.h> // for pipes
#include <sys/wait.h> // for wait

// Global Variables
int pipeInput = 0;
int childPipedOutput = 0;

char * parseToken(char* prompt, int * promptIndex)
{
	//printf("Entering parseToken\n");
	char * token = malloc(sizeof(char) * 50);
	int tokenIndex = 0;
	while(*promptIndex < strlen(prompt) && prompt[*promptIndex] != ' ')
	{
		if(prompt[*promptIndex] != '\n')
		{
			//printf("WRITING CHAR:%c\n",prompt[*promptIndex]);
			// store the current character and advance the index
			token[tokenIndex++] = prompt[*promptIndex];
		}
		*promptIndex+=1;
	}
	// Manually null terminate this string
	token[tokenIndex] = '\0';
	//printf("Exiting parseToken\n");
	return token;
}

int getNextTokenIndex(char* prompt, int * tokenIndex)
{
	//printf("Entering getNextTokenIndex\n");
	//printf("Token Index: %d, prompt Length: %ld", *tokenIndex, strlen(prompt));
	if(*tokenIndex < strlen(prompt))
	{
		// Advance prompt_index to the beginning of the command
		while(prompt[*tokenIndex] == ' ')
		{
			*tokenIndex+=1;
		}
	}
	//printf("Exiting getNextTokenIndex\n");
}

void parse(char* prompt, int prompt_index, int in_fd)
```

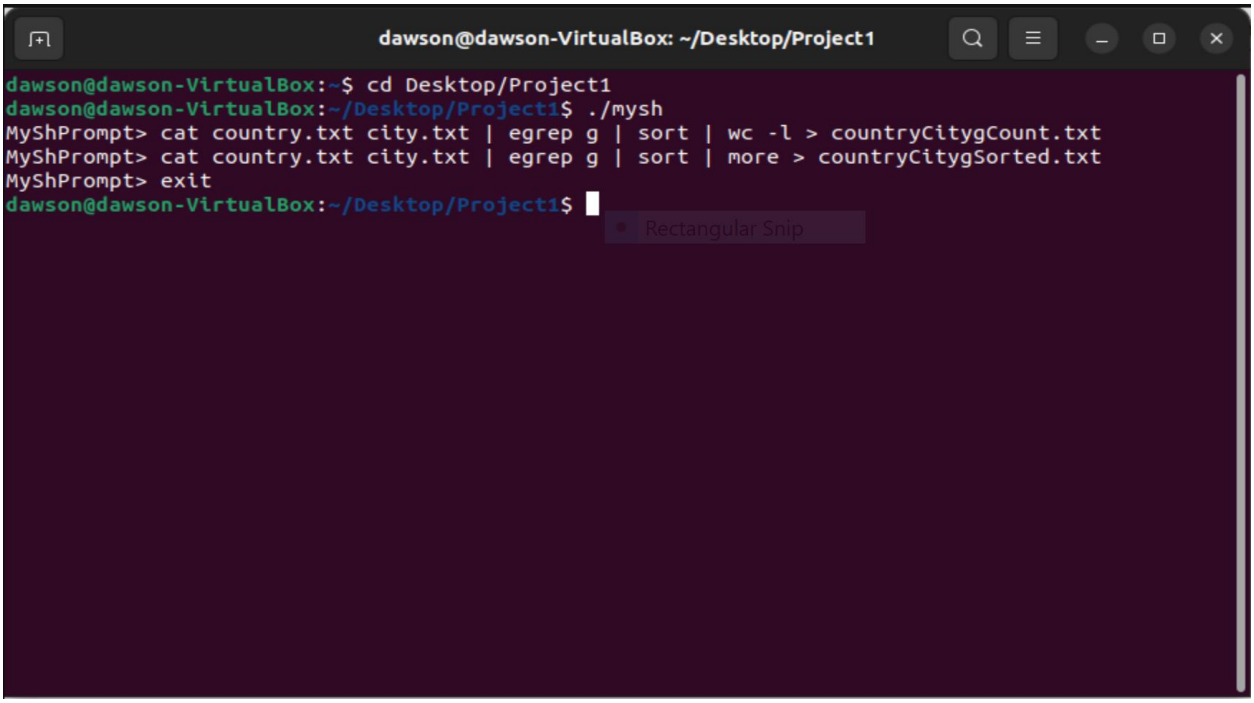
```
{
	//printf("LENGTH OF PROMPT: %lu\n", strlen(prompt));
	int numArgs = 1;
	int argIndex = 0;
	char ** args;
	char * arg;
	char * command;
	char * redirectInputFile;
	char * redirectOutputFile;
	int redirectInput = 0;
	int redirectOutput = 0;
	int pipeOutput = 0;
	int fd_redirectionInput;
	int fd_redirectionOutput;
	int fd_pipe[2];

	//if the last child proccess piped its output, prepare to take it's results as
input using in_fd
	if(childPipedOutput)
	{
		pipeInput = 1;
		//Reset childPipedOutput
		childPipedOutput = 0;
	}

	// Get the starting index of the first token (assuming its command)
	getNextTokenIndex(prompt, &prompt_index);
	command = parseToken(prompt, &prompt_index);
	//printf("Parsed Command:%s\n",command);
	//Increment number of args
	numArgs +=1;
	// Get the next non-whitespaceCharacter
	getNextTokenIndex(prompt, &prompt_index);
	argIndex = prompt_index;

	//Count the number of arguments in the prompt
	while(prompt_index < strlen(prompt) && prompt[prompt_index] != '|' &&
prompt[prompt_index] != '<' && prompt[prompt_index] != '>')
	{
		arg = parseToken(prompt, &prompt_index);
		//printf("Arg to be Added: %s\n",arg);
		getNextTokenIndex(prompt, &prompt_index);
		numArgs += 1;
	}

	// Create Array of args
	args = (char**)calloc(numArgs, sizeof(char*));
	// add comand
	char path[20] = "/usr/bin/";
	strcat(path,command);
	command = path;
	args[0] = calloc(50, sizeof(char));
	args[0] = command;
	int i = 1;

	//Build our Argument Array
	while(argIndex < strlen(prompt) && prompt[argIndex] != '|' && prompt[argIndex] !=
'<' && prompt[argIndex] != '>')
```

```
	{
		arg = parseToken(prompt, &argIndex);
		args[i] = calloc(50, sizeof(char));
		args[i] = arg;
		i++;
		getNextTokenIndex(prompt, &argIndex);
	}

	// Add Null terminated parameter
	args[i] = NULL;

	// Pipe Operator
	if(prompt[prompt_index] == '|')
	{
		prompt_index +=1;
		pipeOutput = 1;
		childPipedOutput=1;

	}
	// Prepare the redirection Input
	if(prompt[prompt_index] == '<')
	{
		prompt_index +=1;
		getNextTokenIndex(prompt, &prompt_index);
		redirectInputFile = parseToken(prompt, &prompt_index);
		redirectInput = 1;
		getNextTokenIndex(prompt, &prompt_index);

	}
	// Prepare the Redirection Output
	if(prompt[prompt_index] == '>')
	{
		prompt_index +=1;
		getNextTokenIndex(prompt, &prompt_index);
		redirectOutputFile = parseToken(prompt, &prompt_index);
		redirectOutput = 1;
		getNextTokenIndex(prompt, &prompt_index);
	}

	pipe(fd_pipe);
	//printf("fd_pipe[0]: %d\nfd_pipe[1]: %d\nin_fd: %d\n",
fd_pipe[0],fd_pipe[1],in_fd);
	int pid = fork();
	if(pid == 0)
	{
		//printf("\tIN CHILD PROCESS\n");
		//close(fd_pipe[0]);
		if(redirectInput)
		{
			//printf("\tREDIRECTING INPUT\n");
			// Open the numbers.txt file. Give user permision to r/w/e
			fd_redirectionInput = open(redirectInputFile, O_RDONLY, S_IRWXU);
			dup2(fd_redirectionInput, STDIN_FILENO);
			close(fd_redirectionInput);
		}
		else if(pipeInput)
		{
			//printf("\tPIPE INPUT\n");
```

```
                        dup2(in_fd, STDIN_FILENO);
                        close(in_fd);
                }
                if(redirectOutput)
                {
                        //printf("\tREDIRECTING OUTPUT\n");
                        // Open the numbers.txt file. Give user permision to r/w/e
                        //printf("\tFile output Name: %s\n", redirectOutputFile);
                        fd_redirectionOutput = open(redirectOutputFile, O_WRONLY | O_CREAT,
S_IRWXU);
                        //printf("\tFile output FD: %d\n", fd_redirectionOutput);
                        dup2(fd_redirectionOutput, STDOUT_FILENO);
                        close(fd_redirectionOutput);
                }
                else if(pipeOutput)
                {
                        //printf("\tPIPE OUTPUT\n");
                        dup2(fd_pipe[1], STDOUT_FILENO);
                }


                execv(args[0], args);
                printf("EXECV FAILED\n");
        }
        else
        {
                wait(NULL);
                //printf("IN PARENT PROCESS\n");
                //printf("fd_pipe[0]: %d\nfd_pipe[1]: %d\nin_fd: %d\n",
fd_pipe[0],fd_pipe[1],in_fd);
                close(fd_pipe[1]);
                //close(in_fd);
                //printf("fd_pipe[0]: %d\nfd_pipe[1]: %d\nin_fd: %d\n",
fd_pipe[0],fd_pipe[1],in_fd);
                if(prompt_index < strlen(prompt))
                {
                        //printf("ENTERING NEXT PARSE\n");
                        parse(prompt,prompt_index, fd_pipe[0]);
                }
                //printf("EXITING PARSE\n");
        }
}

void testParse()
{
        //printf("---START testParse:\n");

        char * prompt;

        //prompt = " cat country.txt city.txt | egrep 'g' | sort | more >
countryCitygSorted.txt";
        //prompt = "cat country.txt city.txt | more";
        //prompt = "ls | sort";
        //prompt = "cat country.txt city.txt | sort | sort | more";
        //prompt = "cat country.txt city.txt | egrep 'g' | more";
        //prompt = "cat country.txt city.txt | egrep 'g' | sort | more";
        //prompt = "egrep 'g' | more";
        //prompt = "cat country.txt city.txt | sort | more > test.txt";
```

```
	//prompt = "sort < city.txt > test2.txt";
	//prompt = "egrep g test.txt > test3.txt";
	//prompt = "cat country.txt city.txt | tail | sort -r | wc > test4.txt";
	//prompt = "ls";

	//printf("PROMPT: %s\n", prompt);
	parse(prompt,0, STDIN_FILENO);

	//printf("---END testParse:\n");
}

void runTests()
{
	testParse();
}

int run()
{
	int exit = -1;
	int MAX_PROMPT_LENGTH = 250;
	char prompt[MAX_PROMPT_LENGTH];
	printf("MyShPrompt> ");

	//Get command from STDIN
	fgets(prompt,MAX_PROMPT_LENGTH,stdin);

	//printf("prompt:%s\n", prompt);
	if(strcmp(prompt, "exit\n") == 0)
	{
		exit = 1;
	}
	else
	{
		//Begin Parsing
		parse(prompt,0,STDIN_FILENO);
	}

	return exit;
}

int main()
{
	for(int i = 0; i < 1; i++)
	{
		i += run();
	}
}
```

Output


dawson@dawson-VirtualBox: ~/Desktop/Project1
dawson@dawson-VirtualBox:~$ cd Desktop/Project1
dawson@dawson-VirtualBox:~/Desktop/Project1$ ./mysh
MyShPrompt> cat country.txt city.txt | egrep g | sort | wc -l > countryCitygCount.txt
MyShPrompt> cat country.txt city.txt | egrep g | sort | more > countryCitygSorted.txt
MyShPrompt> exit
dawson@dawson-VirtualBox:~/Desktop/Project1$
Rectangular Snip