Programming Assignment 4 (PA4) - mygrep

Milestone Due: **Wednesday, May 29 @ 11:59 pm**
Final Due: **Wednesday, June 5 @ 11:59 pm**

Assignment Overview	Getting Started	Example Input	Detailed Overview	Milestone Functions
Post-Milestone Functions	Unit Testing	README File	Extra Credit	Turnin Summary

Assignment Overview

The purpose of this (last!) programming assignment is to further emphasize C and implement a real Unix command (along with using more Standard C Library routines). You will build a program called mygrep that will function very similarly to the real Unix command **grep**. This program will search for a pattern in a list of files and print each line that matches the pattern.

The purpose of this assignment is to gain familiarity with the unix command grep, regular expressions, bit patterns, structs, pointers, dynamic memory allocation, resizing arrays, string operations, and file operations.

We will try not to do too much hand-holding in this PA. You have some experience now and you need to learn how to look up information yourself. You will not be given nearly as much help in the upper-division classes, so now is the time to start doing more on your own.

IMPORTANT NOTE for Assembly routines:

1. **(NEW INFORMATION, DO NOT SKIP READING)**: Make sure you **do not** use registers other than **r0-r3** in your assembly functions. If you are calling a function that requires more than four arguments, **pass in any extra arguments on the stack**. -- **They should be described in function header's stack variable list as well.**
2. Only **fp, lr** are pushed to the stack.
3. Note that values in **r0-r3** will not be preserved after function calls.
4. Remember to note in the **function header** what stack variables you used (i.e somevar -- [fp - 8] -- for calculating length, etc). Stack variables include local variables, formal parameters, and additional arguments. Refer to style guidelines for the specific way to format the header.

Grading

- **README: 10 points** - See README Requirements here and questions below
  - http://cseweb.ucsd.edu/~ricko/CSE30READMEGuidelines.pdf
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 10 points** - See Style Requirements here
  - http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidelines.pdf
- **Correctness: 75 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
  - Make sure you have all files tracked in Git.
- **Extra Credit: 7 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

Gathering Starter Files:
The first step is to gather all the appropriate files for this assignment. Connect to pi-cluster via ssh.

```
$ ssh cs30xyz@pi-cluster.ucsd.edu
```

Create and enter the pa4 working directory.

```
$ mkdir ~/pa4
$ cd ~/pa4
```

Copy the starter files from the public directory.

```
$ cp ~/../public/pa4StarterFiles/* ~/pa4/
```

Starter files provided:

pa4.h	pa4Strings.h	
test.h	testprocessArgs.c	Makefile

Preparing Git Repository:
You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.

Example Input

What is the Grep Command? & Regular Expression Overview
The best way to learn about the grep command is to try it out! We'll be trying out the various options for grep on files in this directory:

```
~/../public/grepPractice/
```

You can search for words in a file like so:

```
grep chips ~/../public/grepPractice/sample.txt
You could put carob chips on there.
```

In this example, grep searches for all instances of the pattern "chips" in the file sample.txt.

Grep can also search through multiple files. Here's an example of how you could use grep on multiple files. Note that though it looks like we are typing a single file argument, the shell glob operator (`*`) will expand into all of the filenames in the directory:

```
grep TODO ~/../public/grepPractice/*
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/sample.txt:// TODO: needs title
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/testprocessArgs.c: * Author: TODO
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/testprocessArgs.c: * UserId: TODO
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/testprocessArgs.c: * Date: TODO
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/testprocessArgs.c: * Sources of help:
TODO
/home/linux/ieng6/cs30x/cs30xyz/../public/grepPractice/testprocessArgs.c:   * TODO: YOU MUST
WRITE MORE TEST CASES FOR FULL POINTS!
```

Grep also works with regular expressions ("grep": **g**lobal **r**egular **e**xpression search & **p**rint). A regular expression is a sequence of characters that define a search pattern. Instead of searching for just a plain string, a regular expression can be designed for powerful searches, such as searching for phone numbers (in a variety of formats) in a text. Here's how you could use grep to search for all 4 digit numbers that are separated by word boundaries:

```
grep '\<[0-9]\{4\}\>' ~/../public/grepPractice/sample.txt
We're going 0900 at J-Gate.
I might be. It all depends on what 0900 means.
Oh, sweet. That's the one you want. The Thomas 3000!
All right. Case number 4475,
Mr. Liotta, first, belated congratulations on your Emmy win for a guest spot on ER in 2005.
```

For this assignment, you won't need to know how to design regular expressions, but your program must still be able to handle regular expressions (we will use a set of standard library functions for this).

Grep has a number of options/flags that modify its behavior slightly. You can look at the manpage for grep (`man -s1 grep`) for more information. Here are the grep options that you'll have to implement in your assignment:

The -i flag

This flag will ignore case distinctions, so that characters that differ only in case match each other. Compare the difference without the -i flag:

```
grep CiNNaBoN ~/../public/grepPractice/sample.txt
```

which has no matches, versus with the -i flag:

```
grep -i CiNNaBoN ~/../public/grepPractice/sample.txt
You know what a Cinnabon is?
```

The -v flag

This flag will only display non-matching lines. Compare the difference without the -v flag:

```
grep start ~/../public/grepPractice/secret.txt
start
start
```

and with the -v flag:

```
grep -v start ~/../public/grepPractice/secret.txt
early
often
```

The -n flag

This flag will print the line number along with matching lines. Compare the difference without the -n flag:

```
grep heart ~/../public/grepPractice/sample.txt
Yes, but who can deny the heart that is yearning?
Not in this fairy tale, sweetheart.
```

and with the -n flag:

```
grep -n heart ~/../public/grepPractice/sample.txt
512:Yes, but who can deny the heart that is yearning?
1143:Not in this fairy tale, sweetheart.
```

The -c flag

This flag will display the number of times the pattern appears, without the lines that matched. Compare the difference without the -c flag:

```
grep heat ~/../public/grepPractice/sample.txt
```


Back to Top

```
It's bread and cinnamon and frosting. They heat it up...
There's heating, cooling, stirring. You need a whole Krelman thing!
There's a little left. I could heat it up.
Yeah, heat it up, sure, whatever.
```

and with the -c flag:

```
grep -c heat ~/../public/grepPractice/sample.txt
4
```

The -e flag

This flag will allow you to reorder where you specify the pattern. Usually the pattern is given before the file arguments. Consider the error that grep outputs when the pattern and the file(s) to search are switched:

```
grep ~/../public/grepPractice/sample.txt heart
grep: heart: No such file or directory
```

compared to the successful execution of the command with the -e flag:

```
grep ~/../public/grepPractice/sample.txt -e heart
Yes, but who can deny the heart that is yearning?
Not in this fairy tale, sweetheart.
```

Flags can be used together

You can also combine flags:

```
grep -nv start ~/../public/grepPractice/secret.txt
2:early
4:often
```

While your program will closely emulate the grep command, there may be some slight differences in output and behavior, so you must make sure that your program exactly matches **the public executable** to get full credit:

Sample Executable

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/mygreptest
```

NOTE:

1. The output of your program MUST match exactly as it appears in the `pa4test` output. You need to pay attention to everything in the output, from the order of the error messages to the small things like extra newlines at the end (or beginning, or middle, or everywhere)!
2. **We are not giving you any sample outputs, instead you are provided some example inputs. You are responsible for trying out all functionality of the program; the list of example inputs is not exhaustive or complete. It is important that you fully understand how the program works and you test your final solution thoroughly against the executable.**

Below, `invalidFilename` refers to a file that does not exist.

Example input that has error output:

```
cs30xyz@pi-cluster-001:pa4$ ./mygrep -f hello
cs30xyz@pi-cluster-001:pa4$ ./mygrep -c foo invalidFilename
```

```
cs30xyz@pi-cluster-001:pa4$ ./mygrep -h
```

Example input that has normal output:

```
cs30xyz@pi-cluster-001:pa4$ ./mygrep --help
cs30xyz@pi-cluster-001:pa4$ ./mygrep --count foo validFilename
cs30xyz@pi-cluster-001:pa4$ ./mygrep -n 'hello world'
cs30xyz@pi-cluster-001:pa4$ ./mygrep -vc '[0-9]+' *.c
```

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```
int processArgs( argInfo_t * info, int argc, char * argv[] );
int lineShouldBePrinted( const char * line, const argInfo_t * info );
char * readLine ( FILE * inputFile );
FILE * openFile( const char * filename );
int search( const argInfo_t * info, const char * filename );
int count( const argInfo_t * info, const char * filename );
```

Assembly routines:

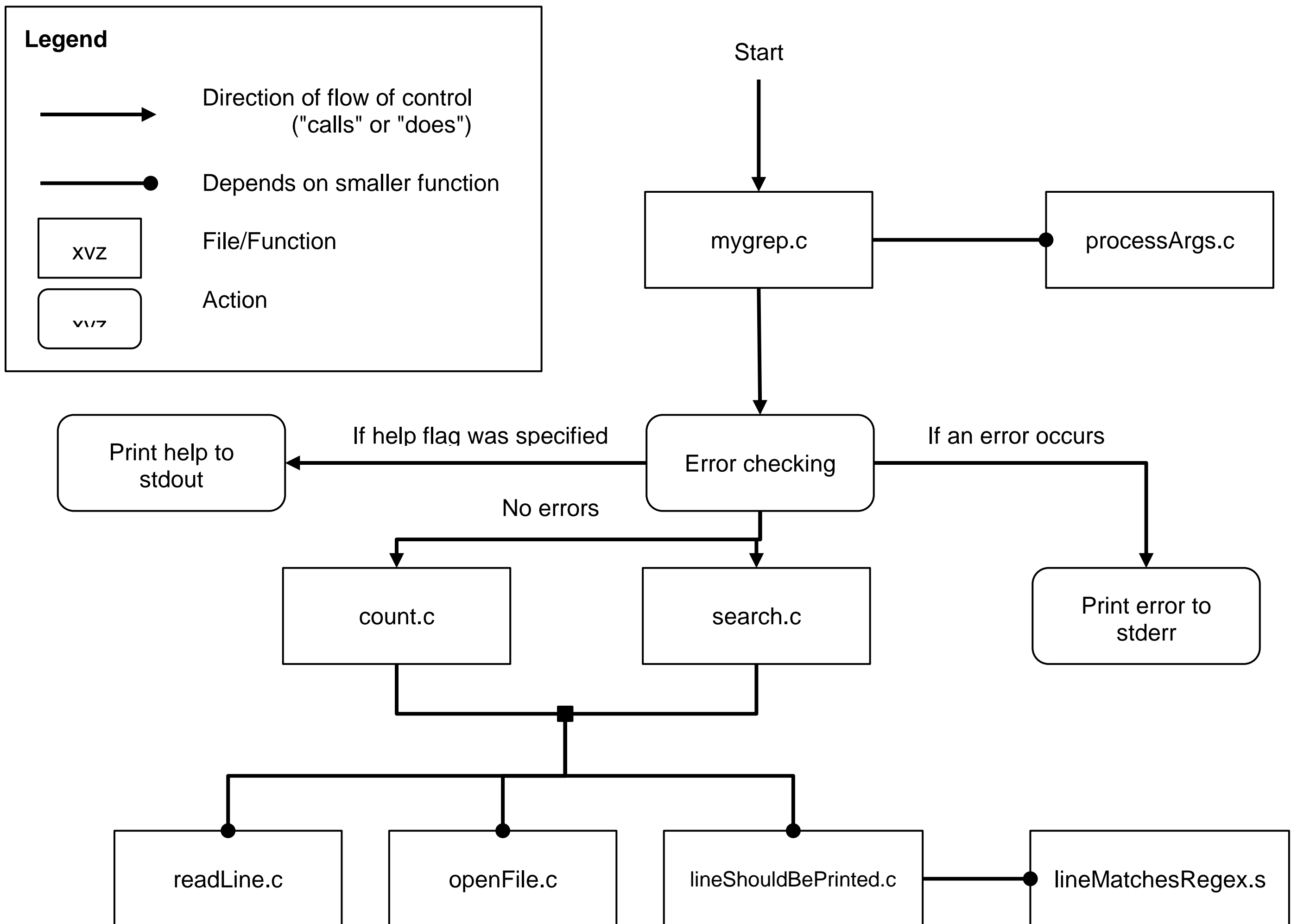
```
int lineMatchesRegex( const regex_t * pattern, const char * line );
```

For the Milestone, you will need to complete:

lineMatchesRegex.s	lineShouldBePrinted.c	processArgs.c	readLine.c
testlineMatchesRegex.c	testlineShouldBePrinted.c	testprocessArgs.c	testreadLine.c

Process Overview:

The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.



The executable should, at a high level, follow the sequence of steps below:

1. Parse command-line arguments and flags.
    a. If an error occurs, print the error to stderr.
    b. If the help flag was specified, print the usage message to stdout.
2. Loop through the specified files:
    a. Call either `count()` or `search()`, depending on what the user-specified flags were.

Milestone Functions to be Written

Listed below are the modules to be written for the milestone.

---

processArgs.c

```
int processArgs( argInfo_t * info, int argc, char * argv[] );
```

Populates `info` with a regex pattern and a bit pattern representing which flags were set in the command line arguments.

Command-line Flags:

Long:	--regexp	--ignore-case	--invert-match	--count	--line-number	--help
Short:	-e	-i	-v	-c	-n	N/A

To populate `info->flags`:

- Set `info->flags` to 0 by default.
- Use `getopt_long()` (`man -s3 getopt_long`) to read each flag. Refer to the usage statement to determine what to pass into `getopt_long()`. Please note that the `--help` flag does NOT have a short option (`-h`) as denoted in the table above.
- For each flag encountered, use bitwise OR to set the corresponding flag bit in `info->flags`.
    - For example, if you encounter `-i`, use a bitwise OR to set the `ARG_I_FLAG` bit.
    - If the `--help` flag is read, set its flag bit and immediately return 0.
- If an unrecognized flag appears, `getopt_long()` will automatically print an error message on your behalf, so you just need to return `ARG_PARSING_ERROR`.
- After parsing all the flags, if there is more than one filename argument passed in, set the `ARG_MULTI_FILE` bit inside `info->flags`. The number of filenames is the number of non-flag arguments. How might you find this number? (`man -s3 getopt_long`)

To set the `info->pattern`:

- You will use `regcomp()` to compile the pattern argument into a `regex_t`, which you will use later to execute regex searches. (Refer to `man -s3 regcomp` for more information, **including its return value**).
- `regcomp(regex_t * preg, const char * regex, int cflags)` takes in 3 parameters:
    - A pointer to where the `regex_t` struct will be stored (look at the struct fields for `argInfo_t`).
    - The regex pattern string, which can come from either:
        - Encountering the `-e` flag. In this case, just use `optarg` for this value.
        - The first non-flag argument in `argv`, if `-e` is never used. Use `argv[optind]` for this value. If `optind` goes out of bounds, then return `ARG_PARSING_ERROR` since no pattern argument exists. After using `optind` for the pattern, **increment** `optind` for the caller so that `optind` is the index of the first filename to read as input.
    - A bit pattern used to determine the type of compilation. To set the bit pattern:
        - Initially set the bit pattern to be equal to `REG_EXTENDED | REG_NOSUB`.
        - If the `-i` flag was provided, use bitwise OR with `REG_ICASE` on top of the existing bit pattern.
- Call `regcomp()` to populate `info->pattern`. If the result of `regcomp()` is non-zero, then return `REGEX_ERROR`.

Reasons for error:

- An unrecognized flag is encountered or `argv` has no pattern argument-- return `ARG_PARSING_ERROR`.
- `regcomp()` has a non-zero return value -- return `REGEX_ERROR`.

Return Value: `ARG_PARSING_ERROR` if an error occurred parsing the flags, `REGEX_ERROR` if the pattern failed to compile, 0 otherwise.

lineMatchesRegex.s

```
int lineMatchesRegex( const regex_t * pattern, const char * line );
```

Checks if a line matches the regex pattern by calling `regexec( pattern, line, 0, 0, 0 )`.

Although `regexec()` takes in five parameters, you cannot use any registers besides r0-r3 to pass arguments -- how do you pass the last argument to `regexec()`? (hint: store extra arguments on the stack).

Note: `regexec()` returns 0 if the pattern matches the line. To follow the convention in C of using non-zero and zero for true and false, respectively, we will return 1 if `regexec()` returns 0, to indicate true (the line **does** match the pattern).

Return Value: 1 if the line matches the regex pattern (i.e. `regexec()` returns 0), otherwise return 0.

---

lineShouldBePrinted.c

```
int lineShouldBePrinted( const char * line, const argInfo_t * info );
```

Checks if `line` should be printed in `search()`.

A line should be printed if it matches the regex pattern found in `info`, unless the -v bit is set in `info->flags`. If the -v bit is set, then lines that **do not** match the regex pattern should be printed. To check if line should be printed, use an assembly helper function you had previously written in this assignment.

Return Value: 1 if line should be printed, else 0.

---

readLine.c

```
char * readLine( FILE * inputFile );
```

Reads a single line from `inputFile` (which you can assume has already been opened), stopping at a newline or when it hits the end-of-file (EOF).

Although we normally load input into an array of size `BUFSIZ`, we run the risk of encountering a line larger in length than `BUFSIZ`, which would be larger than our buffer could hold. To handle this, we will build a string by dynamically-allocating memory so that we can ensure that any lines we read will not be truncated by the size of our buffer. Note that since we are returning the pointer to this string, we will **not** be freeing the dynamically allocated memory inside of `readLine()` (Think: who is responsible for freeing the memory then?).

First, create a `char` buffer of size `LINE_BUFFER_SIZE` on the stack, as well as a dynamically-allocated string using `calloc()`. Initially, your string should be the empty string (how much space would this require?).

In a loop:

- First, try to read a line using `fgets()`.
- Check if you encounter EOF (no bytes read in the buffer).
    - If the dynamically-allocated string is still empty, free the string and return `NULL`.
    - Otherwise, just return the string.

- Check if the newline character was read inside the buffer. If so, replace the newline character with the null terminating character.
- Increase the size of your string with the number of additional characters read using `realloc()`.
- If reallocation fails, print `STR_ERR_MEM_EXCEEDED`, free the string, and return `NULL`.
- Concatenate your string with the characters in your buffer using `strncat()`.
- If a newline was read into the buffer on this iteration, terminate the loop and return the string. Otherwise, continue the loop to read the rest of the line.

Note that contrary to `fgets()`, the string returned by this function will never have a newline character at the end, since we always remove the newline character in the loop above. However, the string should always be null-terminated.

Reasons for error:

- Reached EOF while string is empty -- free string, return `NULL`.
- Memory allocation fails -- print `STR_ERR_MEM_EXCEEDED` to `stderr`, free string, return `NULL`.

Return Value: `NULL` if any error occurred, else a pointer to the null-terminated string containing a line from the file.

Post-Milestone Functions to be Written

Listed below are the modules to be written after the milestone functions are complete.

---

openFile.c

`FILE * openFile( const char * filename );`

Opens `filename` so that it can be read. If the filename is `"-"`, use `stdin` instead.

- If the filename is `"-"`, return `stdin`.
- Check if the file is a regular file using `stat()` (`man -s2 stat`):
  - Create a `struct stat` variable on the stack, set `errno = 0`, and call `stat()` with it.
  - If `errno` is not equal to 0, this means that we can't stat the file. In this case, print `STR_ERR_OPEN_FILE` to `stderr` (use `strerror()` and pass in `errno`) and return `NULL`.
  - Next, check if the file is a directory by using a macro called `S_ISDIR()`, which takes in a `mode_t` struct (use `man -s2 stat` to see which members of `stat` you could use). The macro returns 1 if the file is a directory, in which case, print `STR_ERR_OPEN_FILE` to `stderr` (use `strerror()` and pass in `EISDIR`) and return `NULL`.
- Now that we know the filename refers to a regular file, attempt to open the file using `fopen()`. If this fails (hint: check the value of `errno`), print `STR_ERR_OPEN_FILE` to `stderr` (use `strerror()` and pass in `errno`) and return `NULL`.
- If opening the file succeeds, return a pointer to the opened file.

Reasons for error:

- We can't stat the file -- print `STR_ERR_OPEN_FILE` to stderr, return `NULL`.
- The file is a directory -- print `STR_ERR_OPEN_FILE` to stderr, return `NULL`.
- The file could not be opened -- print `STR_ERR_OPEN_FILE` to stderr, return `NULL`.

Back to Top

Return Value: `NULL` if any error occurred, else a pointer to the opened file.

---

search.c
```
int search( const argInfo_t * info, const char * filename );
```

Opens `filename` and prints out all lines that match the regex pattern found in `info`. This is the core of the `grep` command, as well as your own program. All output in this file will go to **stdout**.

1. First, open the file using `openFile()`. If this fails, then return -1.
2. Loop through the file line-by-line, reading each line with `readLine()`:
   - First, check to see if the line should be printed. If so, **in the following order**:
     - You should use the bit flags set in `info->flags` to check for the following conditions:
       - If there are multiple files, print `STR_FILENAME` with the filename.
       - If the -n flag is set, print `STR_LINENUM` (keep track of which line number you are on).
     - Print out the line itself using `STR_LINE`.
   - Finally, free the line returned from `readLine()` before moving on to the next one.
3. After the loop finishes, close the file and return 0.

Reasons for error:
- The file could not be opened -- return -1.

Return Value: -1 if the file could not be opened, else 0.

---

count.c
```
int count( const argInfo_t * info, const char * filename );
```

Counts the number of lines in `filename` that match the pattern found in `info`. This function will be very similar in structure to `search()`. The key difference is that, instead of printing out each line, you will simply keep a count of the number of lines that match the regex pattern, and then print the total count at the end. Again, output in this file should go to **stdout**.

Things to consider:
- Don't forget to free each line after you are done using it.
- After looping through each line in the file, you need to print the sum total of all the matches in the file:
  - If there are multiple files (use `info->flags` to check), then print `STR_FILENAME`.
  - Print out the count using `STR_COUNT`.
- Make sure to close the file at the end of your function.

Reasons for error:
- The file could not be opened -- return -1

Return Value: -1 if the file could not be opened, else 0.

---

mygrep.c
```
int main( int argc, char * argv[] );
```

This is the main driver of your program. Its main tasks are to parse any flags and arguments that are passed in, and to run the functionality of the `grep` command.

- To parse flags and arguments, first create an `argInfo_t` variable on the stack. Use `processArgs()` with this variable to populate its members.
- If `processArgs()` fails...
  - ...with `ARG_PARSING_ERROR`, then print `STR_SHORT_USAGE` to `stderr` and return `EXIT_FAILURE`.
  - ...with `REGEX_ERROR`, then print `STR_ERR_REGEX` to stderr and return `EXIT_FAILURE`.
- If the help flag was set (check `info.flags`), print `STR_USAGE` to `stdout` and return `EXIT_SUCCESS`.
- If no files were provided as arguments, call either `search()` or `count()` (depending on the flags in `info`), using `STR_STDIN` for the filename.
- Otherwise, loop through the provided files and call either `search()` or `count()` on each filename. Note that if both the -c and -n flags were entered, the -c flag takes precedence since it decides whether `search()` or `count()` will be called.
- At the end of the function, call `regfree()` to free the memory allocated for the `regex_t` pattern field in `info`.

Note: Though search() and count() return different values based on errors, mygrep should just ignore these return values and continue searching/counting the rest of the file arguments -- as the actual grep does.

Reasons for error:
- The arguments could not be processed -- print `STR_SHORT_USAGE` to `stderr`, return `EXIT_FAILURE`.
- Regular expression could not be parsed -- print `STR_ERR_REGEX` to stderr, return `EXIT_FAILURE`.

Return Value: `EXIT_FAILURE` on error, else `EXIT_SUCCESS`.

Unit Testing

You are provided with only one basic unit test file for the Milestone function, `processArgs()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don't do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files **will be collected with the Milestone**, they must be complete before you submit your Milestone.

Unit tests you need to complete:

```
testprocessArgs.c
testlineMatchesRegex.c
testlineShouldBePrinted.c
testreadLine.c
```

To compile:

```
$ make testprocessArgs
```

To run:

```
$ ./testprocessArgs
```

(Replace "testprocessArgs" with the appropriate file names to compile and run the other unit tests)

Customizing LINE_BUFFER_SIZE

For this PA only, we provide extra functionality in the Makefile to let you specify the LINE_BUFFER_SIZE use within readLine(). This may be helpful if you want to test how readLine() handles reading lines longer than LINE_BUFFER_SIZE. Note: a LINE_BUFFER_SIZE of 1 is invalid as fgets() requires at least 2 bytes to read data (one for text from the file and one for the null-terminating character), so you should not be testing with this size.

Also, since Makefiles do not register changes in variables as dependencies, you will need to make clean before you decide to change the LINE_BUFFER_SIZE, otherwise the Makefile will not recompile your executable. This is noted in the compilation examples below.

To compile with a custom size:

```
$ make clean && make LINE_BUFFER_SIZE=<your size here> testreadLine
```

To compile with the default size:

```
$ make clean && make testreadLine
```

When testing your program, we will use the default size, so make sure your program works with the default size.

Optimized Targets

We also supply you an extra set of "optimized" Makefile targets for your tester files. These targets will compile your tester file with all optimizations turned on, similar to how your Milestone functions will be compiled for grading. Sometimes you will get (un)lucky where your program appears to work even when there is something wrong (like incorrect memory layout or modifying a register you shouldn't). Compiling with optimizations will expose some of these hidden problems. Again, this is how the milestone will be tested during grading, so use the optimization targets to try to expose these issues.

However, compiling this way prevents you from using gdb to debug, so make sure to compile the regular way when debugging and try the optimized targets after your tests pass.

To compile with optimizations on:

```
$ make testprocessArgs-opt
```

To run:

```
$ ./testprocessArgs-opt
```

README File

Remember to follow all of the guidelines outlined in the README Guidelines. If you did the extra credit, write a program description for it in the README file as well.

Questions to Answer in your README:

[Vim]

1. How would you replace all occurrences of the word "early" with "often" in Vim?

[Unix]

2. What is the difference between redirecting output with > versus | ?

[C]

3. What are the differences between malloc, calloc, and realloc?

[Academic Integrity]

4. The core values of the Academic Integrity Office are: respect, responsibility, trustworthiness, fairness, honesty, and courage. What are three of your own core values and why are they important to you in excelling with integrity?

Extra Credit

There are 7 points total for extra credit on this assignment.

- Early turnin: **[2 Points]** 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)
- **[5 Points]** Recursive regex search

Getting Started:

Copy over your mygrep.c and processArgs.c to new files called mygrepEC.c and processArgsEC.c

```
$ cp ~/pa4/mygrep.c ~/pa4/mygrepEC.c
$ cp ~/pa4/processArgs.c ~/pa4/processArgsEC.c
```

Compiling:

You can compile the extra credit program using the following command.

```
$ make mygrepEC
```

Sample Executable:

A sample test executable is provided for the EC like the sample executable provided for the regular portion of the assignment. You can run the sample executable using the following command:

```
$ ~/../public/mygrepECtest
```

For extra credit, you will be implementing the recursive flag `-R` for grep. You will implement two new functions called `recursiveGrep()` and `recursiveGrepDir()`, in addition to modifying `mygrepEC.c` and `processArgsEC.c` to support your new flag.

---

recursiveGrepEC.c

```
int recursiveGrep( argInfo_t * info, const char * path );
```

Recursively `greps` on `path`, which can refer to either a regular file or a directory.

If `path` refers to either `stdin` ("-") or a regular file, then just `grep` on the file, calling `search()` or `count()` accordingly.

Otherwise, use `stat()` to find out if `path` is referring to either a file or directory. If you can't stat the file, print `STR_ERR_OPEN_FILE` to `stderr` and return -1.

If `path` refers to a regular file, `grep` on it as usual. If `path` is a directory then recurse on the directory using `recursiveGrepDir()` to grep all its entries and subdirectories. Make sure to use the flags in `info` to determine whether to use `count()` or `search()`.

Reasons for error:

- We can't stat the file -- print `STR_ERR_OPEN_FILE` to `stderr` and return -1

Return Value: -1 if we can't stat the file, else 0.

---

recursiveGrepDirEC.c

```
int recursiveGrepDir( argInfo_t * info, const char * dirpath );
```

Recursively greps on `dirpath`, assuming `dirpath` refers to a directory.

First, set the `ARG_MULTI_FILE` in `info->flags`.

Next, attempt to open `dirpath` as a directory (`man -s3 opendir`). If this fails, then print `STR_ERR_OPEN_FILE` to `stderr` and return -1.

Next, loop across all the entries in the directory. Refer to `man -s3 readdir` for more information on how to do this (hint: it will involve a pointer to a `struct dirent`). Do not recurse on `"."` or `".."` or hidden files (files that start with a `"."`) in the directory. Build the path to the entry using `STR_PATH_JOIN` with a buffer of length `MAX_PATH_LEN` and call `recursiveGrep()` on the path (note: why `recursiveGrep()` and not `recursiveGrepDir()`?).

After you are finished looping across all of the directory's entries, make sure to close the directory and return 0.

Reasons for error:

- The directory could not be opened.

Return Value: -1 if the directory could not be opened, else return 0.

---

processArgsEC.c

```
int processArgs( argInfo_t * info, int argc, char * argv[] );
```

This file will be very similar to `processArgsEC.c`, except with added functionality to handle the `-R` flag.

Reasons for error:

- See description for `processArgs.c`

Return Value: See description for `processArgs.c`

---

myGrepEC.c

```
int main( int argc, char * argv[] );
```

This is the main driver of your recursive grep program. It will be very similar to `mygrep.c` with a few differences:

- Print `STR_EC_USAGE` instead of `STR_USAGE`.
- If no files were provided and the recursive flag `-R` was set, call `recursiveGrep()` on the current directory.
- If files were provided and the recursive flag `-R` was set, call `recursiveGrep()` on each file.

Reasons for error:
- See description for `mygrep.c`

Return Value: See description for `mygrep.c`

Turnin Summary

See the turn-in instructions here. Your file names must match the below *exactly* otherwise our Makefile will not find your files.

Milestone Turnin:
Due: Wednesday night, May 29 @ 11:59 pm

Files required for the Milestone:

lineMatchesRegex.s	lineShouldBePrinted.c	processArgs.c
readLine.c		
testlineMatchesRegex.c	testlineShouldBePrinted.c	testprocessArgs.c
testreadLine.c		

Final Turnin:
Due: Wednesday night, June 5 @ 11:59 pm

Files required for the Final Turn-in:

lineMatchesRegex.s	count.c	pa4.h
	lineShouldBePrinted.c	pa4Strings.h
	mygrep.c	test.h
	openFile.c	Makefile
	processArgs.c	README
	readLine.c	
	search.c	
	testlineMatchesRegex.c	
	testlineShouldBePrinted.c	
	testprocessArgs.c	
	testreadLine.c	

Extra Credit Files:

mygrepEC.c	processArgsEC.c	recursiveGrepDirEC.c
recursiveGrepEC.c		

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.