CSCI 4460/6460 — Large-Scale Programming and Testing
Homework 3 (document version 1.3) — DUE December ~~10~~ 11, 2020
The Regular Expressions Reboot

- This homework is due by 11:59PM EDT on the due date shown above
- This homework is to be completed **individually**; do not share your code with anyone else unless it is for a structured **code review** (which is recommended)
- You **must** continue to use C for this homework assignment, and your code **must** successfully compile via `gcc` with no warning messages when the `-Wall` (i.e., warn all) compiler option is used; we will also use `-Werror`, which will treat all warnings as critical errors
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v18.04.5 LTS; note that the `gcc` compiler is version 7.5.0
- **You can use any standard C libraries that already exist on the Submitty machines (v1.1) except for the regular expression libraries; external libraries are not allowed, so consider writing your own where necessary**

Homework Specifications

In this third homework, you will build on and refactor your first two homework assignments by using C to more fully provide a regular expression interface.

You can use your Homework 1 and/or Homework 2 code and refactor as much as you like—or, also a worthy approach, you can start over from scratch (using what you have learned from the first two assignments).

Required Mindset — Functional Decomposition

Potentially lost in Homework 2 due to its complexity, refocus and use as many “best practice” approaches to designing and implementing your code. If you were frustrated by the previous assignment, try something different as you approach this assignment. Which software development and debugging techniques can truly help you produce high-quality bug-free code?

As suggested previously, continue to make use of **functional decomposition** for this assignment, breaking each task down into smaller functions. Each function should have one clear purpose; if not, refactor.

And once again, **keep careful track** of how many times you compile your code, how many times your code compiles without errors or warnings, and how many times you encounter logic errors (i.e., non-compiler bugs) in your code. You goal is to minimize these latter two measures to the extent possible.

You will again have a cap of only **four** Submitty attempts before you start losing points, so work out your unit tests and test code accordingly.

The Regular Expression Language 2.0

For this assignment, you will (re)implement the general regular expression syntax shown below in support of line-based matching. The details described below are comprehensive and **override any previous requirements** from the first two assignments.

Exact and positional matching. A regular expression (regex) is a string used as a pattern for matching. The regex may contain characters that must match exactly; for example, if the pattern is `printf`, then it will only match exact substrings (i.e., all lines that contain `printf`).

To ensure the beginning or the end of a line is matched, we use `^` and `$` characters, respectively; for example, if the pattern is `^printf`, then it will only match exact substrings that start at the beginning of the line, i.e., the byte at index `[0]` must be `'p'` (or else the line is not a match).

Matching classes of characters. There are a few special characters and backslash sequences that enable classes of characters to be matched. These specific classes of characters are summarized below:

```
.     match any character
\d    match any digit character [0-9]
\D    match any non-digit character
\w    match any letter character [a-zA-Z]
\W    match any non-letter character
\s    match any whitespace character (space or tab '\t')
\S    match any non-whitespace character
```

Matching escaped literals. For characters that have special meaning, we must still be able to match the literal characters. To do so, we can "escape" the character by preceding it with a backslash character. In summary, you must support the following escaped literals:

```
\.    match a single literal '.' character
\\    match a single literal backslash character '\'
\^    match a single literal '^' character
\$    match a single literal '$' character
\?    match a single literal '?' character
\+    match a single literal '+' character
\*    match a single literal '*' character
\{    match a single literal '{' character
\}    match a single literal '}' character
\[    match a single literal '[' character
\]    match a single literal ']' character
\(    match a single literal '(' character
\)    match a single literal ')' character
```

Character set matching. You must support bracket expressions as follows. Any set of verbatim (i.e., untranslated) characters or regex characters (i.e., `.`, `\.`, `\d`, `\D`, etc.) can be combined within square brackets to indicate that one character must be matched from within this group.

Nested groups (i.e., groups within groups) are not allowed.

Also, you must support the optional "not" symbol in bracketed expressions, i.e., use of `^` to indicate any character except those listed. This symbol, if present, will always follow the open bracket `[`.

Repetition through quantifiers. Repetition may be specified by appending the following quantifiers to any individual character, regex symbol, or regex expression:

```
?     match zero or one instances
+     match one or more instances
*     match zero or more instances
{n}   match exactly n instances, where n is a positive integer
{n,}  match at least n instances, where n is a positive integer
{n,m} match at least n and at most m instances, where n is a
      non-negative integer, m is a positive integer, and m > n
```

For `*`, `+`, `{n,}`, and `{n,m}`, use a greedy approach, meaning that you should match as many characters as possible.

These repetition symbols and expressions may be applied to bracket expressions by including them directly after the closing bracket `]`. Further, these repetition symbols and expressions may also be applied to captured groups (see below).

Capturing groups. You can capture and "remember" groups within a match using parentheses to identify each group. Groups are indexed starting at one, and parentheses cannot be nested. As an example, consider the pattern `(\w+)\s(\d{3})` as applied to the following line:

```
blAH bLAh ABCD 0123456789 BLah BlaH bLaH blah
```

Here, based on the parentheses, the line matches the pattern `\w+\s\d{3}`, with group `[1]` set to `ABCD` and group `[2]` set to `012`.

As an example of repetition, the pattern `(\w{2})+` applied to the line above would match the line and identify group `[1]` as `bl` and group `[2]` as `AH`.

Example Regular Expressions

Given the previous two pages of regular expression language specifications, example regular expressions to support (and test) include:

`printf`	match "printf" exactly within a line
`^printf`	match "printf" exactly at the beginning of a line
`printf$`	match "printf" exactly at the end of a line
`colou?r`	match either "color" or "colour"
`\w+`	match a word consisting of one or more alpha characters
`<\w+>`	match an XML/XHTML tag, such as "<title>" or "<body>"
`<(\w+)>`	match an XML/XHTML tag, such as "<title>" or "<body>", identifying the tag as match [1]
`<(\w+)[^>]*>`	match an XML/XHTML tag potentially with attributes etc. (on one line), such as "<title..." or "<body...>", identifying the tag as match [1]
`\d\d\d-\d\d\d\d`	match a phone number, such as "276-2819"
`\d{3}-\d{4}`	match a phone number, such as "276-2819"
`\(\d{3}\)\s\d{3}-\d{4}`	match a phone number, such as "(518) 276-2819"
`(\$\d+\.\d{2})`	match a price in dollars, such as "$0.99" or "$100.00", identifying the price as match [1]
`if\s*\(.+\)`	match an if statement (to some degree) in C/C++/Java
`[aeiou]`	match a single vowel character (i.e., pick one)
`[^aeiou]`	match a single character that is not a vowel
`part [123]`	match "part 1" or "part 2" or "part 3"
`part [12][ab]`	match "part 1a" or "part 1b" or "part 2a" or "part 2b"
`part [123][abc]?`	match "part 1" or "part 2" or "part 3" or any of these three optionally followed by "a" or "b" or "c"
`\w+\d*@rpi\.edu`	match an RPI email address by matching one or more alpha characters, followed by zero or more digit characters, then "@rpi.edu"
`[\w\d]`	match a single character that is either an alpha character or a digit
`[^\w\d]`	match a single character that is not an alpha character or a digit
`[\w\d]+`	match a word consisting of one or more alpha characters and/or digits
`\w+\s\d{3}`	match a word consisting of one or more alpha characters, followed by a whitespace character, followed by exactly three digit characters
`(\w+)\s(\d{3})`	match a word consisting of one or more alpha characters, followed by a whitespace character, followed by exactly three digit characters, capturing the word as match [1] and the three digit characters as match [2]

Modularizing your Regex Program

To support reusability and modularity, you must provide a uniform interface in C for your regular expression code. The function prototypes and requirements are shown below.

```
/*  regex_match() applies the given regex to each line of the file specified
 *   by filename; all matching lines are stored in a dynamically allocated
 *    array called matches
 *
 *  the matches array is dynamically allocated in the regex_match() function;
 *   therefore, it is up to the calling process to free the allocated memory
 *
 *  trim_to_match is a flag that, when set, indicates that each line in the
 *   given file that matches the given regex must be trimmed to show only
 *    the matching substring
 *
 *  the groups arrays is dynamically allocated in the regex_match() function
 *   only if necessary; it contains each group identified by and captured
 *    within parentheses, with the first match at index [1], the second
 *     match at [2], etc. (therefore, index [0] is always a NULL pointer)
 *
 *  the captured_groups integer corresponds to the groups array above;
 *   this value specifies how many groups were identified (e.g., 2)
 *
 *  (v1.2) if multiple lines match, both groups and captured_groups
 *         correspond to the last line matched
 *
 *  regex_match() returns the number of lines that matched (zero or more)
 *   or -1 if an error occurred
 */
int regex_match( const char * filename, const char * regex,
                 char *** matches, int trim_to_match,
                 char *** groups, int * captured_groups );
```

As an example, if the `trim_to_match` parameter is set, the following regular expressions will effectively extract XML/XHTML tags, phone numbers, email addresses, prices, etc.

```
<\w+>              match an XML/XHTML tag, such as "<title>" or "<body>"
<(\w+)>            match an XML/XHTML tag, such as "<title>" or "<body>",
                    identifying the tag as captured_group[1]
<(\w+)[^>]*>       match an XML/XHTML tag potentially with attributes etc.
                    (on one line), such as "<title..." or "<body...>",
                     identifying the tag as captured_group[1]
\w+\d*@rpi\.edu    match an RPI email address by matching one or more alpha
                    characters, followed by zero or more digits, then the
                     string literal "@rpi.edu"
```

```
\d\d\d-\d\d\d\d     match a phone number, such as "276-2819"
(\$\d+\.\d{2})      match a price in dollars, such as "$0.99" or "$100.00",
                      identifying the price as captured_group[1]
```

Note that if multiple matches are possible for a given line, your function should match the leftmost matching substring.

Assumptions

For this implementation, you can make the following assumptions:

1. Assume that the maximum line length for any input file is `256` characters.
2. Assume that the regex is correct in terms of its syntax.

Testing your Code and What to Submit

On Submitty, your `regex_match()` function will be tested via a separate hidden `main()` function. To ensure that your function can be automatically tested on Submitty, do **not** include a `main()` function of your own.

A suggestion here is to enclose your `main()` function within an `#ifdef`, as shown below:

```
  #ifdef USE_MY_MAIN_FUNCTION
  int main( ... )
  {
    /* your main() code here */
    ...
  }
  #endif
```

Given this, you can compile your code locally as follows:

```
bash$ gcc -Wall -Werror -D USE_MY_MAIN_FUNCTION *.c -lm
```

Implementation Details

For your implementation, consider using `fopen()`, `fscanf()`, `fgets()`, `fclose()`, `isalpha()`, `isdigit()`, `isspace()`, and other such string and character functions. Be sure to check out the details of each function by reviewing the corresponding `man` pages.

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution.

Error messages must be one line only and use the following format:

```
  ERROR: <error-text-here>
```

Submission Instructions

To submit your assignment (and perform final testing of your code), please use Submitty. Remember that you will only have four penalty-free submissions before points are deducted. Therefore, be sure to thoroughly review and test your code.

Be careful to submit your code **without** a `main()` function, since all testing will be done via a separate `main()` function on Submitty.

This assignment will be available on Submitty a few days before the due date. To make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!\n" );
    printf( "aaaaaaaaaaaaagggggggghhhh!\n" );
    printf( "when will this all be over???\n" );
#endif
```

And to compile this code in “debug” mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw3.c
```