CSCI 4460/6460 — Large-Scale Programming and Testing
Homework 2 (document version 1.3) — DUE November ~~12~~ ~~16~~ 18, 2020
Text Analysis (and more Regular Expressions) in C

- 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
- 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

Homework Specifications

In this second homework, you will build on your first homework assignment by using C to analyze text and calculate a variety of text-based statistics. In brief, your program should apply regular expression logic extended from the first homework assignment to a specified list of input files. For each file, you will use regular expressions to identify all words, bigrams, and trigrams.

You can use your Homework 1 code and refactor as much of it as you like—or you can start over from scratch (using what you learned from trying to complete Homework 1). Regardless, be sure that all required Homework 1 functionality is still correctly implemented. Therefore, do regression testing to be sure that all tests from Homework 1 still pass.

Required Mindset — Functional Decomposition

As with Homework 1, for this assignment, use as many "best practice" approaches to designing and implementing your code. Try to further identify which new techniques help you produce high-quality bug-free code.

Also, 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.

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.

As with Homework 1, you will have a cap of only **four** Submitty attempts before you start losing points. And a portion of your grade will be based on the level of good commenting, variable naming, program design, program structure, etc. that you achieve.

Extending the Regular Expression Language

Two key updates to the regular expression language from the first homework assignment are required. (You may also extend the regular expression language further if you'd like.)

First, since `.` will match any character, add support for `\.` to match a single period character. See below for an example.

Second, you must support bracket expressions as follows. Any set of verbatim (i.e., untranslated) characters or regex characters (i.e., `.`, `\d`, `\w`, `\s`, etc.) can be combined within square brackets to indicate that one character must be matched in this group.

A repetition symbol (i.e., `*`, `+`, or `?`) may also be applied to the entire group by including it directly after the closing bracket `]`. Note that nested groups (i.e., groups within groups) is not allowed.

You must also 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.

Given the above, some example regular expressions to support (and test) are:

```
[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 digits, then "@rpi.edu"
                    (this time, the "\." matches the . character)
[\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
```

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
 *
 *  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 );
```

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, etc. from a given input.

```
 <\w+>               match an XML/XHTML tag, such as "<title>" or "<body>"
 \w+\d*@rpi\.edu     match an RPI email address by matching one or more alpha
                      characters, followed by zero or more digits, then "@rpi.edu"
 \d\d\d-\d\d\d\d     match a phone number, such as "276-2819"
```

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

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, enclose your `main()` function within an `#ifndef`, as shown below:

```
#ifndef USE_SUBMITTY_MAIN
int main( ... )
{
  /* your main() code here */
  ...
}
#endif
```

Using Regular Expressions to Identify Words

For the next part of this assignment, using your `regex_match()` function **(v1.2)** and/or other reusable functions that you create, your aim is to identify words, bigrams, and trigrams from a series of input files.

Assume you are processing HTML documents (though your program should also work for general text documents). Similar to the `wordfreq.php` example covered in the September 22 notes, perform the following preprocessing steps *before* you start identifying words, bigrams, and trigrams:

1. Ignore any special HTML characters, where the list of special characters for this assignment is ` `, `&quot;`, `&`, `<`, and `>`.
2. Ignore any scripts, i.e., all text within `<script>` tags. Note that `<script>` tag content could span multiple lines.
3. Ignore all HTML tags, including closing tags (e.g., `</title>`, `</h2>`, etc.) and abbreviated tags (e.g., `<br/>`, `<img ... />`, etc.). Though uncommon, tags could span multiple lines, e.g.,

```
<img src="blahblahblah.png" width=92 height=108
   alt="Blah Blah Blah" style="border-width: 0"; />
```

4. Convert everything to lowercase.

After preprocessing is complete, identify **valid words** as words at least two characters in length consisting of alpha characters (i.e., as identified by the `isalpha()` function of the `ctype.h` library).

Also, to capture contractions (e.g., "don't"), allow single quotes to be part of a word, though words must always begin and end with a letter. Further, to simplify implementation, only one single quote can be in a word; therefore, a second single quote would simply act as a word delimiter.

Finally, in general, if a character is not part of a valid word, treat it as a word delimiter.

As a example, consider the input below.[1]

```
  <script>window.jQuery ||
     document.write("<script src='/sites/all/modules/jquery_update/replace/jque
ry/1.10/jquery.min.js'>\x3C/script>")</script>
 <p>his  favorite cereal is Cocoa Puffs (with Trix as a close second).ain't that
       nice</p>
 <img src="blahblahlblah.png" />'isn't' is a word, but 'ain't' ain''t a freakin' word
```

Valid words (left-to-right) are:

```
his favorite cereal is cocoa puffs with trix as close second ain't that nice
isn't is word but ain't ain freakin word
```

[1] **(v1.3)** No typos here—expect input to be "messy" as HTML rules are loosely enforced across Web browsers.

Identifying Bigrams and Trigrams

In addition to identifying individual words, you must also identify bigrams (e.g., United States) and trigrams (e.g., Rensselaer Polytechnic Institute). Note that bigrams and trigrams may span multiple lines. And unlike individual words, the most frequently occurring bigrams and trigrams are deemed to be of interest. Therefore, you will need to identify the top 20 bigrams and the top 12 trigrams from the full set of input files (i.e., top 20 and top 12 across all files).

As you identify bigrams and trigrams, use a list of "stop words" that are skipped to help identify only the "interesting" bigrams and trigrams. Note that when you encounter a stop word, it serves as a delimiter to any bigram or trigram; in other words, bigrams and trigrams must not span a stop word. Use the top 50 words from the AP89 dataset shown in the "Architecting a Search Engine" lecture notes from September 15.

Continuing the example from the previous page, again consider the input below.

```
  <script>window.jQuery ||
     document.write("<script src='/sites/all/modules/jquery_update/replace/jque
ry/1.10/jquery.min.js'>\x3C/script>")</script>
 <p>his  favorite cereal is Cocoa Puffs (with Trix as a close second).ain't that
       nice</p>
 <img src="blahblahlblah.png" />'isn't' is a word, but 'ain't' ain''t a freakin' word
```

Valid words (left-to-right) are still:

```
his favorite cereal is cocoa puffs with trix as close second ain't that nice
isn't is word but ain't ain freakin word
```

Valid bigrams (left-to-right) are:

```
favorite cereal
cocoa puffs
close second
second ain't
nice isn't
ain't ain
ain freakin
freakin word
```

Valid trigrams (left-to-right) are:

```
close second ain't
ain't ain freakin
ain freakin word
```

Calculating Text Statistics

Using whatever data structure(s) you like, keep track of all words that you read in from each file (i.e., document). You will need to keep track of the number of individual word occurrences, the number of bigram occurrences, the number of trigram occurrences, the total number of documents, and the total number of unique words, unique bigrams, and unique trigrams.

After you have processed all of the specified documents, output the most frequently occurring words, bigrams, and trigrams, as shown in the example below. Sort your output by most frequently occurring to least frequently occurring. And for any ties, sort these in alphabetical order. As an example, if you run your program with `book-1984.txt`, your output is as follows:

```
Total number of documents: 1
Total number of words: 101352
Total number of unique words: 8900
Total number of interesting bigrams: 31676
Total number of unique interesting bigrams: 24640
Total number of interesting trigrams: 16161
Total number of unique interesting trigrams: 15501

Top 50 words:
6522 the
3493 of
2445 and
2348 to
2316 was
1964 he
...

Top 20 interesting bigrams:
72 big brother
53 do you
48 you could
47 if you
40 thought police
36 old man
...

Top 12 interesting trigrams:
13 two minutes hate
12 do you know
11 three super states
9 do you remember
9 made no difference
8 anti sex league
...
```

Command-Line Arguments

Each command-line argument specifies the name of an input file to be read and processed. At least one command-line argument must be present.

Assumptions

For this assignment, you can make the following assumptions:

1. Assume that each specified command-line argument (i.e., each filename) is no more than `128` characters.
2. Assume that the maximum line length for any input file is `1024` characters.

Implementation Details

As with the first homework assignment, for your implementation, consider using `fopen()`, `fscanf()`, `fgets()`, `fclose()`, `isalpha()`, `isdigit()`, `isspace()`, and other such string and character functions. For the `regex_match()` function, use `calloc()`.

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>
```

For errors involving command-line arguments, display the following two lines to `stderr` exactly as shown below:

```
ERROR: Invalid arguments
USAGE: a.out <input-file1> [ <input-file2> ... ]
```

Submission Instructions

To submit your assignment (and also 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.

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( "aaaaaaaaaaaaaggggggggghhhh!\n" );
#endif
```

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

```
bash$ gcc -Wall -Werror -D DEBUG_MODE hw2.c
```

Second, output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `setvbuf()` as follows as your first line in `main()`:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure good results on Submitty, this is a good technique to use.