COMP 348: ASSIGNMENT 3

Important Info:

1. *Feel free to talk to other students about the assignment and exchange ideas. That's not a problem. However, when you write the code, you must do this yourself. Source code cannot be shared under any circumstance. This is considered to be plagiarism.*

2. *Assignments must be submitted on time in order to received full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied, as appropriate. You **MUST** verify the contents of your assignment AFTER you submit. You will be graded on the version submitted at the deadline – no other version will be accepted at a later date.*

3. *The graders will be using a standard 1.10.x distribution of Clojure. You cannot use any Clojure libraries and/or components that are not found in the standard distribution. The graders will not have these libraries on their systems and, consequently, your programs will not run properly.*

4. *Do not use build automation tools like Leiningen. These tools are very useful for building big projects but will only complicate matters for a simple assignment like this. You should be able to run your code from the command line simply by invoking the main `clojure` executable (e.g., `clojure -M menu.clj`)*

DESCRIPTION: It's time to try a little functional programming. In this assignment, you will have a chance to gain experience with the Clojure programming language. The program itself *should* be relatively small. However, you will have to think a little differently about the logic in order to actually produce a working solution. In the process, you should get some sense of the syntax and style of a functional language.

In terms of your task, it can be described as follows. You will be developing an application that will allow users to compress and decompress simple text files. A small menu will provide users with a handful of options for manipulating and displaying input. The menu will be particularly convenient, as it will allow both students and graders to evaluate the accuracy of your application.

***NOTE**: To make your life a little less unpleasant, and to ease the workload at the end of the semester, we will be giving you the code for the menu. Essentially, this will be a working shell program that will give you a place to get started. Moreover, it will allow you to see how a small group of functions works together in Clojure. Your job, therefore, will be to add logic to the menu so that the various display and compression functions are invoked, as necessary.*

The first option in the menu will be "Display list of files". As the name implies, selection of this option will display a list of one or more file names. In practice, this list is just the files in the current folder (i.e., the same folder from which you are running your Clojure program.) The image below shows the basic menu and the result obtained when choosing Option1.

```
*** Compression Menu ***
------------------

1. Display list of files
2. Display file contents
3. Compress a file
4. Uncompress a file
5. Exit

Enter an option? 1

File List:
* ./compress.clj
* ./frequency.txt
* ./menu.clj
* ./t1.txt
* ./t2.txt
* ./t3.txt
* ./t4.txt
```

In this case, the list consists of a pair of Clojure source files (your program code), a file called `frequency.txt` (discussed below), and a group of `tn.txt` files (the sample/test files that you or the grader will create). It's possible that there may also be some OS-specific files (e.g., ..DS_Store); that's fine too.

Note that while capturing a directory listing can be somewhat tedious in some languages, it is quite trivial in Clojure and makes use of the `file-seq` function. You can find documentation and samples on the main Clojure web site. You should just need a line or two of code.

We now move on to Option 2 – Display file contents. Here, we are simply going to display the content of the input file that we are currently processing. In the current case, this is one of the t*n*.txt files (note that these files could have any name).

To display a file, we simply prompt a user for the file name, then read the text content and print it to the screen. Again, this is a trivial thing to do and utilizes Clojure's `slurp` function. Use of this function is documented and illustrated on the Clojure web site and should again require just a line or two of code. The image below illustrates the use of Option 2 for the `t1.txt file` shown in the previous File Listing.

```
*** Compression Menu ***
------------------

1. Display list of files
2. Display file contents
3. Compress a file
4. Uncompress a file
5. Exit

Enter an option? 2

Please enter a file name => t1.txt

 The first man is from the last group of people
```

It should be obvious that basic error checking must also be provided. Specifically, we must ensure that both the file names. An example of an "invalid file" message is listed at the right.

```
*** Compression Menu ***
------------------

1. Display list of files
2. Display file contents
3. Compress a file
4. Uncompress a file
5. Exit

Enter an option? 2

Please enter a file name => t1.txg
Oops: specified file does not exist
```

So now we move on to Option 3 and Option 4, which are of course the main focus of the assignment. We begin with Option 3 – Compress a file. In fact, we will be starting with t1.txt, which contains the following text:

```
The first man is from the last group
of people
```

So we need to compress this text file in some way. What sort of compression algorithm are we going to use? A very simple one! Specifically, we are going to convert the original text file into a new text file in which the words in the original file are converted into numbers (again, the output file is a text file, not a binary file).

So let's select Option 3 and provide the name `t1.txt` when prompted for a file name. If all goes well, the compression will be performed, a new compressed file will be created, and the menu will be re-displayed. (Note that with this assignment the screen is not cleared each time the menu is displayed; the menu will simply keep scrolling down the console.)

If we now select Option 1 and display the File List again, we will see that a new file has appeared. In this case, it will be called `t1.txt.ct`. In fact, this is the compressed text (ct) file that was created by our program when we selected Option 3. Because the compressed files are text-based, we can again use Option 2 to display the contents. So if we use Option 2 and provide the name `t1.txt.ct`, we should see something like the following displayed:

```
0 41 374 6 15 0 197 178 1 86
```

This is the compressed version of our original text. So where did these numbers come from? In short, the numbers refer to the frequency of use in the English language. In this case, the word "the" is the most frequently used word in the English language. As a result, it is assigned the numeric label "0". The word "of" is the second most used word and is represented in our compressed file as "1".

Of course, you have to get these frequencies from somewhere. So with the assignment, we have provided a text file called `frequency.txt`. This is a simple text file that contains the 10,000 most common words in English, ordered by their frequency. You will have to use the contents of this file to determine the frequency for a given word. **NOTE:** The file actually contains phrases as well as words (e.g., "the other"). So, in practice, it is only the first occurrence of the word that is relevant; all subsequent occurrences/duplicates are associated with phrases and should thus be ignored.

So now we can select Option 4 to uncompress a file. Once we provide the name of a previously compressed file (e.g., t1.txt.ct), the program will decompress the file and (hopefully) display the

original text directly to the screen (note: we do not create a new file, we just display the compressed contents in an uncompressed form). At this point, both you and the graders will be able to verify that the compression has worked.

So that's the basic idea – compress and decompress using word frequency. Of course, we need to do a little more to make the program interesting. Let's try another text file, say `t2.txt`, with content like the following:

```
The pink elephant is absolutely groovy
```

This time, when we compress the file and use Option 2 to view `t2.txt.ct`, we would see the following:

```
0 6167 elephant 6 5852 groovy
```

Here we notice an obvious difference. Specifically, the words "elephant" and "groovy" are not among the most common 10,000 English words. As a result, we cannot use a numeric frequency to represent them. Instead, we must record the word directly in the compressed file. This also means that when we select Option 4 to decompress, our application must be able to handle this additional complexity and reproduce the original text again.

Okay, so let's try one more file. We'll say that we have a file called `t3.txt` that has the following content:

```
The experienced man, named Oz, representing the 416 area code - and
his (principal) assistant - are not in the suggested @list
[production, development]. Is that actually the correct information?
```

It should be clear that this is a more complex piece of text. Before discussing the details, let's look at the contents of the `t3.txt.ct` file that would represent the compressed version of this text:

```
0 1686 374 , 402 Oz , 2280 0 @416@ 148 617 - 2 18 ( 2156 ) 4103 - 14
23 4 0 957 @ 734 [ 250 , 230 ] . 6 8 759 0 1524 295 ?
```

Let's point out a number of issues:

- There is a lot of *punctuation* (commas, periods, parentheses, etc.). You can see in the compressed file that the punctuation symbols are included in their original form. This is important since punctuation is required in order to properly read English sentences.
- To do this, the punctuation has been separated from the text <u>before</u> determining frequencies. So, for example, you cannot search for "man," in the frequency.txt file. Instead, man is represented as 374 in the compressed file, and the "," comes next.
- The phrase "the 416 area code" contains an integer that must be represented properly in the compressed file. Clearly it cannot be stored simply as 416 since the decompression function(s) would treat this as a frequency. Instead, we record it as @416@ so that we will know that it must be treated differently.

So this complicates the compression/decompression routines. However, there is one more thing to consider. When we decompress the file, we want to get the original text back (i.e., the exact expression shown above). But if we simply decompress the text of `t3.txt.ct` directly, we will likely get something like:

```
the experienced man , named Oz , representing …
```

In other words, the punctuation and presentation style will be wrong. This is because no formatting information can (or should) be stored in the compressed file. It just contains the raw information. So your decompression functions must actually apply basic rules of English text formatting in order to get the data back to a readable form. These "rules" include:

- The first letter in the first word of every sentence must be capitalized.
- End of phrase/sentence punctuation (e.g., [, . ? !] should have a space after but not before. So we would have "the experienced man, named…" or "…correct information?"
- Opening parentheses - either (or [- should have a space before but not after. For example, "his (principal) assistant". Conversely, closing parentheses should have a space after.
- A dash should have space before and after, as in "code – and".
- The @ sign should have a space before but not after, as in "suggested @list". The same would be true of a dollar sign $
- Finally, note that any other punctuation formatting is not relevant and will not be included in the graders' sample. So, for example, a period (.) will only be used to end a sentence. It will not be used within a real number like 45.678

So that's the full description of the assignment's requirements. To receive full value, you must be able to compress and decompress files like the ones above. Note that the actual samples used for grading will be slightly different, but will be of the same basic form.

THINGS TO KEEP IN MIND: The logic itself is not especially difficult to either describe or understand. In fact, if this assignment was written in Java, it would likely not take too long. But it will be written in Clojure and most of you will not be familiar with the style of functional programming. So the following list highlights some of the things you should keep in mind:

- Do NOT try to code the full assignment from the start. You will likely create a mess that simply doesn't work. Start simple and slowly add additional features. So play with the menu first and provide the functions to display files and file content. Then do a very simple compression and decompression on a trivial file. Only then should you start adding additional complexity to the process.
- Keep your functions small. Do not try to create functions that do 4 or 5 different things. This works for imperative programming but is awkward for functional programs. Instead, use *function composition* to produce a sequence of operations. This might be something like `(add_foo_logic (add_baz_logic (add_bar_logic input_data)))` This will essentially act as a pipeline that processes the input_data with `add_bar_logic`, then pipes this intermediate output to `add_baz_logic` and then `add_foo_logic`.

- Use `let` expressions to prepare data/input in some way and assign this logic to a label/binding. You can then use the bound name in the subsequent expression. This will make the code easier to read and simpler to implement. You can see examples of `let` in the `menu.clj` template
- Use the *apply-to-all* functions to encapsulate complex logic in a simple expression. This includes `map, reduce, filter, apply`, etc. These functions are very powerful, relatively simple, and completely avoid the requirement to explicitly implement recursive functions.
- If you need to pass a common data structure(s) to various functions, you might want to pre-build the data structure and associate it with a label. For example, something like `(def myFoo (build_data_structure))`.

Clojure has all of the basic functions you will need to provide the logic for the assignment. But just to help point you in the right direction, you might want to make use some of the following functions:

- `str` – concatenate one or more arguments into a single string (works for text, ints, etc)
- `read-string` – can be used to read a string into an int representation. Note that you should not use read-string with punctuation symbols (this will likely produce strange EOF errors)
- `map/reduce/filter` – built-in recursive processing
- `apply` – apply a function to arguments or elements of a sequence
- `re-find` – returns the next regex match
- `slurp` – read the text of a file into a string
- `spit` – write text to a file
- `zipmap` – directly construct a map/dictionary from the elements of a sequence
- `into` – populates a new sequence with elements from one or more other sequences
- `range` – generate a sequence of integers

You will want also want to make use of the functions in the Clojure `string` module. This includes functions like `split, trim, replace`, etc.

Finally, Clojure offloads some of its functionality to the underlying JVM. So in some cases, you may need to use functions from Java classes such as `Char` and `Integer`. Calling these functions typically uses a dot notation. You can see many simple examples online.

USING REGEX EXPRESSIONS FOR TEXT MATCHING: You will need to use regular expressions to manipulate text. Regexes are standard elements of virtually any modern language and the basic syntax is the same across all of them. You may already been exposed to regular expressions since they are a very fundamental programming concept. But just as a recap, a simple example using the Clojure `string/replace` function might be

```
(clojure.string/replace "The color is red." #"[aeiou]"  #(str %1 %1))
```

Here, the regex - `#"[aeiou]"` – uses [ ] parentheses to represent a group of possible letter matches. The second expression is the *replacement* string. In this case, %1 refers to the letter that was matched. So the `str` function creates a new substring that essentially replaces each matched letter with two copies of the matched letter(s). In this case, we would get

```
"Thee cooloor iis reed."
```

It is also possible to match *subgroups* of characters. We use round parentheses to indicate a subgroup. So we might have something like

```
(clojure.string/replace "fabulous fodder foo food" #"f(o+)(\S+)" "$2$1")
```

In this case, we match two character subgroups in the string. The `$2$1` in the replacement expression represents the second and first subgroup matches ($0 is the entire match). So this replacement would produce

```
"fabulous ddero oo doo"
```

Note as well that regex expressions typically use pre-defined character classes such as `\d` (i.e., a digit). This can make the expression much easier to write.

In addition, we must generally escape *special characters* if we need to match them in the source text. Most punctuation symbols have special meaning in a regex, so if we want to match against a period (.), for example, we would need to write this as `\.`

Finally, note that Clojure essentially uses Java's regex engine. So if you want a reference for Clojure's regex model, you can start by looking at the Java documentation at:

https://docs.oracle.com/javase/tutorial/essential/regex/index.html

EVALUATION: For testing purposes, the markers will simply run your code on several simple text files to see that you are doing appropriate compression and decompression. Partial grades will be given if at least some of the functionality is working. In practice, it should be fairly easy to compress and decompress something like the t1.txt sample listed above. It gets considerably harder to perfectly decompress t3.txt.

DELIVERABLES: The code will be organized into two source files, one containing the menu (and basic error checking that you will add), the other containing the logic for compression and decompression. The first file is of course called `menu.clj` and the second will be `compress.clj`.

In order for `menu` to be able to access the functions in `compress.clj`, we must define a *namespace* for `compress.clj`. If you have any library *aliases*, they can also be added here. So something like the following would be fine:

```
(ns compress
  (:require [clojure.string :as string]))
```

You can also provide a namespace for `menu`. The name doesn't actually matter, however, since `menu` will run directly from the command line.

Important: The Clojure interpreter will NOT automatically look for modules in the current folder. So we must specify this explicitly. There are a number of ways to do this (including a `deps.edn` file), but the Clojure distribution on docker is a little more limited. Plus, we want to keep things very simple.

So we are simply going to set the `CLASSPATH` environment variable so that it includes the current folder. From the command line in docker you can type:

```
export CLASSPATH=./
```

This would have to be set each time you logged in. If you want this to be set permanently – and you are working on the docker Linux distribution - you can simply add the `export` line to the `.bashrc` file in the `/root` folder.

If the Classpath has been set properly, you can now run your application as:

```
clojure -M menu.clj
```

Once you are ready to submit, place the `menu.clj` fand `compress.clj` files into a zip file. Do not include sample files since the graders will provide these. The name of the zip file will consist of "a3" + last name + first name + student ID + ".zip", using the underscore character "_" as the separator. For example, if your name is John Smith and your ID is "123456", then your zip file would be combined into a file called a3_Smith_John_123456.zip". The final zip file will be submitted through the course web site on Moodle. You simply upload the file using the link on the assignment web page.

Good Luck