COMP 348: ASSIGNMENT 3

Note that 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.

DESCRIPTION: It's time to try a little functional programming. In this case, your job will be to develop a very simple Sales Order application with Clojure. REALLY simple. In fact, all it will really do is load data from a series of three disk files. This data will then form your Sales database. Each table will have a "schema" that indicates the fields inside. So your DB will look like this:

cust.txt: This is the data for the customer table. The schema is

<custID, name, address, phoneNumber>

An example of the cust.txt disk file might be:

```
1|John Smith|123 Here Street|456-4567
2|Sue Jones|43 Rose Court Street|345-7867
3|Fan Yuhong|165 Happy Lane|345-4533
```

Note that no error checking is required for any of the data files. You can assume that they have been created properly and all fields are present. Each field is separated by a "|" and contains a non-empty string.

prod.txt: This is the data for the product table. The schema is

<prodID, itemDescription, unitCost>

An example of the prod.txt disk file might be:

```
1|shoes|14.96
2|milk|1.98
3|jam|2.99
4|gum|1.25
5|eggs|2.98
6|jacket|42.99
```

sales.txt: This is the data for the main sales table. The schema is

<salesID, custID, prodID, itemCount>

An example of the sales.txt disk file might be:

```
1|1|1|3
2|2|2|3
3|2|1|1
4|3|3|4
```

The first record (salesID 1), for example, indicates that John Smith (customer 1) bought 3 pairs of shoes (product 1). Again, you can assume that all of the values in the file (e.g., custID, prodID) are valid.

So now you have to do something with your data. You will provide the following menu to allow the user to perform actions on the data:

```
*** Sales Menu ***
------------------

1. Display Customer Table
2. Display Product Table
3. Display Sales Table
4. Total Sales for Customer
5. Total Count for Product
6. Exit

Enter an option?
```

The options will work as follows

1. You will display the contents of the Customer table. The output should be similar (not necessarily identical) to

   1: ["John Smith" "123 Here Street" "456-4567"]
   2: ["Sue Jones" "43 Rose Court Street" "345-7867"]
   3: ["Fan Yuhong" "165 Happy Lane" "345-4533"]

2. Same thing for the prod table.

3. The sales table is a little different. ID values aren't very useful for viewing purposes, so the custID should be replaced by the customer name and the prodID by the product description, as follows:

   1: ["John Smith" "shoes" "3"]
   2: ["Sue Jones" "milk" "3"]
   3: ["Sue Jones" "shoes" "1"]
   4: ["Fan Yuhong" "jam" "4"]

4. For option 4, you will prompt the user for a customer name. You will then determine the total value of the purchases for this customer. So for Sue Jones you would display a result like:

   Sue Jones: $20.90

   This represents 1 pair of shoes and 3 cartons of milk. Note: If the customer is not valid, you can either indicate this with a message or return $0.00 for the result.

5. Here, we do the same thing, except we are calculating the sales count for a given product. So, for shoes, we might have:

   Shoes: 4

   This represents three pairs for John Smith and one for Sue Jones. Again, if the product is not found, you can either generate a message or just return 0.

6. Finally, if the Exit option is entered the program will terminate with a "Good Bye" message. Otherwise, the menu will be displayed again.

So that's the basic idea. Here are some additional points to keep in mind:

1. You do not want to load the data each time a request is made. So, before the menu is displayed the first time, your data should be loaded and stored in appropriate data structures. So, assuming a `loadData` function in a module called `db`, an invocation like the following would create and load a data structure called `custDB`

   ```
   (def custDB (db/loadData "cust.txt"))
   ```

   The `custDB` data structure can then be passed as input to any function that needs to act on the data. Note that, once it is loaded, the data is never updated.

2. As a functional language, Clojure uses recursion. If possible, use the `map`, `reduce` and `filter` functions whenever you can. (Note that you may sometimes have to write your own recursive functions when something unique is required).

3. This is a Clojure assignment, not a Java assignment. So Java should not be used for any main functionality. That said, it might be necessary to use Java classes to convert text to numbers in order to do the sales calculations. An example with the `map` function is shown below:

   ```
   (map #(Integer/parseInt %) ["6" "2" "3"])
   ```

4. It is worth noting, however, that this can also be done with Clojure's `read-string` function. This can be used to translate "numeric" strings, including floating point values. So we could do something like:

```
(* 4 (read-string "4.5"))        ; = 18
```

5. The I/O in this assignment is trivial. While it is possible to use complex I/O techniques, it is not necessary to read the text files. Instead, you should just use `slurp`, a Clojure function that will read a text file into a string, as in:
   ```
   (slurp "myFile.txt")
   ```

6. For the input from the user, the `read-line` function can be used. If you print a prompt string to the screen (e.g., "please enter a name", you may want to use `(flush)` before `(read-line)` so that the prompt is not cached (and therefore not displayed).

7. For string processing, you will want to use `clojure.string`. You can view the online docs for a list of string functions and examples (https://clojuredocs.org/clojure.string)

8. Do not worry about efficiency. There are ways to make this program (both the data management and the menu) more efficient, but that is not our focus here. We just want you to use basic functionality to try to get everything working.

DELIVERABLES: Your submission will have just 3 source files. The "main" file will be called `app.clj` and will be used to simply load the information in the text files into a group of data structures and then pass this data to the function that will provide the initial functionality for the app. The second file, `menu.clj` will, as the name implies, provide the interface to the user. The third file will be called `db.clj` and will contain all of the data management code (loading, organizing, etc.). Do not include any data files with your submission, as the markers will provide their own.

PROJECT ENVIRONMENT: It is easy to run a single Clojure file from the command line (i.e., `clojure myfile.clj`). It becomes a little more tedious when the app is made up of multiple files. In practice, large projects are typically built with a tool called Leinengen. Use of Leiningen, however, is overkill for this kind of assignment and will likely make building and testing more problematic for most students.

We will therefore keep things as simple as possible. Your files, both source code and data will be located in the current folder. Each of the three source files listed above will define its own `namespace`. For example:

```
(ns app
  (:require [db])
  (:require [menu]))
```

IMPORTANT: This will allow the files/modules to interact with one another. However, Clojure accesses modules relative to the current `CLASSPATH`. By default, the `CLASSPATH` will probably not be set on your machine, so Clojure will fail to run your program, with errors saying that it cannot find your modules. The simplest thing to do is to simply set the `CLASSPATH` to include the current folder (this is what the graders will do). From the Linux command line, you can just say

```
export CLASSPATH=./
```

Once this is done, you can run your app simply by typing

```
clojure app.clj
```

Note that this `CLASSPATH` variable is temporary and would have to be reset each time you start Linux. If you want it to be automatically configured every time you log in, you could add the export line to the `.bashrc` file in your login folder.

SUBMISSION: Once you are ready to submit, place the .clj files into a zip file. 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.

FINAL NOTE: While you may talk to other students about the assignment, all code must be written individually. Any sharing of assignment code will likely lead to an unpleasant outcome.

Good Luck