Important:

1] Assignments must be submitted on time in order to receive full value. Appropriate late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied.

2] It is the student's responsibility to verify that the assignment has been properly submitted. You can NOT send an alternate version of the assignment at a later date, with the claim that you must have submitted the wrong version at the deadline. Only the original submission will be graded (though you can certainly re-submit multiple times before the due date)

3] These are individual assignments. Feel free to talk to one another and even help others understand the basic ideas. But the source code that you write must be your own.

*4]*** This assignment will be written in Clojure. It is a relatively popular language and, as a result, there are 3rd party Clojure libraries that can be downloaded from the web. YOU CAN NOT USE ANY OF THAT. The purpose of the assignment is to help you become familiar with the basic form of the language – it is not for you to simply download libraries that someone else has written. So 100% of the code you write must work with the standard Clojure distribution (it already includes a huge number of supporting functions). The graders will NOT download any external Clojure modules in order to run your programs – if it doesn't run with the standard distribution, it will not be graded.*

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 using the Clojure language. 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. All text is case-sensitive so “John” and “john” are different people. Finally, there are no duplicate records/customers.

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
```

Again, the data is valid – no duplicates and text is case sensitive.

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 to the following:

   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 exact formatting does not matter. You can use commas as separators or round brackets instead of square brackets. The important thing is that each record lists the ID, followed by the data associated with the ID. Records should be sorted by ID.

   Note that the records are NOT guaranteed to be sorted in the data file (as they are in the illustration above). In addition, ID numbers are not guaranteed to be consecutive numbers (e.g., the IDs could be 7, 3, 2, 9, 14)

2. Same thing for the prod table – it will be sorted by Product ID (again, the data file may not be sorted)

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"]

   Again, the list should be sorted by Sales ID (the data file may not be sorted)

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 (in our simple example).

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.

Note that if a given customer or product does not exist when using menu option 4 or 5, an appropriate response is given (either a "cust/prod not found" message or total sales/count = 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. There are a few final 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 (you can do this any way that you like).
2. This is a Clojure assignment, **not** a Java assignment. So Java should not be embedded in the Clojure code for any important functionality. It might be necessary to use Java classes, for example, to convert text to numbers in order to do the sales calculations. That’s OK, but Java should not be used for much more than that.
3. You can **not** use **while** loops in your program. Clojure provides a **while**, but the purpose of this assignment is to program in a functional style. So any iteration/looping must be done with either recursion or application of the *apply-to-all* style functions (e.g, map, reduce, filters). The *apply-to-all* functions are particularly powerful – use them!
4. The I/O in this assignment is trivial. While it is possible to use complex I/O techniques, it is not necessary just to read the text files. Instead, you should just use “slurp”, a Clojure function that will read a text file into a string. For the input from the user, the “read-line” function can be used.
5. Do not worry about efficiency. There are ways to make this program more efficient (both the data management and the menu), but that is not our focus here. I just want you to use basic functionality to try to get everything working.
6. For production level Clojure development, many programmers use a build automation framework called **leiningen** (similar to ant and maven with Java). However, leiningen is overkill for this assignment and will complicate your life and possibly that of the grader as well. So I suggest that you do NOT use it for this assignment.

   Instead, just create your code and test it directly from the command line. This is what the graders will be doing. Something like ***repl.it*** is likely more than enough for this. If you do use leiningen, or any other development tools, however, you must still make sure that your final submission can run from the command line with no additional configuration required.

DELIVERABLES: Ordinarily we would organize our code into multiple source files. However, this can complicate the grading slightly as Clojure relies on the Java Classpath to locate additional modules. We don’t want to have to deal with inconsistencies between student configurations, so you will just use a single source file for your code. It will be called `sales.clj`. This is all you will

submit. The markers will provide the text files for testing, which will be stored in the same folder as your source file.

Once you are ready to submit, place the sales.clj file into a zip file (I know that you don't need a zip file for one source file but it makes the submission process easier). The name of the zip file will consist of "a2" + 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