Important Info:

1. *Feel free to talk to other students about the assignment. 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. At the end of the term, before final grades are submitted, I will scan all coding assignments during the term to look for plagiarized work. These will be sent to the university for investigation, regardless of the grade that was originally assigned. Do NOT put yourself in this position.*

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.*

3. *The graders will be using a standard 1.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 myapp`*)*

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 about the algorithm 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 can assume that you are an intrepid explorer, searching for buried treasure. Specifically, you are underground, wandering through a series of very dark tunnels. You have no light and no signs to guide your way. All you can do is blindly feel around and wander through the tunnels until you hopefully enter the room that contains the treasure. You can, however, leave a trail of breadcrumbs behind you so that if you come to the end of a tunnel and you can't go any further, you can get back to where you were and try again in another direction.

This may all sound very abstract, but a simple example will illustrate what has to be done. A map of the tunnels will be stored in a simple text file. The map is used by the application to determine if the explorer can move in a certain direction. The map could look like this:

```
---#--###----
-#---#----##-
####-#-#-#-##
---#---#-#---
-#-####---##-
-#------#----
-############
------------@
```

Here, the **–** characters indicate that you are free to move in this direction. The **#** character indicates that you cannot move any further in this direction and you should go somewhere else. The **@** character indicates the location of the treasure. In this case, it is in the bottom right corner, but it could be anywhere in the map.

In your application, you will always begin searching in the top left corner of the map. So when you run your code, you might print something like the following to the screen:

```
This is my challenge:

---#--###--#-
-#---#----##-
####-#-#-#-##
---#---#-#---
-#-####---##-
-#------#----
-############
------------@


Woo hoo, I found the treasure :-)

+++#--###--#-
!#+++#+++-##-
####+#+#+#!##
+++#+++#+#!!!
+#+#####++!##!
+#+++++++#!!!!
+############
++++++++++++@
```

Note that we first print the current map and then indicate success or failure. In this case, we were successful. You will also see that the map has been updated to indicate how the walk was done. The ***+*** characters indicate the path that led to the treasure. The ***!*** characters indicate the tunnels that were tried but did not to lead to a viable path. Note, for example, that after starting in the top left corner, the explorer tried to go down but that path was a dead end. So a ***!*** was used to mark a bad path and then the explorer went to the right, which eventually lead to the treasure.

Let's modify the input file and try another treasure hunt.

```
This is my challenge:

---#--###--#@
-#---#----##-
####-#-#-#-##
---#---#-#---
-#-####---##-
-#------#----
-############
-------------


Uh oh, I could not find the treasure :-(

!!!#!!###!!#@
!#!!!#!!!!##-
####!#!#!#!##
!!!#!!!#!#!!!
!#!####!!!##!
!#!!!!!!#!!!!
!############
!!!!!!!!!!!!!
```

In this case, there was no way to get to the treasure. The ***!*** characters indicate that we tried almost every possible pathway but eventually had to give up (there is one **–** location that we couldn't reach). Oh well.

So that's the entire problem. You will read the map from a file called **map.txt** (in the same folder as the application), try to find the treasure, and then provide a final updated map that shows how you explored the tunnels and, of course, whether you were successful or not.

EVALUATION: For testing purposes, the markers will simply run your code on several different map.txt files to see that you do the right thing. It is possible that more than one path may be possible – you only have to find one of them. You should also make sure that every row in the input file has the same number of columns; otherwise, your code may crash if the map is not valid.

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 `treasure.clj`. This is all you will submit. Again, the markers will provide the *map.txt* test file, which will be stored in the same folder as your source file.

Once you are ready to submit, place the `treasure.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