COMP 348: ASSIGNMENT 3

<u>Important Info</u>: *Please read before going any further*

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.*

3. *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.*

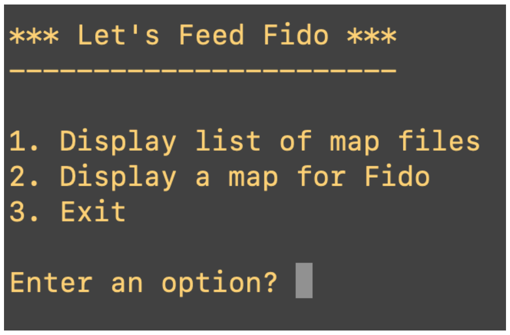
4. *The graders will be using a standard 1.1x distribution of Clojure. You **CANNOT** use any Clojure libraries and/or components that are not found in the standard distribution. The graders will not have these libraries on their systems and, consequently, your programs will not run properly.*

5. *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 fido.clj` or `clojure -M fido.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.

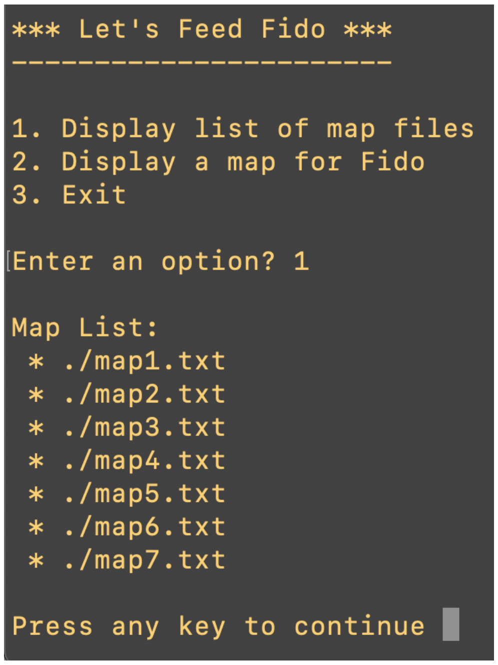
In terms of your task, it can be described as follows. You will be writing an app that allows a dog named Fido to locate her favourite food. She is very hungry, but the food is well hidden. Specifically, it is somewhere inside an old abandoned building. The building has many hallways but also many locked doors. So Fido will wander through the building trying to find her way to the food.

DETAILS This may all sound very abstract, but we will walk through the process to see exactly what must be done. To begin, we will use a small menu to guide Fido's hunt for food. In fact, a menu template is provided with the assignment description. This will ensure that everyone starts at the same place and that your focus is on the main logic of the app. In addition, since Clojure will be a new language for virtually all of you, the template gives you a chance to see how a simple functional program operates. The image below shows the menu.



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

Enter an option?
```

Below the menu is a user prompt. Apart from "Exit", you really just have two options. **Option 1** will be used to display a list of map files that will represent the buildings that Fido can explore (more on these files below). These files will be located in the current folder, along with your Clojure source code. The image below shows what this might look like.



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

Enter an option? 1

Map List:
 * ./map1.txt
 * ./map2.txt
 * ./map3.txt
 * ./map4.txt
 * ./map5.txt
 * ./map6.txt
 * ./map7.txt

Press any key to continue
```

Note the following.

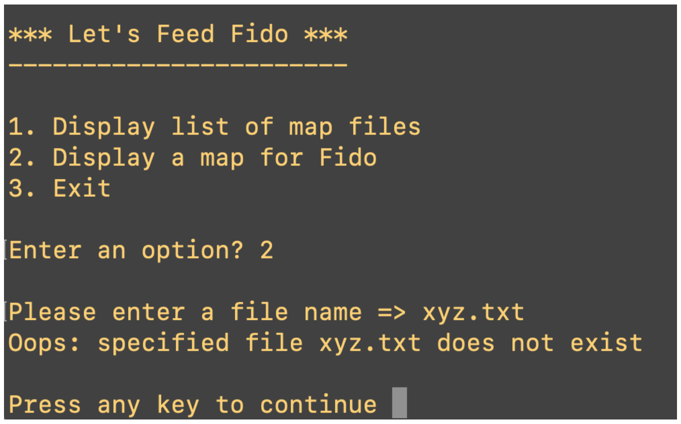
- The "*" symbol is just used to make a nice "bullet" list. It is not part of the file name itself.
- Map files are simply plain text files with a .txt extension
- The list only shows the map files, not all files in the folder (i.e., it does not show your Clojure source files).

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, along with some additional Java methods (e.g., `.isFile`). You can find documentation and samples on the main Clojure web site. You should just need a few lines of code to get a basic file list. That said, it is a little more effort to format the list nicely and exclude the Clojure source files. For that you might need to use something like `re-matches` to make use of a simple regular expression/pattern.

Before going any further, you will note that there is a "Press any key to continue" message listed after the file display. The idea here is that when you hit Enter (or even another character and then Enter), the screen will be cleared and the menu will be displayed again. This logic is not included in the template program. In the template menu, the output just scrolls continuously. So you would need to add this logic to the template. With Clojure you can clear the screen with the following expression `(print (str (char 27) "[2J"))`

Okay, so now it's time to move on to **Option 2**, which is the main focus of the app. Here you will identify the building map that Fido will have to use to find her food. When we select this option, we will be prompted for a file name. As illustrated in the Option 1 file listing, these are just simple text files. Once the user specifies the file name, we will read the file, display it, and proceed with the hunt for food.

Of course, we have to do a little error checking first. If the specified file does not exist, we should provide an appropriate message, as indicated below.



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

Enter an option? 2

Please enter a file name => xyz.txt
Oops: specified file xyz.txt does not exist

Press any key to continue
```

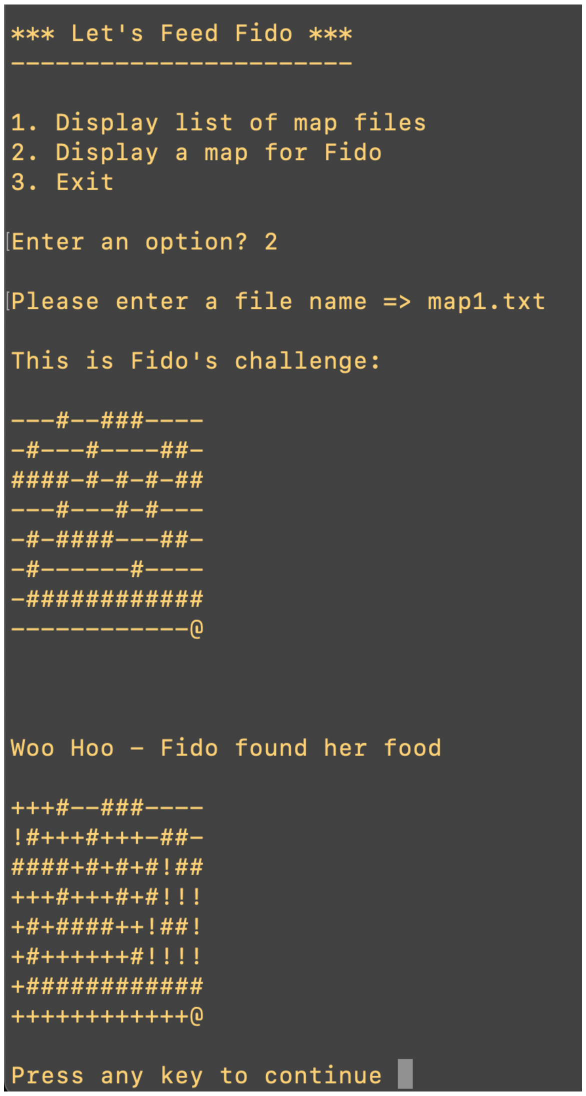
Let's now assume that a valid file name has been given. In this case, we will read the file from disk and prepare for the hunt. So this would be a good time to look at the format of a valid map file. A simple example is listed below.

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

Here, the **–** characters indicate that Fido is free to move in this direction. In other words, a **–** represents an open hallway. In contrast, the **#** character indicates that a locked door prevents any further progress in this direction, so Fido must take try another direction. Finally, the **@** character indicates the location of the food. In this case, it is in the bottom right corner, but it could be anywhere in the map. Note as well that in your app, Fido will always <u>begin</u> searching in the top left corner of the map.

So how do you read the map from disk? Because this is just a simple text file, it is absolutely trivial in Clojure. You will simply use the `slurp` function to read the contents of the file into a string. At that point, your app can store/convert the string into any data structure that you like (i.e., based on how you plan to search the map).

Now back to our app. If the user provides a valid file name – and all goes well – we will see something like the following:

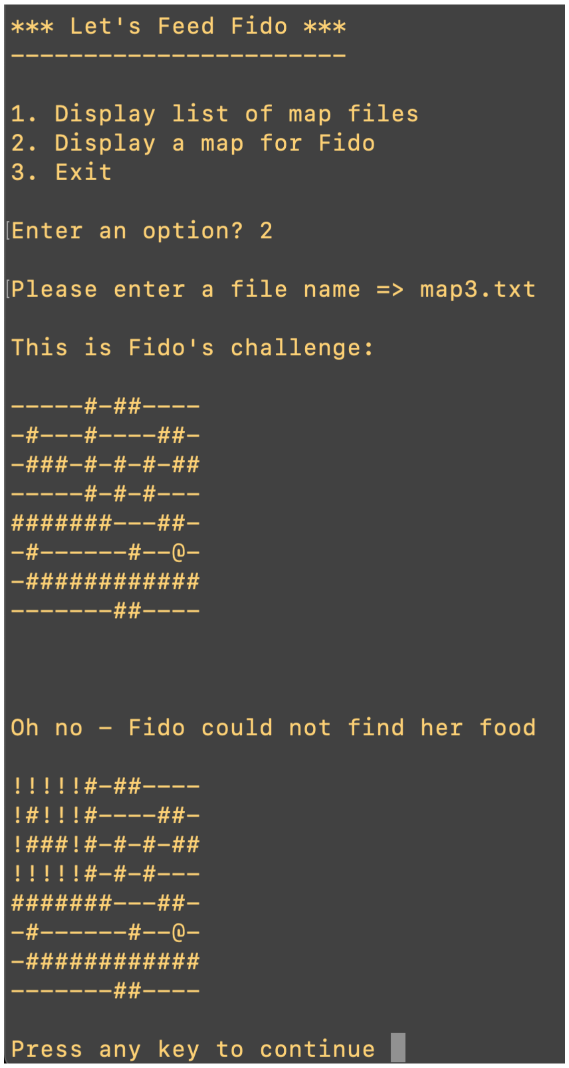


```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

Enter an option? 2

Please enter a file name => map1.txt

This is Fido's challenge:

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



Woo Hoo - Fido found her food

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

Press any key to continue
```

A lot happened here, so let's review. First, we displayed "Fido's Challenge". This is just a listing of the map file that we *slurped* from disk. Below this is the important part – the result of Fido's hunt.

In this case, Fido was successful – she found her food. How do we know this? If you look at the second map, you will see that it is an updated version of the original and includes two new symbols. The ***+*** characters indicate the path that led to the food. In other words, if you start in the top left corner of the map and follow the + chars, the path will take you to the @ symbol. The ***!*** characters , on the other hand, represent paths Fido tried but that did not to lead to the food. Note, for example, that after starting in the top left corner, Fido tried to go down but that path was a dead end. So a ***!*** was used to mark the bad route. You might wonder why Fido needs to remember the bad paths at all. The reason is that if Fido does not take note of this (she has a VERY good memory), she will run into a serious problem if a pathway creates a loop (i.e., she comes back to the same spot she was before). In this case, she could end up circling forever in the building.

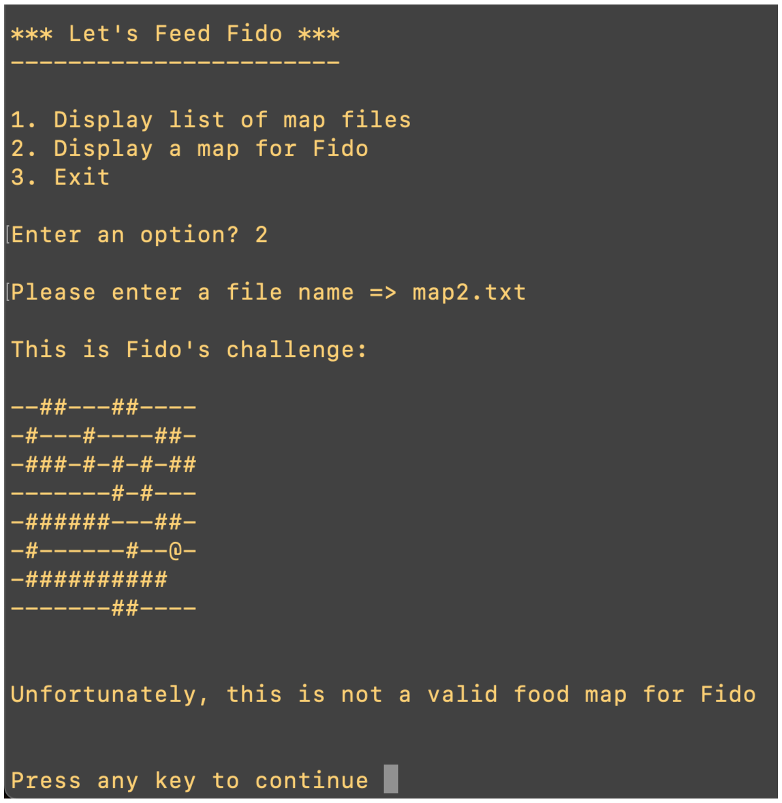
Let's try another input file and see what happens. The image below shows a different search.



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

Enter an option? 2

Please enter a file name => map3.txt

This is Fido's challenge:

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



Oh no - Fido could not find her food

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

Press any key to continue
```

This time, Fido did not find her food, as there was no viable path. We can see this in the updated map. Specifically, she tried several different routes but all of them were blocked. So there is just a set of ! chars and no + chars this time. You will also note that there was a loop here, so Fido was able to figure that out and just stop once she realized that she couldn't go any further.

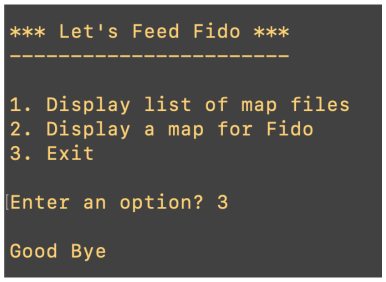
One final consideration is the validity of the map itself. In order for the map to be searchable, it really needs to be a proper rectangle. In other words, there must be at least one "row" and all rows should be the same size (it is *really* miserable to search otherwise). So before Fido starts hunting for food, we must ensure that the map itself is valid. If it is not, we would see something like the following:



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

[Enter an option? 2

[Please enter a file name => map2.txt

This is Fido's challenge:

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


Unfortunately, this is not a valid food map for Fido


Press any key to continue
```

In this case, we can see that the second last row is missing two symbols. So we simply provide an error message in this case and direct the user back to the menu.

Finally, of course, if the user chooses to stop playing (by selecting Exit), we print a Good Bye message and terminate the app. The image below illustrates what this would look like.



```
*** Let's Feed Fido ***
-----------------------

1. Display list of map files
2. Display a map for Fido
3. Exit

[Enter an option? 3

Good Bye
```

And that's it!

THINGS TO KEEP IN MIND: The logic itself is not especially difficult to either describe or understand. That said, navigating through the maps without getting lost, stuck, or looping forever isn't necessarily trivial. Moreover, the app will be written in Clojure and most of you will not be familiar with the general 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 basic map content. Then start taking a few steps through the map and see what happens. Only then should you start adding additional complexity to the process.
- Keep your functions small. Do not try to create functions that do 10 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_only.clj` template. Note that (i) more than one label can be defined in a single `let` expression and (2) there can be more than one `let` in a function (i.e., we can use one `let` to capture a result, and then have another `let` that uses the result captured by the first `let`.
- 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 can avoid the requirement to explicitly implement recursive functions. Note that `mapv` and `filterv` will return vectors rather than lists.
- That said, you may not be able to avoid recursion altogether. There is no simple *apply-to-all* function that will transform the map into its final state.
- If you need to return multiple items from a function, it is probably easiest to do this with a vector. So if we return [value1 value], we can then use `(get result 0)` or `(get result 1)` in the calling function to retrieve either of the two values.
- If you have functions that call one another (i.e., foo calls bar and bar calls foo), you may want to *declare* these functions at the top of the module, as in `(declare foo)`. This works a little like C function prototypes and ensures that you don't have to worry about the order of the function definitions (in most cases, `declare` should not be necessary).
- Basic equality testing is done with `(= foo bar)` or `(not= foo bar)`.
- *Compound* tests can be done with the form `(or test1 test2)` or `(and test1 test2)`.
- Last, but certainly NOT least, never forget that Clojure data types are *immutable*. This means that all changes are *local to the function* unless you return the updated data to the calling function.

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). If you are using this to combine the elements of a vector or sequence, you may want to use `(apply str…)`. This will apply `str` to the individual elements of the sequence, instead of just wrapping the sequence itself in quotes.
- `map/reduce/filter` – built-in recursive processing
- `apply` – apply a function to arguments or elements of a sequence
- `assoc` – update a value in a vector
- `re-matches` – match a pattern against a string
- `slurp` – read the text of a file into a string
- `split-lines` - break a string around the \n character
- `into` – populates a new sequence with elements from one or more other sequences

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

Clojure also 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` or methods like `.isFile()` or `.exists()`. Calling these functions typically uses a dot notation. You can see many simple examples online. For the most part, however, you should not need to do this very often (plus the syntax is messy).

Finally, you are free to use any of the sample code listed on the main Clojure docs site. This is NOT plagiarism. In addition to the docs site itself, you should make use of the **Clojure Cheatsheet**. It provides a nice overview of the main Clojure elements and makes it much easier to identify functions that you might want to use. You can find it at: https://clojure.org/api/cheatsheet

IMPORTANT NOTE: You can NOT use any modules not present in the standard Clojure distribution. So you can't, for example, use external/online packages for manipulating grids or matrices. The graders will not download or import any such packages and your code will simply not run if you rely on these things. It is your job to create and manage your own data structures within the app.

EVALUATION: For testing purposes, the markers will simply check your menu functionality and then run your code on several different map.txt files to see that you do the right thing in various scenarios. Note that it is possible that more than one viable path may be available in a map – you only have to find one of them. Note as well that the maps may be a little bigger or smaller than the ones used in the illustrations above, so you should not hard-code the map size.

DELIVERABLES: The code will be organized into two source files. The first provides the menu and processes user requests, while the second one contains the logic for navigating and manipulating the food maps. The first file is called `fido.clj` and the second is `food.clj`.

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

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

You can also provide a namespace for `fido`. The name doesn't actually matter, however, since `fido` 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 just 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 fido.clj
```

Or perhaps the following (depending on the Clojure version)

```
clojure -M fido.clj
```

Once you are ready to submit, place the `fido.clj` fand `food.clj` files into a zip file. Do not include map 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