Lab 3

Due Wednesday, February 26 by midnight

Handout 7
CSCI 334: Spring 2025

Coding Guidelines

Each question in this assignment should go into the appropriate project directory. For example, the solution to question 1 should be in a folder called "`q1`". When a solution is a program, one should be able to `cd` into the question directory and then run your program by typing the command "`dotnet run`", with additional arguments depending on the question.

Every program should be split into two pieces: a "`Program.fs`" file that contains the `main` method and associated program-startup helpers (if needed), and another "`Library.fs`" file that contains the function(s) of interest in the question. Library code should be contained within a module named "`CS334`". Be sure to provide usage output (defined in `main`) for all programs that require arguments. For full credit, your program should both build and run correctly.

If any of your programs take input from the user, be sure that your program validates input: when a user fails to supply input, or supplies input that does not make sense, your program should print a usage message and return with a nonzero exit code. Users should never experience a program crash in this class; exceptions should be prevented from arising or be caught whenever bad input is encountered. Think through problem corner cases carefully.

Turn-In Instructions

Turn in your work using your assigned `git` repository. The name of your repository will have the form `https://aslan.barowy.net/cs334-s25/cs334-lab03-<USERNAME>.git`. For example, if your CS username is `abc1`, the repository would be `https://aslan.barowy.net/cs334-s25/cs334-lab03-abc1.git`.

You should have received an invite to commit to the repository via email. If you did not receive an email, please contact me right away!

Group Programming Assignment

This is a partner lab. You may work with another classmate if you wish, and you may co-develop solutions. Remember: although you can work on code together, you must each independently write up and submit your solution. No code copying is allowed. Tell me who your partner is by committing a `collaborators.txt` file to your repository. **Be sure to commit this file whether you worked with a partner or not.** If you worked by yourself, `collaborators.txt` should contain something like "I worked by myself." (5 points)

This assignment is due on Wednesday, February 26 by midnight.

Reading

1. **(As Needed)** "F#: The Cool Stuff"
2. **(Required)** "Higher-Order Functions"

Problems

Q1. (25 points) ..... Mapping Functions

Write a function `censor` that takes a list of banned words and a list of words to potentially censor.

```
let censor (banned: string list)(words: string list) : string list = ...
```

Censored words are replaced with `XXXX`. For example,

```
> censor
-   ["party"; "hoxsey"]
-   ["we're"; "going";"to";"skip";"class";"for";"a";"party";"on";"hoxsey"]
- ;;
val it: string list =
  ["we're"; "going"; "to"; "skip"; "class"; "for"; "a"; "XXXX"; "on"; "XXXX"]
```

Your `censor` function must use `List.map`, making use of another function `memberOf`. `memberOf` is a recursive function that takes a list of banned words and a single word to potentially censor, returning `true` if the word is in the banned list and `false` otherwise. Censoring should work regardless of case (i.e., "hoxsey" and "HOXSEY" should be considered the same).

```
let rec memberOf(banned: string list)(word: string) : bool =
```

`memberOf` should not call any other functions except that you may use the following case-insensitive string comparison function,

```
System.String.Equals(s1, s2, System.StringComparison.CurrentCultureIgnoreCase)
```

where `s1` and `s2` are strings.

You should be able to run your program on the command line by supplying a path to a banned word list and a sequence of banned words.

```
$ dotnet run banned.txt I need an extension because I got lost in the steam tunnels
I need an XXXX because I got lost in the XXXX XXXX

$ dotnet run banned.txt I like programming languages more than the fried rice at Blue Mango
I like programming languages more than the XXXX XXXX at XXXX XXXX
```

If the program is run without any arguments, or if the banned file does not exist, it should print out the following usage string and quit with exit code 1:

```
Usage: <banned.txt> <word_1> [... <word_n>]
```

Here are some tips for making the above work.

- To read in a file, you can use the following construct, which opens `filename` and returns a list of strings, one string for each line of the file.

  ```
  IO.File.ReadLines(filename) |> Seq.toList
  ```

- If `filename` does not exist, the above will throw `System.IO.FileNotFoundException` exception. Prevent that from happening by using the `System.IO.File.Exists` method or just handle the exception when it is raised.

- To convert an array to a list, use the `Array.toList` function.
- F# has array-slicing capabilities that let you easily take a subset of an array, just like in Python. See the F# documentation.
- Finally, a list of strings `strs` can be concatenated into a single string with a separator of your choice (e.g., `" "`) like so:

```
System.String.Join(" ", strs)
```

Which words to include in your banned word list is up to you, however, be sure to include at least the set of words that make the above examples work.

The project directory for this question should be called "`q1`". Your `censor` and `memberOf` functions should be in a module called `CS334` stored in a file called `Library.fs` and your `main` function should be in a file called `Program.fs` as in previous labs. As usual, write your program to guarantee that user-provided input makes sense and does not throw an exception.

Q2. (20 points) ..... F# Map for Trees

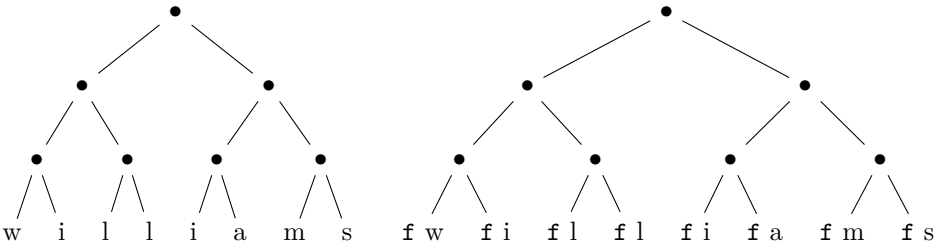
(a) The binary tree datatype

```
type Tree<'a> =
| Leaf of 'a
| Node of Tree<'a> * Tree<'a>
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).
Write the function

```
let rec maptree f t = ???
```

where `f` is a function and `t` is a tree. `maptree` should return a new tree that has the same structure as `t` but where the values stored in `t` have the function `f` applied to them.
Graphically, if `f` is a function that can be applied to values stored in the leaves of tree `t`, and `t` is the tree on the left, then `maptree f t` should produce the tree on the right.



For example, if `f` is the function `let f x = x + 1` then
`maptree f (Node(Node(Leaf 2, Leaf 3), Leaf 4));;`
should evaluate to `Node (Node (Leaf 3,Leaf 4),Leaf 5)`.

(b) In a comment block above your `maptree` definition, explain your definition in one or two sentences. Comment blocks in ML look like the following.

```
(*
 * Says hello to the given name.
 *
 * @param   name The name.
 * @return       Nothing.
 *)
let sayHello name =
    printfn "Hello %s!" name
```

Be sure to provide `@param` and `@return` tags.

(c) What type does F# give to your function? Why isn't it the type `('a → 'a) → Tree<'a> → Tree<'a>`? Provide an answer in the comment block of your `maptree` function.

The project directory for this question should be called "`q2`". You should be able to run your program on the command line by typing, for example, "`dotnet run`" and output like the kind shown above should be printed to the screen. Be sure to provide several examples that demonstrate that your function works correctly.

Q3. (50 points) ..... Grid Game

This question asks you to write a small game called "Grid Game." In F# we do not use `for` or `while` loops and we do not use mutable state, but we are still able to write programs of appreciable complexity. To implement this game in F#, you will need to make extensive use of recursion and pattern matching. If you want to stretch yourself, you are encouraged to use the higher-order functions `map` and `fold` whenever appropriate. However, they are not required.

When a user starts the game, they should be shown the current board and be prompted to make a move.

```
$ dotnet run
[x][ ][ ][=]
[=][=][ ][ ]
[ ][ ][ ][=]
[ ][ ][ ][*]
Enter a move (u, d, l, r, exit):
```

Here, the user's position is marked by an `x`. Walls are indicated by a `=`, and a goal is indicated by a `*`. If you're feeling creative, feel free to customize the game display, but be sure to keep the mechanics as specified in this handout.

The user can move by entering a move and pressing Enter↵. For example, typing r and then Enter↵ moves to the right.

```
[x][ ][ ][=]
[=][=][ ][ ]
[ ][ ][ ][=]
[ ][ ][ ][*]
Enter a move (u, d, l, r, exit): r Enter↵
[ ][x][ ][=]
[=][=][ ][ ]
[ ][ ][ ][=]
[ ][ ][ ][*]
Enter a move (u, d, l, r, exit):
```

If the user attempts to move off the board or into a wall, the game should tell them that they can't do that.

```
[ ][x][ ][=]
[=][=][ ][ ]
[ ][ ][ ][=]
[ ][ ][ ][*]
Enter a move (u, d, l, r, exit): d Enter↵
There's an obstacle in your path.  Try again.  Valid moves are u, d, l, r, exit.
```

When the user finds the goal, the game should tell them that they found it, and it should then quit.

```
[ ][ ][ ][=]
[=][=][ ][ ]
[ ][ ][ ][=]
[ ][ ][x][*]
Enter a move (u, d, l, r, exit): r Enter
You found the goal!
$
```

The user can also explicitly ask to exit the game.

```
Enter a move (u, d, l, r, exit): e x i t Enter
Bye!
$
```

You should implement the following functions, and they should all be kept in your `Library.fs` file in a `CS334` module. The only function in your `Program.fs` should be your `main` function. Before starting coding, you might take some time to plan out how all of the functions relate to each other.

(a) `initBoard` returns an initialized board.

```
initBoard: unit -> Location[][]
```

Be careful when defining this function. A declaration, like `let initBoard = ...`, *is not a function.*

A board should be represented as a `Location[][]` (i.e., a 2D array), where the first index represents the row (y coordinate) and the second index represents the column (x coordinate). A `Location` is defined as follows:

```
type Location =
| Empty
| Wall
| Goal
```

You may use a fixed initial board for this function, and the size of the board is up to you. The board should contain at least one obstacle and at least one goal but it may contain multiple obstacles and multiple goals. The easiest way to do that is to use a nested array literal. For example, here is a `bool[][]` literal.

```
let arr = [| [| true; false |]; [| false; true |] |]
```

For an extra credit opportunity, alter `initBoard` so that it

- initializes the board randomly and
- with a random size and
- at least one goal is attainable.

You may not use loops to solve this, which means that your code must recursively initialize each board position using `System.Random`. Note: this bonus is difficult.

(b) `initPosition` returns an initial player position.

```
initPosition: board: Location[][] -> Position
```

where a `Position` is defined as

```
type Position = { row: int; col: int }
```

This is an example of an F# record type. You can initialize a record using a record literal, like so:

```
let r1 = { row = 101; col = -34 }
```

One convenient feature of records is the ability to use F#'s copy and update shorthand to create a new record based on an old one. For example,

```
let r2 = { r1 with col = 0 }
```

`r2` will have the values `row = 101` and `col = 0`.

As with `initBoard`, you may use a fixed initial position. For an extra credit opportunity, alter `initPosition` so that it

- initializes the position randomly such that
- the game is not instantly won.

Again, you will need to use recursion and `System.Random` to correctly implement the bonus solution. Note: this bonus is not difficult.

(c) The `play` function is where all the action occurs. It should be called once for each turn.

```
play: board: Location[][] -> pos: Position -> unit
```

where `board` is the board and `pos` is the position after the previous turn.

This function should display the board, prompt the user for a valid move, and check that the move does or does not win the game. If the game is won the program should inform the user and then quit. If the game can continue, `play` should call itself recursively with the updated position.

(d) The `display` function prints a board to the screen.

```
display: board: Location[][] -> row: int -> col: int -> pos: Position -> unit
```

where `board` is the board, `row` and `col` are the positions being printed, and `pos` is current position of the player.

At each iteration of the `display`, it should either print something and then call itself recursively, or return a `unit`. You may find the `printfn` and `printf` functions useful; the distinction between the two being that the former function appends a newline at the end of the printed string and the latter does not. The initial call to `display` should always start with a `row` of `0` and a `col` of `0`. The function also prints the player's location on the board.

(e) The `prompt` function repeatedly prompts a user until they provide it with a valid move or an exit command.

```
prompt: board: Location[][] -> pos: Position -> Position
```

where `board` is the board and `pos` is the current position.

`prompt` should call the `System.Console.ReadLine()` function to read input. If the user provides bad input, the function should tell them, and continue to prompt until the input is acceptable. Bad input comes in two forms. It is either invalid, meaning that the move is unrecognized (e.g., the user enters `z`), or it runs the player into an obstacle. The `prompt` function should rely on the helper function `move` (described below) and the `PositionUpdate` type to determine what to do.

```
type PositionUpdate =
| Update of Position
| Invalid
| Obstacle
| Exit
```

If the user tells the `prompt` function that they would like to exit the game, the program should exit immediately using F#'s `exit: int -> 'a` function. Once the user has provided a valid command, `prompt` should return an updated `Position`.

(f) The `move` helper function should process the input entered by a user and return a `PositionUpdate` so that `prompt` knows what to do.

```
move: board: Location[][] -> pos: Position -> movstr: string -> PositionUpdate
```

where `board` is the board, `pos` is the current position, and `movstr` is the string entered by the user (e.g., `d`). Valid values of `movstr` are `u`, `d`, `l`, `r`, and `exit`. The `move` function should rely on the `hitsObstacle` function (described below) to determine whether the user has run into an obstacle.

(g) The `hitsObstacle` function returns `true` if a user has run into an obstacle or runs off the board and `false` otherwise.

```
hitsObstacle: board: Location[][] -> pos: Position -> bool
```

where `board` is the board and `pos` is the proposed position.

(h) Finally, the `gameWon` function returns `true` if the player has moved into a goal location.

```
gameWon: board: Location[][] -> pos: Position -> bool
```

where `board` is the board and `pos` is the position returned by `prompt`.

The project directory for this question should be called "`q3`". You should be able to run this program using "`dotnet run`" without any additional arguments.

Q4. ($\frac{1}{10}^{\text{th}}$ bonus point) ..... Optional: Feedback

I always appreciate hearing back about how easy or difficult an assignment is.

For $\frac{1}{10}^{\text{th}}$ of a bonus to your final grade, please fill out the following Google Form.