Lab 2

Due Monday, February 19 by 10:00pm

Handout 6
CSCI 334: Spring 2024

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 the git repository assigned to you. The name of the git repository will have the form https://aslan.barowy.net/cs334-s24/cs334-lab02-<USERNAME>.git. For example, if your CS username is abc1, the repository would be https://aslan.barowy.net/cs334-s24/cs334-lab02-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 Monday, February 19 by 10:00pm.

Reading

1. **(Required)** "Advanced F#"
2. **(As needed)** Microsoft's Official F# Documentation

Problems

Q1. (45 points) .... Zipping and Unzipping

(a) Write a function `zip(xs: 'a list)(ys: 'b list) : ('a * 'b) list` that computes the product of two lists of arbitrary length. You should use pattern matching to define this function:

```
> zip [1;3;5;7] ["a";"b";"c";"d"];;
val it : (int * string) list = [(1, "a"); (3, "b"); (5, "c"); (7, "d")]
```

When one list is longer than the other, repeatedly pair elements from the longer list with the last element of the shorter list.

```
> zip [1;3] ["a";"b";"c";"d"];;
val it : (int * string) list = [(1, "a"); (3, "b"); (3, "c"); (3, "d")]
```

In the event that one or both lists are completely empty, return the empty list. Note that in `dotnet fsi`, calling the function as below will produce an error because F# cannot determine the type of the element of an empty list.

```
> zip [1;3;5;7] [];;

  zip [1;3;5;7] [];;
  ^^^^^^^^^^^^^^^^


code/stdin(14,1): error FS0030: Value restriction. The value 'it'
has been inferred to have generic type
    val it : ((int * int * int * int) * '_a) list
Either define 'it' as a simple data term, make it a function with
explicit arguments or, if you do not intend for it to be generic,
add a type annotation.
```

To make empty lists work, explicitly provide a type for the return value.

```
> let xs : (int * int) list = zip [1;3;5;7] [];;
val xs : (int * int) list = []
```

(b) Write the inverse function, `unzip(xs: ('a * 'b) list) : 'a list * 'b list`, which behaves as follows:

```
> unzip [(1,"a"); (3,"b") ;(5,"c"); (7,"de")];;
val it : int list * string list = ([1; 3; 5; 7], ["a"; "b"; "c"; "de"])
```

(c) Write `zip3(xs: 'a list)(ys: 'b list)(zs: 'c list) : ('a * 'b * 'c) list`, that zips three lists.

```
> zip3 [1;3;5;7] ["a";"b";"c";"de"] [1;2;3;4];;
val it : (int * string * int) list =
  [(1, "a", 1); (3, "b", 2); (5, "c", 3); (7, "de", 4)]
```

You must use `zip` in your definition of `zip3`.

(d) Provide a `main` function that exercises all of the above cases, plus a few more that you think of yourself.

The project directory for this question should be called "`q1`". You should be able to run this program using "`dotnet run`" without any additional arguments.

Q2. (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. Therefore, to implement this game in F#, you will need to make extensive use of recursion and pattern matching.

When the 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): rEnter↵
[ ][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): dEnter↵
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): rEnter↵
You found the goal!
$
```

The user can also explicitly ask to exit the game.

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

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 "`q2`". You should be able to run this program using "`dotnet run`" without any additional arguments.