Introduction

The major theme of this assignment is to practice I/O in Haskell.

For full credit, complete EITHER Part 1. (GTicTacToe) and Part 2. (Cipher) OR Part 1. (GTicTacToe) and Part 3. (Line Editor), based on how you can most effectively further your understanding of Haskell and functional programming. Briefly,

- Part 2. (Cipher): A simple command-line encryption/decryption program. Understand the logical separation of pure functions from I/O actions. Gain experience using standard library I/O actions.

Note: Although each part has a different number of questions and points, each part will be weighted equally in the Homework 05 grade.

Read and follow the course Haskell Style Guide (https://www.cs.rit.edu/~mtf/teaching/20191/psfp/style.html). Especially, look for opportunities to eliminate unnecessary complexity (use anonymous functions, use partial applications, use sectioning, use functionals). Also, look for opportunities to use standard library functions.
Generalized Tic-Tac-Toe

Download GTicTacToe.hs, which defines types for players and boards (for generalized Tic-Tac-Toe games).

A number of games, such as Tic-Tac-Toe and Gomoku, may be seen as special cases of $m, n, k$-games: two player games, in which the players alternate marking spaces on an $m \times n$ board and the winner is the first player to mark $k$ spaces in a row horizontally, vertically, or diagonally. Thus, Tic-Tac-Toe is the 3,3,3-game and (free-style) Gomoku is the 15,15,5-game.

The Player type is defined as follows:

```haskell
import Data.Either
import Data.List
```

The Board type is defined as follows:

```haskell
import qualified Data.Map.Strict as Map

import Text.Read (readMaybe)
-- readMaybe :: Read a => String -> Maybe a
```

A board is comprised of the integers $m$, $n$, and $k$ and a mapping from positions (pairs of integers) to $\text{Maybe} \, \text{Player}$. Note the invariant asserts that in an $m,n,k$-game, $m$ is greater than zero, $n$ is greater than zero, and $k$ is greater than one. Furthermore, note the invariant asserts that the keys of the mapping exactly correspond to positions $(i,j)$ such that $0 \leq i < m$ and $0 \leq j < n$. Hence, the $\text{Map.Map} (\text{Int,Int}) \ (\text{Maybe} \, \text{Player})$ represents an $m \times n$ board: a $\text{Nothing}$ at position $(i,j)$ represents a free space, while a $\text{Just} \, X$ or $\text{Just} \, 0$ at position $(i,j)$ represents a marked space.

Be sure to review the documentation of the $\text{Data.Map.Strict}$ module (https://downloads.haskell.org/ghc/8.6.5/docs/html/libraries/containers-0.6.0.1/Data-Map-Strict.html).

1. (5pts) Define a function

   $$\text{new} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Board}$$

   such that $\text{new} \, m \, n \, k$ returns an initial board for an $m,n,k$-game; all spaces are unmarked in an initial board. Examples:

   - $\text{new} \, 3 \, 3 \, 3$
     $\sim \text{Board} \, 3 \, 3 \, 3 \ (\text{fromList} \ [(0,0),\text{Nothing}),(0,1),\text{Nothing}),(0,2),\text{Nothing}),$
     $\ (1,0),\text{Nothing}),(1,1),\text{Nothing}),(1,2),\text{Nothing}),$
     $\ (2,0),\text{Nothing}),(2,1),\text{Nothing}),(2,2),\text{Nothing}]))$

   - $\text{new} \, 5 \, 3 \, 3$
     $\sim \text{Board} \, 5 \, 3 \, 3 \ (\text{fromList} \ [(0,0),\text{Nothing}),(0,1),\text{Nothing}),(0,2),\text{Nothing}),$
     $\ (1,0),\text{Nothing}),(1,1),\text{Nothing}),(1,2),\text{Nothing}),$
     $\ (2,0),\text{Nothing}),(2,1),\text{Nothing}),(2,2),\text{Nothing}),$
     $\ (3,0),\text{Nothing}),(3,1),\text{Nothing}),(3,2),\text{Nothing}),$
     $\ (4,0),\text{Nothing}),(4,1),\text{Nothing}),(4,2),\text{Nothing}))))$

   - $\text{new} \, 3 \, 5 \, 3$
     $\sim \text{Board} \, 3 \, 5 \, 3 \ (\text{fromList} \ [(0,0),\text{Nothing}),(0,1),\text{Nothing}),(0,2),\text{Nothing}),$
     $\ (0,3),\text{Nothing}),(0,4),\text{Nothing}),(1,0),\text{Nothing}),(1,1),\text{Nothing}),(1,2),\text{Nothing}),$
     $\ (1,3),\text{Nothing}),(1,4),\text{Nothing}),(2,0),\text{Nothing}),(2,1),\text{Nothing}),(2,2),\text{Nothing}),$
     $\ (2,3),\text{Nothing}),(2,4),\text{Nothing}))))$
2. (5pts) Define a function

\[ \text{draw} :: \text{Board} \to \text{String} \]

such that \( \text{draw} b \) returns a string that, when printed, is a two-dimensional visualization of the board. The visualization of an \( m, n, k \)-game should have \( m \times n \) spaces, delineated by \( m - 1 \) horizontal lines (drawn with \( - \)) and \( n - 1 \) vertical lines (drawn with \( | \)) with \( (m - 1) \times (n - 1) \) intersections (drawn with \( + \)). Marked spaces should be drawn with \( X \) or \( O \) and free spaces should be drawn with a space. Position \( (0,0) \) should be at the upper-left, position \( (m - 1,0) \) should be at the lower-left, position \( (0,n - 1) \) should be at the upper-right, and position \( (m - 1,n - 1) \) should be at the lower-right.

Examples:

- \( \text{draw (new 3 3 3)} \sim \) " | | \n-+-+-\n | | \n-+-+-\n | | \n"

- \( \text{putStr (draw (new 3 3 3))} \Rightarrow \\
  | | \\
  -+-+-
  | | \\
  -+-+-
  | | \\

- \( \text{putStr (draw (new 5 3 3))} \Rightarrow \\
  | | \\
  -+-+-
  | | \\
  -+-+-
  | |
  -+-+-
  | |
  -+-+-
  | |

- \( \text{putStr (draw (new 3 5 3))} \Rightarrow \\
  | | | | \\
  -+++-+-+-
  | | | |
  -+++-+-+-
  | | | |

See \( \text{tictactoeA.stdout, tictactoeB.stdout, three_six_threeA.stdout, three_six_threeB.stdout, gomokuA.stdout, and gomokuB.stdout} \) for additional examples.
3. (5pts) Define a function 

- \( \text{kiars} :: \text{Board} \rightarrow \left[ \left( (\text{Int}, \text{Int}), \text{Maybe Player} \right) \right] \)

such that \( \text{kiarsBoard} \ b \) returns a list of all \( k \)-in-a-row sequences of spaces from the board \( b \); each space is annotated with its position.

Examples:

- \( \text{all } \lambda \ ps \rightarrow \text{length } ps = 3 \) (\( \text{kiars} \ \text{(new 3 3 3)} \)) \( \sim \) True
- \( \text{all } \lambda \ ps \rightarrow \text{length } ps = 3 \) (\( \text{kiars} \ \text{(new 5 3 3)} \)) \( \sim \) True
- \( \text{all } \lambda \ ps \rightarrow \text{length } ps = 3 \) (\( \text{kiars} \ \text{(new 3 5 3)} \)) \( \sim \) True
- \( \text{all } \lambda \ ps \rightarrow \text{length } ps = 5 \) (\( \text{kiars} \ \text{(new 3 5 5)} \)) \( \sim \) True
- \( \text{length } (\text{kiars} \ \text{(new 3 3 3)}) \) \( \sim \) 8
- \( \text{length } (\text{kiars} \ \text{(new 5 3 3)}) \) \( \sim \) 20
- \( \text{length } (\text{kiars} \ \text{(new 3 5 3)}) \) \( \sim \) 20
- \( \text{length } (\text{kiars} \ \text{(new 15 15 5)}) \) \( \sim \) 572
- \( \text{kiars} \ \text{(new 3 3 3)} \)
  \( \sim \) \[
  \left[ \left( (0,0), \text{Nothing} \right), \left( (1,0), \text{Nothing} \right), \left( (2,0), \text{Nothing} \right) \right],
  \left[ \left( (0,0), \text{Nothing} \right), \left( (0,1), \text{Nothing} \right), \left( (0,2), \text{Nothing} \right) \right],
  \left[ \left( (0,0), \text{Nothing} \right), \left( (1,1), \text{Nothing} \right), \left( (2,2), \text{Nothing} \right) \right],
  \left[ \left( (0,1), \text{Nothing} \right), \left( (1,1), \text{Nothing} \right), \left( (2,1), \text{Nothing} \right) \right],
  \left[ \left( (0,2), \text{Nothing} \right), \left( (1,2), \text{Nothing} \right), \left( (2,2), \text{Nothing} \right) \right],
  \left[ \left( (2,0), \text{Nothing} \right), \left( (1,1), \text{Nothing} \right), \left( (2,0), \text{Nothing} \right) \right],
  \left[ \left( (2,0), \text{Nothing} \right), \left( (2,1), \text{Nothing} \right), \left( (2,2), \text{Nothing} \right) \right]
\]

Note that the order in which the position/space pairs appear in a \( k \)-in-a-row sequence and the order in which the \( k \)-in-a-row sequences appear in the result list do not matter. However, no position/space pair should appear more than once in a \( k \)-in-a-row sequence and no \( k \)-in-a-row sequence should appear more than once in the result list.
4. (5pts) Define a function

- \texttt{promptForAndValidate :: Read a => String \to (a \to \text{Bool}) \to \text{IO} a}

such that \texttt{promptForAndValidate name chk} returns an I/O action that, when run, repeatedly outputs a prompt message using the string \texttt{name} and inputs a line until the line may be successfully converted from a string to a value (using \texttt{readMaybe}) and that value satisfies the predicate \texttt{chk}; if a line may not be successfully converted from a string to a value or that value does not satisfy the predicate \texttt{chk}, then the I/O action outputs an error message using the string \texttt{name} before looping; if a line may be successfully converted from a string to a value and that value satisfies the predicate \texttt{chk}, then that value is the result of the I/O action.

Examples:

- \texttt{promptForAndValidate "a non-negative integer" (>0) \to}
  
  Enter a non-negative integer: 
  5
  \to 5

- \texttt{promptForAndValidate "a non-negative integer" (>0) \to}
  
  Enter a non-negative integer: 
  five
  ** Invalid input for a non-negative integer. **
  Enter a non-negative integer: 
  -1
  ** Invalid input for a non-negative integer. **
  Enter a non-negative integer: 
  10
  10
  \to 10

- \texttt{promptForAndValidate "a pair that sums to 42" (\(\langle a, b \rangle \to a + b == 42\)) \to}
  
  Enter a pair that sums to 42: 
  42
  ** Invalid input for a pair that sums to 42. **
  Enter a pair that sums to 42: 
  \((4,2)\)
  ** Invalid input for a pair that sums to 42. **
  Enter a pair that sums to 42: 
  \((42,42)\)
  ** Invalid input for a pair that sums to 42. **
  Enter a pair that sums to 42: 
  \((21,21)\)
  \to \((21,21)\)
5. (15pts) Define an I/O action

- \texttt{play :: Board} \rightarrow \texttt{Player} \rightarrow \texttt{IO ()}

that, when run, coordinates the play of an \(m,n,k\)-game. The \texttt{Board} argument is the “current” board and the \texttt{Player} argument indicates the player to make the next move. Thus, the I/O action \texttt{play b p} should:

- output a blank line;
- output a visualization of the “current” board (hint: use \texttt{draw});
- prompt the appropriate player for his/her move. A move is a position (pair of integers); a move must be valid with respect to the “current” board (hint: use \texttt{promptForAndValidate});
- if the move ends the game, then output a message declaring the winner or declaring a draw;
- if the move does not end the game, then recursively call \texttt{play} with the “new” board and the other player.

See Figure 1 for an example interaction.

The \texttt{tictactoeA.stdin}, \texttt{tictactoeB.stdin}, \texttt{three_six_threeA.stdin}, \texttt{three_six_threeB.stdin}, \texttt{gomokuA.stdin}, and \texttt{gomokuB.stdin} files provide sample inputs, while the \texttt{tictactoeA.stdout}, \texttt{tictactoeB.stdout}, \texttt{three_six_threeA.stdout}, \texttt{three_six_threeB.stdout}, \texttt{gomokuA.stdout}, and \texttt{gomokuB.stdout} files provide the corresponding outputs. Note that the .stdin files contain only the inputs, while the .stdout files contain only the outputs. The \texttt{tictactoeA.stdinout}, \texttt{tictactoeB.stdinout}, \texttt{three_six_threeA.stdinout}, \texttt{three_six_threeB.stdinout}, \texttt{gomokuA.stdinout}, and \texttt{gomokuB.stdinout} files provide interleaved inputs and outputs, as might appear on the screen when running the program.

A reference solution executable on \texttt{queeg.cs.rit.edu} or the Linux ICL machines is available on the CS Department file system at:

\texttt{/usr/local/pub/mtf/psf-20191/homework05/GTicTacToe}

Hints:

- Because \texttt{instance Read Int} and because \texttt{instance (Read a, Read b) \Rightarrow Read (a, b)}, the type \texttt{(Int, Int)} is an instance of \texttt{Read}.
- Define a function \texttt{isWin :: Board} \rightarrow \texttt{Player} \rightarrow \texttt{Bool} that uses \texttt{kiars} to determine if a player has won.
- Define a function \texttt{isDraw :: Board} \rightarrow \texttt{Bool} that determines if the game is a draw. (There is a one-line solution exploiting the fact that \texttt{Map (Int,Int)} is an instance of \texttt{Foldable}.)
G TicTacToe

Enter m:
-3
** Invalid input for m. **
Enter m:
3
Enter n:
3
Enter k:
three
** Invalid input for k. **
Enter k:
3

| | |
-+-+-
| | |
-+-+-
| | |
Enter Player X's move:
(0,0)
X | |
-+-+-
| | |
-+-+-
| | |
Enter Player O's move:
center
** Invalid input for Player O's move. **
Enter Player O's move:
(1,1)
X | |
-+-+-
| | |
-+-+-
| | |
Enter Player X's move:
(0,0)
** Invalid input for Player X's move. **
Enter Player X's move:
(0,1)
X |X|
-+-+-
| | |
-+-+-
| | |
Enter Player O's move:
(2,0)
X |X|
-+-+-
| | |
-+-+-
| | |
Enter Player X's move:
(1,2)
X |X|
-+-+-
| | |
-+-+-
| | |
Enter Player O's move:
(0,2)
X |X|O
-+-+-
|O|X
-+-+|
| | |
Player O wins!

Figure 1: GTicTacToe main Example
1. (10pts) Define functions

- `encrypt :: Int -> String -> String`
- `decrypt :: Int -> String -> String`

that performs (simple) encryption/decryption of a string with an integer key $k$ that satisfies $0 \leq k < 26$. The encryption algorithm works as follows:

- Break the input string into lines.
- For each (ASCII lower-case and ASCII upper-case) letter in the $i$th string (0-based indexing), rotate the letters forward by $(i + 1) \times k$. (For example, with $k = 7$ and $i = 0$, 'a' becomes 'h' and 'X' becomes 'E', and with $k = 19$ and $i = 5$, 'a' becomes 'k' and 'X' becomes 'H'.) Non-letter characters are unchanged.
- Reverse the lines.
- Combine the lines into the output string.

Examples:

- `encrypt 0 "1. apple!
2. banana?
3. cherry$
4. date#"
  ~> "4. date#
3. cherry$
2. banana?
1. apple!"
- `encrypt 1 "1. apple!
2. banana?
3. cherry$
4. date#"
  ~> "4. hexi#
3. fhkub$
2. dcpccp
1. bqqmfi"
- `encrypt 7 "1. apple!
2. banana?
3. cherry$
4. date#"
  ~> "4. fcvg#
3. xczmmt$
2. pobobo
1. hwswl"
- `encrypt 19 "1. apple!
2. banana?
3. cherry$
4. date#"
  ~> "4. byrc#
3. hmnjwd$
2. mmzmzm
1. tiixex"
- `decrypt 0 "4. date#
3. cherry$
2. banana?
1. apple!"
  ~> "1. apple!
2. banana?
3. cherry$
4. date#"
- `decrypt 1 "4. hexi#
3. fhkub$
2. dcpccp
1. bqqmfi"
  ~> "1. apple!
2. banana?
3. cherry$
4. date#"
- `decrypt 7 "4. fcvg#
3. xczmmt$
2. pobobo
1. hwswl"
  ~> "1. apple!
2. banana?
3. cherry$
4. date#"
- `decrypt 19 "4. byrc#
3. hmnjwd$
2. mmzmzm
1. tiixex"
  ~> "1. apple!
2. banana?
3. cherry$
4. date#"

Note: Encrypting or decrypting with a key of 0 simply reverses the lines of the string.

Hints:

- Useful Prelude functions include:
  - `lines :: String -> [String]`
  - `unlines :: [String] -> String`
- Useful Data.Char functions include:
  - `ord :: Char -> Int`
  - `chr :: Int -> Char`
  - `isAsciiUpper :: Char -> Bool`
  - `isAsciiLower :: Char -> Bool`
2. (10pts) Define an I/O action

- main :: IO ()

that performs command-line encryption/decryption of a file. In particular, the Cipher program expects 4 command-line arguments:

- a mode \( m \), either \texttt{enc} or \texttt{dec}
- an integer key \( k \), satisfying \( 0 \leq k < 26 \)
- an input file name \texttt{inFile}
- an output file name \texttt{outFile}

If the Cipher program is executed with the wrong number of command-line arguments or the mode or integer key are invalid, then a simple usage message should be displayed. If the Cipher program is executed with valid arguments, then the appropriate encryption/decryption of the input file should be written to the output file.

Hints:

- Useful Prelude types and functions include:
  - \texttt{type FilePath = String}
  - \texttt{readFile :: FilePath -> IO String}
  - \texttt{writeFile :: FilePath -> String -> IO ()}
- Useful System.Environment functions include:
  - \texttt{getArgs :: IO [String]}
- Useful Text.Read functions include:
  - \texttt{readMaybe :: Read a => String -> Maybe a}

The \texttt{fruits.txt}, \texttt{fruits.enc0}, \texttt{fruits.enc1}, \texttt{fruits.enc7}, \texttt{fruits.enc19}, \texttt{gettysburg.txt}, and \texttt{gettysburg.enc7} files provide sample plain-text and cipher-text files.

A reference solution executable on \texttt{queeg.cs.rit.edu} or the Linux ICL machines is available on the CS Department file system at:

/usr/local/pub/mtf/psfp-20191/homework05/Cipher
Line Editor

Download LineEditor.hs.

This is an open-ended design and implementation problem for those who are ready to tackle the development of a small Haskell program. In essence, the goal is to design and implement a text-based, line-oriented editor, somewhat in the style of ed and ex; see https://en.wikipedia.org/wiki/Line_editor.

The basic premise of a text-based, line-oriented editor is that the editor represents the document as a collection of lines with a reference to the “current” line. Each iteration of an interactive loop, it prompts for a command and process the command. Commands include displaying a portion of the document around the current line, editing/replacing the current line with new text, inserting a new line of text before the current line, deleting the current line of text, navigating in the document, loading and saving files, undoing and redoing edits, etc.

See Figure 2 for an example interaction.

1. (30pts) Define an I/O action
   • main :: IO ()
     that executes a line-editor program.

2. (10pts) In the README: comment, briefly describe the design and functionality of the line editor.

Required functionality includes the following commands:

- v: View the three lines before the current line, the current line, and the three lines after the current line.
- e text: Edit/replace the current line with text.
- i text: Insert text before the current line, becoming the current line.
- d: Delete the current line (making the next line the current line).
- f: Move forward a line.
- b: Move backward a line.
- 1 file: Discard the current document, load file as the current document and position cursor at (before) the first line.
- s file: Save the current document to file.
- u: Undo the last change to the document. (Note: Undoing “skips” over display and navigation commands and may leave the cursor at a different position.)
- q: Quit
- ?: Help; display the list of commands and a brief description of each.

Additional functionality might include:

- v n: View the n lines before the current line, the current line, and the n lines after the current line.
- V l: View the three lines before the lth line of the document, the lth line of the document, and the three lines after the lth line of the document. (Maybe a negative l counts backwards from the end of the document.)
- V l n: View the n lines before the lth line of the document, the lth line of the document, and the n lines after the lth line of the document.
- a: Move to (before) the first line of the document.
- z: Move to (after) the last line of the document.
- f n: Move forward n lines.
- b n: Move backward n lines.
- j l: Jump to (before) the lth line of the document. (Maybe a negative l counts backwards from the end of the document.)
- s text: Search (forward) and move to the next line that contains text.
Line Editor

> l gettysburg.txt
> v
> * Gettysburg Address
>   . Abraham Lincoln
>   . Four score and seven years ago our fathers brought
>     f
>     f
>     v
>     . Gettysburg Address
>     . Abraham Lincoln
>     *
>     . Four score and seven years ago our fathers brought
>     . forth on this continent a new nation, conceived in
>     . liberty, and dedicated to the proposition that all
>   > i 1863-11-19
>   > v
>   . Gettysburg Address
>   . Abraham Lincoln
>   * 1863-11-19
>   .
>   . Four score and seven years ago our fathers brought
>   . forth on this continent a new nation, conceived in
>   > e November 19, 1863
>   > v
>   . Gettysburg Address
>   . Abraham Lincoln
>   * November 19, 1863
>   .
>   . Four score and seven years ago our fathers brought
>   . forth on this continent a new nation, conceived in
>   > b
>   > b
>   > v
>   * Gettysburg Address
>   . Abraham Lincoln
>   . November 19, 1863
>   > u
>   > v
>   . Gettysburg Address
>   . Abraham Lincoln
>   * 1863-11-19
>   .
>   . Four score and seven years ago our fathers brought
>   . forth on this continent a new nation, conceived in
>   > s gettysburg.txt
>   > q

Figure 2: LineEditor main Example
- S re: Search (forward) and move to the next line that has a substring matched by the regular expression re.

- r pat text: Replace all occurrences of pat in the current line with text. (How should pat and text be delimited if either or both could contain spaces?)

- R pat text: Globally replace all occurrences of pat in the document with text.

- r: Redo the last change to the document. (Note: Redoing “skips” over display and navigation commands and may leave the cursor at a different position. Note: It is illegal to redo after an edit (i.e., in order to redo the last edit, there must have been no edits since the undo).)

- u n: Undo the last n changes to the document.

- r n: Redo the last n changes to the document.

- Warn and prompt for confirmation when quitting or loading if the current document is modified (or introduce confirming commands q! and l! file).

Some design considerations:

- Use ideas from the EditList.hs recitation to represent the document and the undo/redo log.

- Strive for a clean separation of pure functions from I/O actions.

- Strive for a design where help-text is never out of date. For example, a single data structure that holds both the help text for a command and the operation performed by the command.

Requirements and Submission

Your submission must :load into ghci without errors; submissions that have parse errors or type errors will receive no credit. Submissions that violate code style guidelines will lose up to 25%.

Submit either GTicTacToe.hs and Cipher.hs or GTicTacToe.hs and LineEditor.hs to the Homework05 Assignment on MyCourses by the due date.