University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Functional Programming (H)			
Coursework Number	Programming Assignment			
Deadline	Time:	16:30	Date:	6th December 2024
% Contribution to final course mark	25			
Solo or Group ✓	Solo	X	Group	
Anticipated Hours	20			
Submission Instructions	Via Moodle: submit your Deck.hs and Game.hs in a zip file with the filename corresponding to your student ID. See handout for further details.			
Please Note: This Coursework cannot be Re-Assessed				

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Functional Programming (H) 2024-25
Assessed Exercise: Solitaire

Simon Fowler

v3, November 15, 2024

This assignment is due Friday 6th December 2024, at 4:30PM.

Please read over this handout carefully and look over the supplied code before beginning work, as some of your questions may be answered later. Please let us know if there are any apparent errors or bugs. We will update this handout to add clarifications and fix any issues; such updates will be announced on Teams. The handout is versioned and the most recent version will always be available from the Moodle page.

Submission When you are ready to submit, please create a ZIP file containing only your `Deck.hs` and `Game.hs` files, and a `status.txt` file detailing the exercises you have attempted. The `status.txt` does not need to include any more information and will not be marked.

Please submit a single `zip` file to the Moodle submission box, where the name of the file corresponds to your student ID (for example, `2600000F.zip`). Your submission will be assessed according to the usual University policies on lateness and plagiarism. Remember that this is *individual* work and you must not share code with others.

There are a total of 66 marks available.

Important: *We expect your code to compile. Submissions that do not compile because of syntax or type errors will be capped at 15 marks.*

Working on the Lab Machines You are welcome to work on either your own machine or the lab machines. Unfortunately the native Stack installation on the lab machines is unreliable this year, and we will get this fixed for next year. In the meantime, we have created a Virtual Machine Image with GHC and Stack pre-installed, and you can use this with VirtualBox (which is installed on the lab machines). After downloading the image, import it following these instructions. You will then be able to run `stack` from a terminal window.

Solitaire

Solitaire is a single-player card game where a player aims to sort all cards in ascending order onto four *pillars* or *foundation stacks* for each suit. Note that there are multiple variations of Solitaire. In this assignment we will consider 1-draw Klondike Solitaire.

Figure 1 shows a screenshot of Google's implementation of Solitaire (you can play this yourself by googling "solitaire" – we recommend you play a few games to familiarise yourself with the rules and gameplay). There are four main components to the game:

- The *deck* is a face-down pile of cards.
- The *discard pile* is a face-up pile of cards that have been drawn from the deck.

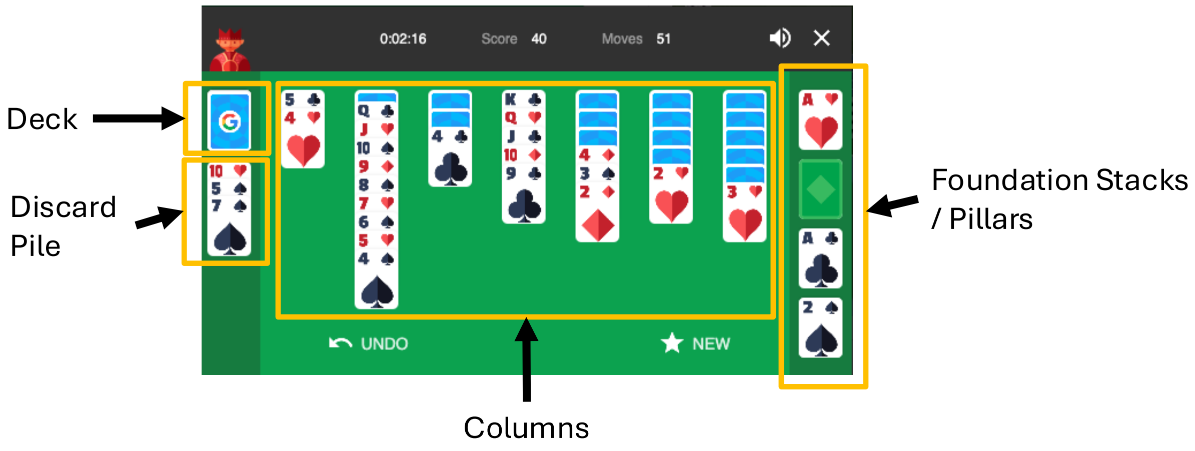
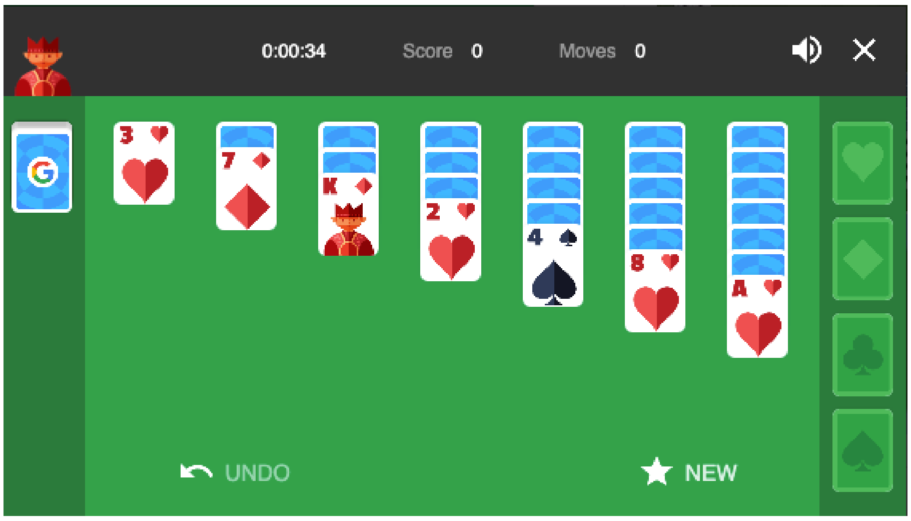


Figure 1: Solitaire game (Google's implementation)

- There are four *foundation stacks* or *pillars*, one for each suit. These are initially empty. The goal of the game is to add cards to the pillars in ascending order (so beginning with Ace, then 2, all the way up to King). The game is won when a King is on top of every foundation stack.
- There are 7 *columns* or *stacks*. These initially consist of some face-down cards and a single face-up card. A face-down card is turned over whenever it reaches the bottom of the column. The face-up cards in each column must be arranged in descending order with alternating colours.

The game begins with one face-up card in the first column, one face-down and one face-up card in the second, two face-down cards and one face-up card in the third, and so on:



In each turn, the player can make the following moves:

- Draw a card from the deck; if the deck is empty then the discard pile is flipped over and used as the new deck.
- Move a card from the top of the discard pile to a column or pillar.
- Move any number of visible cards from the bottom of one column to another.
- Move a card from the bottom of a column to a pillar.
- Move a card from the top of a pillar to the bottom of a column.

In each case the columns must remain in descending order with alternating colours, otherwise the move is invalid. Similarly the pillars can only be built up in ascending order. The aim of the game is to sort all cards in ascending order on the pillars, meaning that the top card of each pillar is a King.

In this assignment, you will implement a text-based version of Solitaire in Haskell. We have given you some code that models the basic data structures for the game, and a parser for user input.

Skeleton Code

The skeleton code has the following structure:

```
.
├── Setup.hs
├── app
│   └── Main.hs
├── package.yaml
├── src
│   ├── CommandParser.hs
│   ├── Deck.hs
│   ├── Error.hs
│   └── Game.hs
└── stack.yaml
```

Here is the description of each file; you only need to modify those marked with a star (⋆).

- `Setup.hs`, `package.yaml`, and `stack.yaml` are project files used by Stack; you should not edit these.
- `app/Main.hs` is the main entry point of the program. It contains the user input loop.
- `src/CommandParser.hs` contains the parser for user commands.
- `src/Error.hs` contains code for reporting and displaying errors.
- `src/Deck.hs` (⋆) contains the data structures for representing cards and the deck. You will need to edit this code to implement deck creation and shuffling.
- `src/Game.hs` (⋆) contains the game logic. You will need to edit this to implement the game logic.

You are welcome (and encouraged) to add any helper functions you like, and you are also welcome to use any library functions that do not require additional dependencies. **However, please do not change any of the supplied data structures or type signatures.**

Running the skeleton code. To run the skeleton code, from the `solitaire` directory, run `stack run`. If you are on Windows, please first run the `unicode_setup.ps1` script from a PowerShell window (if you get an error about execution policies, just copy the contents of the file into PowerShell).

You can also use `stack ghci` and import the various files from the interactive environment (e.g., to interactively work with `Game.hs`, run `stack ghci` and then type `:l src/Game.hs`).

Part 1: Deck Creation and Shuffling (`Deck.hs`)

In the first part of the assignment you will create a deck and implement a shuffling algorithm.

Exercise 1 (Deck Creation).
Implement the function `deckOf52 :: [Card]` *to create a deck that contains one of each card in the standard deck of 52; the order does not matter. Specifically the deck should contain, for each suit, the three face cards (Jack, Queen, King) and all of the cards from Ace to 10.*

Hint*: The* `Value` *type implements the* `Bounded` *and* `Enum` *typeclasses, so you can use sequence notation (e.g.* `[Ace .. King]`*).*

[4 marks]

The next task is to implement shuffling using the *Fisher-Yates shuffle algorithm*. The Fisher-Yates shuffle works by swapping each element in the sequence for a randomly-determined other element. It can be described by the following pseudocode, taken from the Wikipedia page:

```
-- To shuffle an array a of n elements (indices 0..n-1):
for i from 0 to n - 2 do
     j <- random integer such that i <= j < n
     exchange a[i] and a[j]
```

Remember from the lectures and lab sheets that random number generation is pure, after initially seeding the random number generator with some entropy. The `System.Random` library implements a random number generator; you are likely to find the `uniformR` function useful. We have implemented a function `shuffleDeck :: IO Deck` that seeds the RNG and calls `shuffle` for you, and this is called from the `main` function.

Exercise 2 (Shuffling).
Implement the `shuffle :: StdGen -> Deck -> Deck` *function using the Fisher-Yates shuffle. It will help to define a helper function to swap two elements in a list.*

[6 marks]

Part 2: Game Logic (`Game.hs`)

Board Representation

The game state is represented by the `Board` data type found in `Game.hs`, which also makes use of the `Pillars` data type and the `Column` type alias:

```
data Pillars = MkPillars {
    spades :: Maybe Value,
    clubs :: Maybe Value,
    hearts :: Maybe Value,
    diamonds :: Maybe Value
} deriving (Eq)
```

```
data Board = MkBoard {
    boardDeck :: [Card],
    boardDiscard :: [Card],
    boardPillars :: Pillars,
    boardColumns :: [Column]
} deriving (Eq)
```

```
type Column = [(Card, Bool)]
```

The `Pillars` data type is a record mapping each suit to a `Maybe Value` representing the top card on that pillar; this representation is chosen such that we don't need to worry about invalid states such as pillars containing cards with different suits, or out-of-order cards.

`Column` is a type alias for a list of `(Card, Bool)` pairs, where the `Bool` is `True` if that card is visible, and `False` otherwise. The head of the list is the most recently played card.

The `Board` data type itself is a record with four fields:

- `boardDeck` is a list of cards representing the deck. The head of the list represents the top card of the deck.

```
Deck size: 7
Discard:  ♠5,  ♠6
Pillars:
  Spades: ♠2
  Clubs: <empty>
  Hearts: ♥2
  Diamonds: ♦2

[0] [1] [2] [3] [4] [5] [6]
 ♦K  ♥K ??? ??? ??? ??? ???
 ♣Q  ♠Q ??? ??? ???  ♦9 ???
 ♥J  ♦J  ♠K  ♦6 ???     ???
♠10      ♥Q  ♣5  ♥6      ♦5
 ♥9      ♣J  ♥4
 ♣8     ♦10
 ♥7      ♣9
 ♣6
 ♥5
 ♣4
 ♦3
 ♣2
```

(a) Example board

```
MkBoard {
    boardDeck = [ ♠4, ♠8, ♠7, ♥8, ♥3, ♦8, ♠J],
    boardDiscard = [ ♠6, ♠5],
    boardPillars = MkPillars {
        spades = Just 2,
        clubs = Nothing,
        hearts = Just 2,
        diamonds = Just 2
    },
    boardColumns = [
        [(♣2,True),(♦3,True),(♣4,True),
         (♥5,True),(♣6,True),(♥7,True),
         (♣8,True),(♥9,True),(♠10,True),
         (♥J,True),(♣Q,True),(♦K,True)],
        [(♦J,True),(♠Q,True),(♥K,True)],
        [(♣9,True),(♦10,True),(♣J,True),
         (♥Q,True),(♠K,True),(♣3,False),
         (♣A,False)],
        [(♥4,True),(♣5,True),(♦6,True),
         (♥10,False),(♦Q,False)],
        [(♥6,True),(♣7,False),(♣10,False),
         (♣K,False)],
        [(♦9,True),(♠3,False)],
        [(♦5,True),(♦7,False),(♦4,False),
         (♠9,False)]
    ]
}
```

(b) Board representation

Figure 2: Example board and internal representation

- `boardDiscard` is a list of cards representing the discard pile. The head of the list represents the top card of the discard pile.
- `boardPillars` is the representation of the pillars.
- `boardColumns` is a list of columns, ordered from left to right.

As an example, Figure 2a shows an example game, and Figure 2b shows its internal representation.

Invariants. The `Board` data type expects the following invariants to hold:

- The length of `boardColumns` is always precisely 7. An empty column is represented by an empty list.
- Each non-empty column must contain 0 or more non-visible cards, followed by 1 or more visible cards. Consequently this means that the top card of a column must always be visible.
- The *visible* cards in each column must be arranged in descending order with alternating colours. Because our internal representation treats each column as a stack (with the most-recently placed card at the head of the list), the list representing each column should therefore be in *ascending* order.
- If we were to expand out all of the pillars (i.e., expanding the `clubs = Just Three` pillar to `[♣3, ♣2, ♣1]`), then concatenating the expansions of the pillars, the discard pile, the deck, and the contents of each of the columns would give (a permutation of) the deck of 52. Specifically, this means that the total number of cards in play never changes.

Board Display

The next task is to write a `Show` instance to display the board. The `show` function for the board should:

- Print “Deck size: n” (where n is the size of the deck).
- Print “Discard: ” along with the three most recent cards, with the most recent at the end of the list.
- Print “Pillars: ”, and then on separate lines print each suit and the card on the top of the pillar. If a pillar is empty, print `<empty>`.
- Print each column side-by-side. Each column should first display its index (e.g., `[0]`), followed by the least-recent card down to the most-recent card at the bottom of the stack. Every non-visible card should be represented by `???`.

As a concrete example, see Figure 2.

We have implemented a `Show` instance for the `Card` data type for you, which will handle the colouring of red suits when printed to the console, and ensures that the representation of each card has exactly three characters.

Note: Don’t worry too much about exact capitalisation and spacing. We aren’t automarking this and you won’t be penalised for minor differences.

Exercise 3 (Show instance)**.** *Write a* `Show` *instance for the* `Board` *data type.*

[10 marks]

Initial Setup and Win Checking

Setup The next task is to handle the initial board setup, given a shuffled deck.

To set up the board, we need to deal each column in turn. The first column should have a single card face-up; the second column should have one card face-down and one face-up; the third should have two cards face-down and one face-up, and so on until we have filled 7 columns. Cards are dealt left-to-right (starting with the leftmost column), and top-to-bottom on each column.

As an example, setting up the board with an un-shuffled deck (that orders suits as Spades, then Clubs, then Diamonds, then Hearts) would result in the following columns:

```
[0] [1] [2] [3] [4] [5] [6]
 ♠A ??? ??? ??? ??? ??? ???
     ♠3 ??? ??? ??? ??? ???
         ♠6 ??? ??? ??? ???
            ♠10 ??? ??? ???
                 ♣2 ??? ???
                     ♣8 ???
                         ♦2
```

All pillars should be initialised to be empty (i.e., set to `Nothing`), the discard pile should be empty, and the deck should consist of all cards not used in the initial board setup.

Exercise 4 (Setup)**.** *Implement the* `setup :: Deck -> Board` *function.*

[10 marks]

Win Checking The game is won when every pillar contains a King. Consequently this will mean that there are no other cards on the board.

Exercise 5 (Win checking)**.** *Implement the* `isWon :: Board -> Bool` *function that returns* `True` *if the top card on each pillar is a King.*

[2 marks]

Commands

The next part of the assignment involves implementing the game logic. Since the implementation is text-based, playing the game involves using text-based commands, as summarised in the following table:

Long command	**Short command**	`Command` **data constructor**	**Description**
`draw`	`d`	`Draw`	Draws a card from the deck
`move` n `from` s_1 `to` s_2	`mv` n s_1 s_2	`Move Count StackIndex StackIndex`	Moves n cards from stack s_1 to stack s_2
`movest` s_1 s_2	—	`MoveStack FromStack ToStack`	Moves the visible contents of stack s_1 to stack s_2
`movefd` s	—	`MoveFromDiscard StackIndex`	Moves the card from the top of the discard stack to stack s
`movetp discard`	`movetp d`	`MoveToPillar FromDiscard`	Moves a card from the discard pile to the pillars
`movetp` s	—	`MoveToPillar (FromStack StackIndex)`	Moves a card from stack s to the pillars
`movefp` st s	—	`MoveFromPillar Suit StackIndex`	Moves the card at the top of the pillar for suit st to the top of stack s (e.g., `movefp clubs 2`)
`solve`	—	`Solve`	Attempts to repeatedly move the top of each stack to the pillars, until no more cards can be moved.

You do not need to worry about parsing these in; this is done by `CommandParser.hs`.

Helper functions. It helps to first define some simple helper functions that will be useful when implementing the commands.

- `flipCards :: Board -> Board`, which makes the top card on each column visible. The skeleton code calls this after each move.
- `updateColumn :: Int -> Column -> [Column] -> [Column]`, which given an index `idx`, a column `c`, and a list of columns `cs`, returns `cs` replacing the column at `idx` with `c`.
- `canStack :: Card -> Card -> Bool` which, given two cards `c1` and `c2`, returns `True` if we can stack `c1` onto `c2` (i.e., that `c1` is the opposite colour to `c2`, and is one less in value).
- `canStackOnPillar :: Card -> Maybe Value -> Bool`, which returns `True` if the given card `c` can be put on the given pillar `p`. More specifically:
  - Returns `True` if `p = Nothing` and the value of `c` is `Ace`.
  - Returns `True` if `p = Just c2`, and the value of `c` is precisely 1 more than `c2`.
  - Returns `False` otherwise.

Exercise 6 (Helper functions). *Implement the following functions based on the above descriptions:*

- `flipCards :: Board -> Board`
- `updateColumn :: Int -> Column -> [Column] -> [Column]`
- `canStack :: Card -> Card -> Bool`
- `canStackOnPillar :: Card -> Maybe Value -> Bool`

[8 marks (2 for each function)]

Draw Drawing from the deck involves placing the top card from the deck onto the discard pile.

In the case that the deck is empty but the discard pile has cards in it, the discard pile is reversed and treated as the new deck, and then a card is drawn from the new deck. In the case that both the deck and the discard pile are empty, then drawing is not possible and the function should return `Left DeckEmpty`.

Exercise 7 (Draw). *Implement the* `draw :: Board -> Either Error Board` *function.*

[2 marks]

Move The command `move` n `from` s_1 `to` s_2 command attempts to move n cards from the bottom of column s_1 to the bottom of s_2. For example, consider the following board where we are moving 2 cards from column 2 to column 6:

```
 [0] [1] [2] [3] [4] [5] [6]
  ♥K  ♦K  ♠K  ♣K         ???
  ♣Q  ♠Q  ♦Q             ???
  ♦J  ♥J  ♣J              ♠J
 ♣10 ♠10 ♥10             ♦10
  ♥9  ♦9  ♠9              ♣9
  ♣8  ♠8  ♥8              ♦8
  ♥7      ♠7              ♣7
  ♣6      ♥6
  ♥5      ♣5
  ♣4
```

$\xrightarrow{\texttt{move 2 from 2 to 6}}$

```
 [0] [1] [2] [3] [4] [5] [6]
  ♥K  ♦K  ♠K  ♣K         ???
  ♣Q  ♠Q  ♦Q             ???
  ♦J  ♥J  ♣J              ♠J
 ♣10 ♠10 ♥10             ♦10
  ♥9  ♦9  ♠9              ♣9
  ♣8  ♠8  ♥8              ♦8
  ♥7      ♠7              ♣7
  ♣6                      ♥6
  ♥5                      ♣5
  ♣4
```

This is safe because the most recently-played card in column 6 is the 7 of clubs, and the 2nd card (read bottom-up) from stack 2 is the 6 of hearts which is the opposite colour and has a value of one less.

It is only possible to move cards to an empty pillar if the top card in the column is a King.

There are several ways that the `move` command can fail:

- If the function is called with an invalid number of cards (i.e., less than 1), then the function should return `Left InvalidCount`.
- If n is greater than the number of visible cards in column s_1, then the function should return `Left MovingTooManyCards`.
- If s_2 is empty and the top card in s_1 is not a King, then the function should return `Left ColumnKing`.
- If the nth card from the bottom of s_1 cannot be placed safely on the bottom of s_2 (e.g., because the colour is not opposite, or the ordering would be violated), then the function should return `Left WrongOrder`.

Exercise 8 (Move). *Implement the*
`move :: Int -> Int -> Int -> Board -> Either Error Board` *function.*

[6 marks]

Move stack Sometimes it is useful to move all visible cards on a stack onto another stack. This follows the same rules as `move` (i.e., the top card of the stack being moved must be the opposite colour to, and with a value of one less than, the bottom card of the target stack). For example:

```
[0] [1] [2] [3] [4] [5] [6]
 ♦9 ???  ♠K ??? ??? ??? ???
 ♠8  ♣6  ♦Q ??? ??? ??? ???
 ♦7  ♦5     ???  ♠3 ??? ???
             ♠9      ♥2 ???
             ♥8         ???
             ♠7          ♦4
             ♥6
```

```
movest 1 0
──────────→
```

```
[0] [1] [2] [3] [4] [5] [6]
 ♦9  ♥J  ♠K ??? ??? ??? ???
 ♠8      ♦Q ??? ??? ??? ???
 ♦7         ???  ♠3 ??? ???
 ♣6          ♠9      ♥2 ???
 ♦5          ♥8         ???
             ♠7          ♦4
             ♥6
```

Exercise 9 (Move stack). *Implement the* `moveStack :: Int -> Int -> Board -> Either Error Board` *function. We recommend implementing this by calling your existing* `move` *function.*

[2 marks]

Move from discard The `movefd` command moves a card from the top of the discard pile to the given column, provided that doing so would maintain the board invariants (i.e., that the card is the opposite colour and one to the most-recently played card in that column). For example, in the following game we are moving the five of clubs from the discard pile to column 5:

```
Deck size: 20
Discard: ♣10,  ♥K,  ♣5
Pillars:
  Spades: <empty>
  Clubs: <empty>
  Hearts: ♥A
  Diamonds: <empty>

[0] [1] [2] [3] [4] [5] [6]
 ♠Q ??? ??? ??? ??? ??? ???
     ♥5  ♠9 ??? ??? ??? ???
     ♣4     ??? ??? ??? ???
             ♠3 ??? ??? ???
                 ♣K ??? ???
                     ♦6 ???
                         ♥6
```

```
movefd 5
────────→
```

```
Deck size: 20
Discard: ♣10,  ♥K
Pillars:
  Spades: <empty>
  Clubs: <empty>
  Hearts: ♥A
  Diamonds: <empty>

[0] [1] [2] [3] [4] [5] [6]
 ♠Q ??? ??? ??? ??? ??? ???
     ♥5  ♠9 ??? ??? ??? ???
     ♣4     ??? ??? ??? ???
             ♠3 ??? ??? ???
                 ♣K ??? ???
                     ♦6 ???
                     ♣5  ♥6
```

There are several possible failure conditions:

- If the discard pile is empty, then the function should return `Left DiscardEmpty`.
- Only a King can be moved to an empty column; if the user attempts to move a different card to an empty column then the function should return `Left ColumnKing`.
- If the card cannot be moved to the given column because it would violate colour alternation or card ordering then the function should return `Left WrongOrder`.

Exercise 10 (Move from discard). *Implement the* `moveFromDiscard :: Int -> Board -> Either Error Board` *function.*

[3 marks]

Move to pillar The `movetp` command moves a card to the pillars. There are two ways we can move a card to the pillar: either from the discard pile directly, or from a column. For this to be possible, it must be the case that we are either placing an Ace on an empty pillar, or we are placing a card whose value is one more than the current card on the pillar.

For example, in the following we move the top card on the discard pile (the ace of clubs) to the pillars:

```
Deck size: 21
Discard:  ♦6,  ♦K,  ♣A
Pillars:
  Spades: <empty>
  Clubs: <empty>
  Hearts: <empty>
  Diamonds: <empty>


[0] [1] [2] [3] [4] [5] [6]
 ♦9 ??? ??? ??? ??? ??? ???
     ♣6  ♦Q ??? ??? ??? ???
            ??? ??? ??? ???
             ♠9 ???  ♥2 ???
             ♥8  ♣2     ???
             ♠7         ???
             ♥6          ♠8
```

$\xrightarrow{\texttt{movetp discard}}$

```
Deck size: 21
Discard:  ♦6,  ♦K
Pillars:
  Spades: <empty>
  Clubs: ♣A
  Hearts: <empty>
  Diamonds: <empty>


[0] [1] [2] [3] [4] [5] [6]
 ♦9 ??? ??? ??? ??? ??? ???
     ♣6  ♦Q ??? ??? ??? ???
            ??? ??? ??? ???
             ♠9 ???  ♥2 ???
             ♥8  ♣2     ???
             ♠7         ???
             ♥6          ♠8
```

Afterwards, we can then move the two of clubs from column 4 to the club pillar:

```
Deck size: 21
Discard:  ♦6,  ♦K
Pillars:
  Spades: <empty>
  Clubs: ♣A
  Hearts: <empty>
  Diamonds: <empty>


[0] [1] [2] [3] [4] [5] [6]
 ♦9 ??? ??? ??? ??? ??? ???
     ♣6  ♦Q ??? ??? ??? ???
            ??? ??? ??? ???
             ♠9 ???  ♥2 ???
             ♥8  ♣2     ???
             ♠7         ???
             ♥6          ♠8
```

$\xrightarrow{\texttt{movetp 4}}$

```
Deck size: 21
Discard:  ♦6,  ♦K
Pillars:
  Spades: <empty>
  Clubs: ♣2
  Hearts: <empty>
  Diamonds: <empty>


[0] [1] [2] [3] [4] [5] [6]
 ♦9 ??? ??? ??? ??? ??? ???
     ♣6  ♦Q ??? ??? ??? ???
            ??? ??? ??? ???
             ♠9  ♠K  ♥2 ???
             ♥8         ???
             ♠7         ???
             ♥6          ♠8
```

There are again several failure conditions:

- If the card source is the discard pile, and the discard pile is empty, then the function should return `Left DiscardEmpty`.
- If the card source is a column, and the column is empty, then the function should return `Left ColumnEmpty`.
- Attempting an invalid move should result in `Left WrongPillarOrder`.

Exercise 11 (Move to pillar). *Implement the* `moveToPillar :: CardSource -> Board -> Either Error Board` *function.*

[4 marks]

Move from pillar The `movefp` st s command moves the card at the top of the pillar for suit st (where $st \in \{\texttt{spades}, \texttt{clubs}, \texttt{hearts}, \texttt{diamonds}\}$) onto column s, as long as such a move will preserve the alternation of colours and descending ordering.

```
Deck size: 16
Discard:  ♥8,  ♣6,  ♣5
Pillars:
  Spades: <empty>
  Clubs: ♣2
  Hearts: ♥A
  Diamonds: ♦2


[0] [1] [2] [3] [4] [5] [6]
     ♦K  ♣K ??? ??? ??? ???
     ♣Q  ♦Q ??? ??? ??? ???
     ♥J     ???  ♥6 ???  ♠2
             ♦9  ♠5 ???
             ♠8      ♠3
             ♦7
             ♠6
             ♦5
             ♠4
```

$\xrightarrow{\texttt{movefp hearts 6}}$

```
Deck size: 16
Discard:  ♥8,  ♣6,  ♣5
Pillars:
  Spades: <empty>
  Clubs: ♣2
  Hearts: <empty>
  Diamonds: ♦2


[0] [1] [2] [3] [4] [5] [6]
     ♦K  ♣K ??? ??? ??? ???
     ♣Q  ♦Q ??? ??? ??? ???
     ♥J     ???  ♥6 ???  ♠2
             ♦9  ♠5 ???  ♥A
             ♠8      ♠3
             ♦7
             ♠6
             ♦5
             ♠4
```

The function should return `Left PillarEmpty` if the pillar is empty, `Left ColumnKing` if trying to move a non-King card from the pillar to an empty column, or `Left WrongOrder` if moving the card from the pillar would violate the card ordering.

Exercise 12 (Move from pillar). *Implement the* `moveFromPillar :: Suit -> Int -> Board -> Either Error Board` *function.*

[3 marks]

Solve Often there may be multiple cards that can be moved to the pillars. The `solve` command repeatedly tries to move the top card of each column to the pillaars, until no more moves are possible.

For example, using `solve` on the following board would win the game since it would begin by moving ♣2 to the pillars, followed by ♦3, followed by ♠5, and so on, until all kings are placed on the pile. (In general, the game is solvable once all covered cards become visible).

```
Deck size: 0
Discard:
Pillars:
  Spades: ♠4
  Clubs: ♣A
  Hearts: ♥3
  Diamonds: ♦2


[0] [1] [2] [3] [4] [5] [6]
 ♥K ♣10  ♣K  ♦K  ♥4  ♣6  ♠K
 ♠Q  ♦9  ♦Q  ♣Q      ♥5  ♥Q
 ♥J  ♣8  ♣J  ♦J          ♠J
     ♥7 ♦10 ♠10         ♥10
     ♠6  ♠9  ♥9          ♣9
     ♦5  ♥8  ♠8          ♦8
     ♣4  ♠7  ♦7          ♣7
     ♦3  ♦6              ♥6
     ♣2  ♠5              ♣5
                         ♦4
                         ♣3


Input command >> solve
Congratulations, you win!
```

The `solve` function can be called even when the board is not immediately solvable; it just stops once no more moves can be made.

Exercise 13 (Solve)**.** *Implement the* `solve :: Board -> Board` *function.*

[6 marks]

Changelog

- v1 (8th November 2024): Initial Release
- v2 (12th November 2024): Clarified that `Left ColumnKing` should be returned when attempting to move a non-King from a pillar to an empty column in Exercise 12
- v3 (15th November 2024): Fixed typo with `canStackOnPillar` description, which should require cards to be stacked in *ascending* order