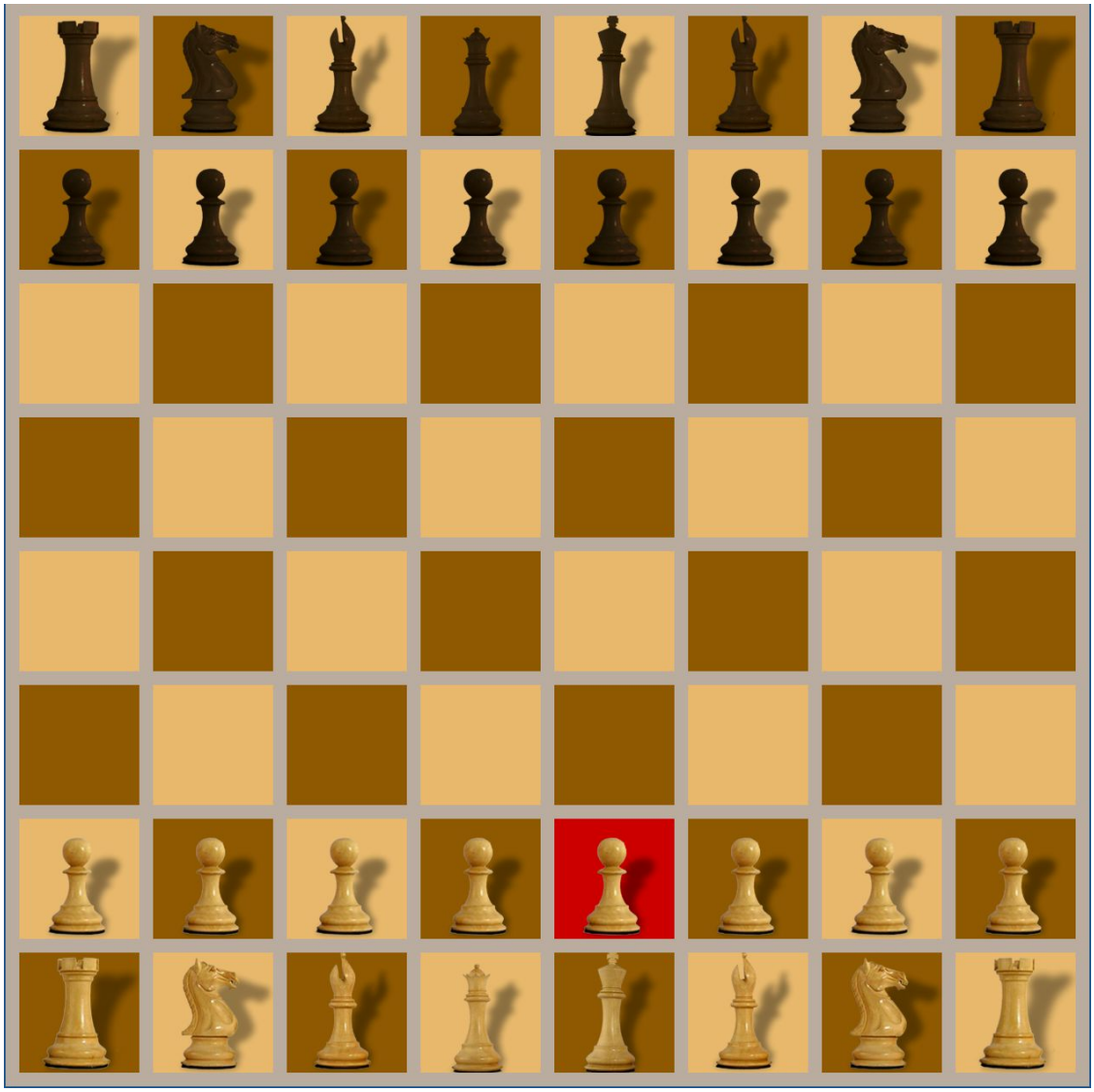
Programación Funcional

Trabajo Práctico Final

Functional Chess!



Alumno

Francisco Depascuali 53080

Profesores:

- Martinez Lopez,Pablo Ernesto
- Pennella,Valeria Verónica

Índice

Introducción

Este trabajo práctico tiene como objetivo implementar el juego Ajedrez en un lenguaje funcional.
La elección del trabajo fue completamente arbitraria: en primer lugar, que sea un juego de mesa fue porque lo consideré divertido y en segundo lugar, el ajedrez es un juego que siempre me gusto.

Lenguaje utilizado

El lenguaje de programación utilizado fue Elm, que es de tipo Functional-Reactive. La decisión se basó en que la sintaxis es muy similar a la de Haskell, y también porque ofrece un sistema de señales que permite modelar side effects.

Modos de juego

Se puede jugar 1 vs 1, contra la computadora o computadora vs computadora.

Instrucciones de uso

El ajedrez se puede jugar online en el siguiente link: http://functionalchess.herokuapp.com/ . Allí están todas las instrucciones de uso.

La implementación se encuentra en github:
https://github.com/FranDepascuali/FunctionalChess

Inteligencia Artificial

En principio, había tomado la decisión de implementar una inteligencia robusta para la computadora. La idea era tener un puerto en el cual se codificaba y mandaba el tablero actual y tener otro programa corriendo, escrito en otro lenguaje por razones de eficiencia (por ejemplo C), que procese la información y devuelva la jugada a realizar. Para determinar la jugada, utilizaría el algoritmo minimax.

Las razones por las cual no fue llevado a cabo fueron:

1. La complejidad era mucho mayor de la que había previsto. Por ejemplo, para aplicar el algoritmo MiniMax, uno debe poder asignarle un puntaje a cada uno de los posibles tableros. Ahora bien, esto implica determinar el puntaje de cada ficha. El problema es que el puntaje de cada ficha depende de varias variables, por ejemplo la cantidad de fichas actuales en el tablero o la posición en la que están. Al investigar sobre cómo se asigna puntajes a las fichas en otros programas de Ajedrez, encontré que no hay una forma definida, sino que hay cierto grado de subjetividad.

2. Personalmente, me interesa la inteligencia artificial en forma de redes neuronales, algoritmos genéticos o agentes independientes. Aplicar esas técnicas para el ajedrez se vuelve particularmente difícil, desde decidir la estructura ideal de las redes neuronales hasta determinar que es una mutación para un ajedrez en algoritmos genéticos, por lo cual llevaría un tiempo y una dificultad considerable.

Por estos motivos, se utilizó la estrategia random, siempre en cuando el movimiento no deje descubierto al rey.

Arquitectura

En el ajedrez hay dos tipos de colores: blanco y negro y 6 tipos de fichas: Rey, Reina, Alfil, Caballo, Torre, Peón.

```
module Piece where

type PieceColor = White | Black
type PieceType = Pawn | Knight | Bishop | Rook | Queen | King
type Piece = Piece PieceColor PieceType
...
```

Me pareció correcto representar una casilla del tablero, que puede contener o no una pieza.

```
module Tile where

type alias Tile = Maybe Piece
...
```

Una vez definida la casilla del tablero, se determinó como representar al tablero.

```
module Board where

type alias Board = Array (Array Tile)
...
```

Al tener el tablero, el próximo paso fue definir la representación del estado del juego.

```
module GameModel where

type Progress = InProgress | WhiteWon | BlackWon
type alias PlayerColor = PieceColor
type GameType = OneVSOne | OneVSComputer

type alias GameState = {
    board: Board,
    gameProgress: Progress,
    turn: PlayerColor,
    selected: Maybe Position,
    cursorAt: Position,
    gameType: GameType,
```

```
    mayhem: Bool,
    seed: Maybe Seed
}
...
```

El estado de juego en un momento particular puede ser definido por:

- El tablero
- El progreso del juego (Si algún jugador ganó o si continúa el juego)
- El turno, que indica si debe mover el jugador blanco o negro.
- La posición de la pieza seleccionada, si la hubiere.
- Donde se encuentra el cursor (el que se mueve con el teclado o aswd)
- El tipo de juego: 1 vs 1 ó 1 vs computadora
- El estado mayhem (“caos” en español), que cuando se encuentra activado, la computadora juega contra sí misma.
- Una semilla, que utilizamos para generar números aleatorios.

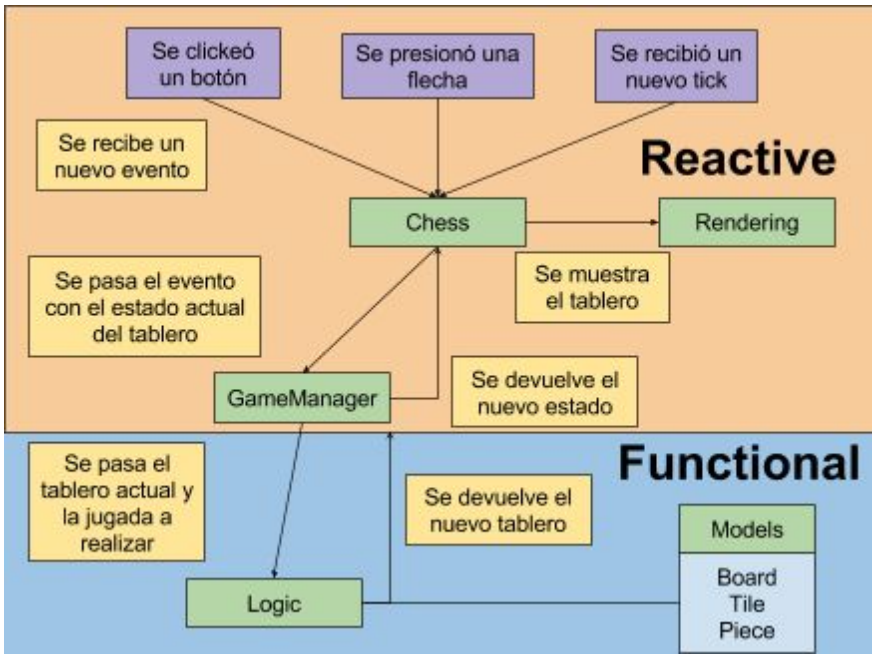
Resta definir la interacción del usuario con el juego.

```
module InputModel where

import Signal exposing (..)
import Keyboard

type Direction = Up | Down | Left | Right | None

type Update = Move Direction | NewGameButtonPressed Bool | GameTypeChanged Bool |
EnterPressed Bool | MayhemActivated Bool | Tick Float

type alias Input = { -- inputs that the game will depend of
    action: Update, -- the user actions
    currentTimestamp: Int -- the current time
}
...
```

Aquí tenemos:

1) El tipo `Direction`, que se refiere a que lugar se mueve el cursor
2) El tipo `Update` engloba los posibles eventos que ocurren en la aplicación
    a) `Move Direction`: Se presionó alguna flecha o aswd.
    b) `EnterPressed Bool`: Se presionó el enter.
    c) `NewGameButtonPressed Bool`, `GameTypeChanged Bool`, `MayhemActivated Bool`: Se refieren a si se presiono sus botones correspondientes.
    d) `Tick Float`: Para poder simular el estado mayhem, en el cual la computadora juega contra sí mismo, mandamos un evento cada segundo para que se actualice el juego.
3) El tipo `Input`, en donde se almacena el evento que se recibió y que contiene el momento en que comenzó el juego `currentTimestamp`, utilizado para el random.

Para dar una mayor idea de la arquitectura, visualmente se representa de la siguiente manera:



Reactive: La capa Reactive es donde se maneja el estado, es decir, donde se reciben los nuevos eventos y donde se encarga de desplegar la UI.

Functional: La capa Functional dispone de los algoritmos para determinar si una jugada puede realizarse. Esto permite una mejor forma de testear (dado que todo en la capa functional se puede testear) y también permite mantener la transparencia referencial y evitar side effects, por lo tanto manteniendo todas las funciones puras.

En la imagen se pueden apreciar los otros módulos de los cuales dispone la aplicación:

- **Chess**: El módulo principal. Aquí se definen los puertos (para comunicarse con el exterior) y se realiza un fold sobre el estado del juego y el input recibido.
- **GameManager**: Maneja los estados del juego, aquí es donde se recibe un estado y se devuelve uno nuevo, dependiendo de las teclas que ingresó el usuario y del estado del juego.
- **Logic**: Se encarga de realizar el movimiento y devolver el nuevo tablero, si es que el movimiento es válido.
- **Rendering**: Declara el estilo y cómo se va a representar visualmente el ajedrez.
- **Utils**: Funciones útiles utilizadas por los otros módulos.

Implementación

El programa comienza con

```
-- Folds the input into the gameState, starting with the defaultGame.
gameState: Signal GameState
gameState = foldp checkValidGame defaultGame input
```

que aplica un fold sobre `GameState`, recibiendo el estado inicial y el input. Llama a `checkValidGame` que pertenece al módulo GameManager.

En GameManager se recibe el estado del juego y el input y revisa si el juego terminó o no. Si el juego no terminó, realiza un movimiento si es la computadora o se fija si el movimiento es válido si es un jugador.

Juega algún jugador
En este caso, se llama a la función `canMakeMove` del módulo Logic, que decide si el movimiento es correcto o no. Si es válido, se realiza el movimiento.

Juega la computadora
Es más complicado que el caso anterior. Como ya se aclaró antes, la inteligencia es jugar al azar (siempre y cuando no se deje en jaque al rey, mientras sea posible).
El primer approach fue llamar recursivamente a una función que seleccionara una pieza al azar, un movimiento al azar para esa pieza. Si el movimiento era válido, entonces se realizaba esa jugada, sino seguia llamandose recursivamente a la función.
El problema de esta implementación era que si se estaba en jaque mate, loopeaba hasta infinito dado que no existía una pieza que pudiera moverse para la cual el rey no quedara desprotegido. Por lo tanto, se realizó de la siguiente manera:

1. Se implementó una función `shuffle`, que dada una lista, devuelve la lista desordenada.
2. Se obtienen una lista de todas las piezas del color de la computadora que estuviera jugando en ese momento.
3. Se le aplica la función `shuffle` a esa lista
4. Se obtiene la primer pieza de la lista.
5. Se genera la lista de posibles movimientos para esa pieza.
6. Se aplica shuffle sobre esa lista de movimientos
7. Se fija uno por uno si algún movimiento es válido. Si es así, lo retorna, si no existe movimiento válido para esa pieza sigue con la próxima.
8. Si no existe movimiento posible habiendo recorrido todas las piezas y todos los movimientos, entonces retorna el juego finalizado, con el color de la computadora en ese momento como perdedor.

Juegue algún jugador o juegue la computadora, en ambos casos utilizamos la función principal del programa: `canMakeMove`, del módulo de Logic.
La definición de la función es la siguiente:

```
canMakeMove: Board -> PieceColor -> Piece -> Position -> Position -> Bool
canMakeMove board color piece from to =
    let
      tileTo = readTile to board
    in
      pieceColorMatches color piece &&
      not (colorMatches color tileTo) &&
      pieceCanReachPosition board piece from to &&
      let
        newBoard = makeMove board from to
      in
        notInCheck newBoard color
```

Lo que hace esta función es lo siguiente:

1. Lee la casilla a la cual se quiere mover en `tileTo`
2. Verifica los siguientes requisitos:
    a. `pieceColorMatches`: El color de la pieza que se quiere mover debe de ser el mismo que el del jugador.
    b. `not (colorMatches color tileTo)`: No debe haber una pieza del mismo color en la casilla a la cual se quiere mover.
    c. `pieceCanReachPosition`: Decide si una función puede llegar a una posición. Se explicará con más detalle a continuación.
3. Si todas estas reglas se cumplen, entonces falta verificar que al realizar el movimiento, el rey no quede en jaque. Ésta es, en mi opinión, una de las diferencias más importantes de la versión imperativa a la versión funcional: En una versión imperativa, elegiría otro approach, cómo verificar con el propio tablero si se puede realizar el movimiento. En cambio, con esta versión correspondiente al paradigma funcional, se realiza el movimiento, obteniendo un nuevo tablero y se verifica que no se encuentre en jaque.

Otra función importante recién mencionada es `pieceCanReachPosition`. La función se define de la siguiente manera:

```
pieceCanReachPosition: Board -> Piece -> Position -> Position -> Bool
pieceCanReachPosition board piece from to = notOtherPieceInPath board piece from to &&
(moveIsValid board piece from to || canAttack board piece from to)
```

Primero, se fija que no exista otra pieza en el camino. Si este es el caso, entonces verifica que: el movimiento sea válido ó que la pieza pueda atacar a la posición. Ésta última la utilizamos solamente para el peón, porque el ataque no es igual a su sentido de movimiento (se mueve verticalmente pero ataca diagonalmente). Para todas las otras piezas moverse o atacar es lo mismo.

La decisión de implementar `pieceCanReachPosition` de esa manera nos permite utilizarla en `canMakeMove` (explicado anteriormente) y también en la función `inCheck`, que se fija si un jugador determinado se encuentra en jaque.

```
inCheck: Board -> PieceColor -> Bool
inCheck board color = let
```

```
                        pieces = getPiecesFromBoard board
                        otherColorPieces = getPiecesForColor board (switchColor color)
                        king = head (filter (\(t, x, y) -> t == Piece color King) pieces)
                    in
                        case king of
                            Nothing -> False -- Should not enter here, but king is Maybe.
                            Just kingPosition -> any (\(piece, x, y) -> pieceCanReachPosition
board piece (x,y) (second kingPosition, thrd kingPosition)) otherColorPieces
```

1) Se obtiene una lista con las piezas del tablero.
2) Se obtiene una lista con las piezas del color contrario del tablero.
3) Se obtiene el rey del color que se recibe por parámetro.
4) Se fija si existe alguna de las piezas del color contrario que pueda llegar al rey. Si esto es así, quiere decir que está en jaque.

Hasta aquí se explicaron las funciones más importantes, las otras se encuentran disponibles en el anexo y en el código.

Aclaraciones

- Se respetó que cada módulo importe solamente lo necesario. Por ejemplo, el módulo Chess:

  ```
  module Chess where

  import InputModel exposing (..)
  import GameModel exposing (GameState, defaultGame)
  import GameManager exposing (checkValidGame)

  import Signal exposing (..)
  import Rendering exposing (display)
  import Time exposing (every, second)
  import Date exposing (year, hour, minute, second, fromTime)
  ...
  ```

  Todos los módulos siguen ese estilo: Primero, los imports correspondientes a los módulos implementados que requiere. Por ejemplo, el módulo chess solo conoce la función `checkValidGame` de `GameManager`, no debe por ningún motivo conocer el módulo correspondiente a la lógica. Luego, se deja un espacio y se declaran los módulos importados de las librerias de Elm. Cuando se utiliza `exposing (..)` significa que se importa el módulo completo, sino se decide explícitamente que importar (por ejemplo con el `checkValidGame`).

- Para obtener un nuevo número aleatorio, se utiliza el Seed que contiene el GameState, el cual nos da un número al azar y un nuevo Seed, que pasará a ser el Seed del nuevo GameState. Se utiliza la hora actual como Seed inicial.

- En cuanto a la implementación, no puedo determinar si los modelos que utilice son los mejores. En particular, la duda que me surgió es si una casilla, que actualmente se define `type alias Tile = Maybe Piece` debería o no contener la posición de la pieza, por ejemplo `type alias Tile = Maybe Piece Position`. En un principio consideré

que no, dado que me pareció que los modelos deberían de ser inertes, por lo cual indicar una posición implica que implícitamente sepan del estado. Ahora bien, cuando uno habla de una casilla, piensa a la posición como un elemento intrínseco de ésta. Me pareció más fuerte el primer aspecto, por lo que dejé esa implementación. Cabe mencionar que haber utilizado la segunda hubiera hecho más fácil los cálculos, ya que en muchos lugares se necesita saber la posición de la pieza.

- Por otro lado, la decisión de utilizar un Array para representar el tablero. En principio, el tablero era de tipo List y después se cambió a un Array. La decisión se determinó por dos motivos:
    1. Un tablero es semánticamente mejor representado por un Array que por una List, dado que tenemos un número de filas y columnas.
    2. En principio, la versión de Elm no contenía Array, pero luego lo incluyó en la próxima versión. Investigué sobre como lo implementaron[1] y me pareció muy interesante, por lo que también me llevó a utilizarlo.

Bugs

Ningún programa está exento de bugs, y Functional Chess no es la excepción. Tuve dos muy importantes: Uno de ellos fue que el programa compilaba, pero fallaba en ejecución, con un error de tipos, algo del estilo “error de tipo: se llamó a length sobre una estructura desconocida”. Luego de mucho tiempo, me di cuenta que el error era de Elm (dado que es un error que debería de captar el compilador) y no de mi programa. Lo que hice fue escribir la función de nuevo, paso por paso y lo pude solucionar.

Por otra parte, un bug que todavía sigue en la aplicación es que actualmente cuando uno apreta el botón para activar el modo mayhem, ocurre que algunas veces llega por duplicado el evento del click del botón.

Aquí vemos que en la página (chess.html) se apretó una vez el botón de mayhem, mientras que en el programa escrito en elm, se recibió dos veces el evento. No pude decidir si el error es por algo de Elm o un error en la implementación.

Futuras extensiones

Una duda que surgió es si un programa de ajedrez puede aprender jugando de forma random contra sí mismo. Hipotéticamente, pienso que si tuviera tiempo tendiendo a infinito, eventualmente llegaría a aprender a jugar contra sí mismo. Si dispongo de tiempo en un futuro, me gustaría seguir el proyecto y agregar distintos algoritmos de inteligencia artificial para identificar de cual manera aprende más rápido.

---

[1] http://elm-lang.org/blog/announce/0.12.1

Por otra parte, hay un asunto que es particularmente ineficiente y es que el estado del juego se está actualizando cada 1 segundo. Lo ideal sería sólo recibir los eventos de actualización si se encuentra activado el modo Mayhem. El problema es que si está activado o no este modo corresponde al estado del juego, y no al modelo de input, no pude encontrar la manera de filtrar solo el tick solamente si está activado el modo Mayhem.

Conclusiones

La verdad que disfruté mucho realizando este juego. En principio, no entendía bien la parte reactive del paradigma, y me di la cabeza contra la pared bastantes veces. El modificar la UI también me resultó en un principio difícil. Se dió que dejaba el juego un tiempo, lo volvía a ver y me la pasaba resolviendo cosas de estilo que estaban mal.
También, el juego mutó mucho desde el principio a lo que es ahora: empezó siendo un juego de consola hecho en haskell, con una implementación mucho más básica.
Vale la pena mencionar también que el mayor avance lo obtuve luego de empezar a utilizar Swift por razones laborales. Swift es un lenguaje OOP con muchas características del paradigma funcional. Cómo utilicé el approach FRP en Swift, al volver al ajedrez entendí completamente este paradigma.

Bibliografía

http://scrambledeggsontoast.github.io/2014/05/09/writing-2048-elm/
http://elm-lang.org/

Anexo

Piece.elm

```
module Piece where

import Utils exposing (..)
import List exposing (filter, any)

type PieceColor = White | Black
type PieceType = Pawn | Knight | Bishop | Rook | Queen | King
type Piece = Piece PieceColor PieceType

switchColor: PieceColor -> PieceColor
switchColor color = case color of
                      Black -> White
                      White -> Black

stringToPiece: String -> String -> Piece
stringToPiece c t = Piece (stringToColor c) (stringToType t)

stringToColor: String -> PieceColor
stringToColor c = case c of
                    "White" -> White
                    "Black" -> Black

stringToType: String -> PieceType
stringToType t = case t of
                    "Rook" -> Rook
                    "Knight" -> Knight
                    "King" -> King
                    "Bishop" -> Bishop
                    "Queen" -> Queen
                    "Pawn" -> Pawn

pieceColorMatches: PieceColor -> Piece -> Bool
pieceColorMatches color (Piece c t) = c == color

pieceToString: Piece -> (String, String)
pieceToString (Piece c t) = (toString c, toString t)

initialPieces: List (Position, Piece)
initialPieces = initialPiecesFromColor White ++ initialPiecesFromColor Black

initialPiecesFromColor: PieceColor -> List (Position, Piece)
initialPiecesFromColor c = initialSpecialPiecesFromColor c ++ initialPawnsFromColor c

initialSpecialPiecesFromColor: PieceColor -> List (Position, Piece)
initialSpecialPiecesFromColor c = List.map2 (\j t -> ((getInitialRowFromColor c t, j),
Piece c t)) [0..(boardSize - 1)] [Rook, Knight, Bishop, Queen, King, Bishop, Knight, Rook]

initialPawnsFromColor: PieceColor -> List (Position, Piece)
```

```
initialPawnsFromColor c = List.map (\j -> ((getInitialRowFromColor c Pawn, j), Piece c
Pawn)) [0..(boardSize - 1)]

getInitialRowFromColor: PieceColor -> PieceType -> Int
getInitialRowFromColor c t = case (c,t) of
                                (Black, Pawn) -> 1
                                (White, Pawn) -> 6
                                (Black, _) -> 0
                                (White, _) -> boardSize - 1

-- We only care for pawns first moves, as this functions is used for checking if pawn can
move one or two spaces.
isFirstMove: Piece -> Position -> Bool
isFirstMove piece (x,y) = case piece of
                        Piece White Pawn -> x == 6
                        Piece Black Pawn -> x == 1
                        otherwise -> False

first: (Piece, Int, Int) -> Piece -- If named fst, it conflicts with fst
first (piece, x, y) = piece

second: (Piece, Int, Int) -> Int -- If named snd, it conflicts with snd
second (piece, x, y) = x

thrd: (Piece, Int, Int) -> Int
thrd (piece, x, y) = y

canMoveRook: Int -> Int -> Bool
canMoveRook difCol difRow = difCol == 0 || difRow == 0 && not (difCol == 0 && difRow == 0)

canMoveQueen: Int -> Int -> Bool
canMoveQueen difx dify = canMoveRook difx dify || canMoveBishop difx dify

canMoveKnight: Int -> Int -> Bool
canMoveKnight difCol difRow = case (difCol,difRow) of
                              (1,2) -> True
                              (2,1) -> True
                              otherwise -> False

canMoveBishop: Int -> Int -> Bool
canMoveBishop difx dify = difx /= 0 && difx == dify

canMoveKing: Int -> Int -> Bool
canMoveKing difCol difRow = case (difCol, difRow) of
                            (1,1) -> True
                            (1,0) -> True
                            (0,1) -> True
                            otherwise -> False

piecePossibleMoves: Piece -> Position -> Position -> List (Position)
piecePossibleMoves piece from to =
                            case piece of
                            Piece _ Rook -> rookPossibleMoves from to
                            Piece _ Bishop -> bishopPossibleMoves from to
```

```
                        Piece _ Queen -> queenPossibleMoves from to
                        Piece White Pawn -> whitePawnPossibleMoves from to
                        Piece Black Pawn -> blackPawnPossibleMoves from to
                        otherwise -> []

whitePawnAttackPositions: Position -> Position -> Bool
whitePawnAttackPositions from to = to == (from `minus` (1,1)) || to == (from `minus`
(1,-1))

blackPawnAttackPositions: Position -> Position -> Bool
blackPawnAttackPositions from to = to == (from `plus` (1,1)) || to == (from `plus` (1,-1))

queenPossibleMoves: Position -> Position -> List (Position)
queenPossibleMoves from to = rookPossibleMoves from to ++ bishopPossibleMoves from to

whitePawnPossibleMoves: Position -> Position -> List (Position)
whitePawnPossibleMoves from to = case abs (rowsBetween from to) of
                                 1 -> []
                                 2 -> [(fst from - 1, snd from)]
                                 otherwise -> []

blackPawnPossibleMoves: Position -> Position -> List (Position)
blackPawnPossibleMoves from to = case abs (rowsBetween from to) of
                                 1 -> []
                                 2 -> [(fst from + 1, snd from)]
                                 otherwise -> []

rookPossibleMoves: Position -> Position -> List (Position)
rookPossibleMoves from to = let
                              difRow = fst to - fst from
                              difCol = snd to - snd from
                            in case (difRow, difCol) of
                              (0, _) -> filter (\position -> sameRow from position && any
(\x -> x == getColumn position) (interval (getColumn from) (getColumn to))) (rookMoves from
to)
                              (_, 0) -> filter (\position -> sameColumn from position &&
any (\x -> x == getRow position) (interval (getRow from) (getRow to))) (rookMoves from to)
                              otherwise -> []

bishopPossibleMoves: Position -> Position -> List (Position)
bishopPossibleMoves from to = let
                                difRow = fst to - fst from
                                difCol = snd to - snd from
                              in case (sign difRow, sign difCol) of
                              (1, 1) -> filter (\position -> position `downRightOf` from &&
position `upperLeftOf` to) (bishopMoves from to)
                              (-1, -1) -> filter (\position -> position `upperLeftOf` from
&& position `downRightOf` to) (bishopMoves from to)
                              (1, -1) -> filter (\position -> position `downLeftOf` from &&
position `upperRightOf` to) (bishopMoves from to)
                              (-1, 1) -> filter (\position -> position `upperRightOf` from
&& position `downLeftOf` to) (bishopMoves from to)
                              otherwise -> []
```

```
bishopMoves: Position -> Position -> List (Position)
bishopMoves from to = filter (\x -> x /= to && x /= from && canMoveBishop (abs (fst x - fst
from)) (abs (snd x - snd from))) allPositions

-- All positions between the rook and its objective, without from and to
rookMoves: Position -> Position -> List (Position)
rookMoves from to = filter (\x -> x /= to && x /= from && canMoveRook (fst x - fst from)
(snd x - snd from)) allPositions
```

Tile.elm

```
module Tile where

import Piece exposing (..)
import Utils exposing (..)

import Maybe exposing (Maybe)
import List exposing (map)

type alias Tile = Maybe Piece

stringToTile: (String, String) -> Tile
stringToTile (c,t) = case (c,t) of
                        ("","") -> Nothing
                        otherwise -> Just (stringToPiece c t)

tileToString: Tile -> (String, String)
tileToString t = case t of
                Nothing -> ("", "")
                Just p -> pieceToString p

initialTiles: List (Position, Tile)
initialTiles = map (\(position, piece) -> (position, Just piece)) initialPieces

colorMatches: PieceColor -> Tile -> Bool
colorMatches color tile = mapWithDefault (pieceColorMatches color) tile False
```

Board.elm

```
module Board where

import Piece exposing (..)
import Tile exposing (..)
import Utils exposing (..)

import Array exposing (Array, set, repeat)
import Maybe exposing (Maybe, andThen)
import List exposing (foldr, filter, filterMap)

type alias Board = Array (Array Tile)

emptyBoard: Board
emptyBoard = repeat boardSize <| repeat boardSize <| Nothing
```

```
startBoard: Board
startBoard = setTiles (initialTiles) emptyBoard

readTile: Position -> Board -> Tile -- (!) returns Maybe, so arr ! j is a Maybe Tile and we
need Tile
readTile (i,j) g = g ! i `andThen` \arr -> arr ! j `andThen` identity

setTile: Position -> Tile -> Board -> Board
setTile (i,j) t g = let r = g ! i in
                      case r of
                        Nothing -> g
                        Just arr -> set i (set j t arr) g

setTiles: List (Position, Tile) -> Board -> Board
setTiles arr b = foldr (\(position, tile) board -> setTile position tile board) b arr

tilesWithCoordinates: Board -> List (Tile, Int, Int)
tilesWithCoordinates b = foldr (\i tiles -> tiles ++ getTilesForRow i b) []
[0..(boardSize-1)]

getTilesForRow: Int -> Board -> List (Tile, Int, Int)
getTilesForRow i b = foldr (\j arr -> arr ++ [(readTile (i,j) b, i , j)]) []
[0..(boardSize-1)]

makeMove: Board -> Position -> Position -> Board
makeMove board from to =  let
                            tile = readTile from board
                          in
                            setTiles [(from, Nothing), (to, tile)] board

whitePawnCanAttack: Position -> Position -> Board -> Bool
whitePawnCanAttack from to board = whitePawnAttackPositions from to && (readTile to board)
/= Nothing

blackPawnCanAttack: Position -> Position -> Board -> Bool
blackPawnCanAttack from to board = blackPawnAttackPositions from to && (readTile to board)
/= Nothing

getPiecesFromBoard: Board -> List (Piece, Int, Int)
getPiecesFromBoard board = filterMap (\(t, x, y) -> case t of
                                                      Nothing -> Nothing
                                                      Just piece -> Just (piece, x, y))
(tilesWithCoordinates board)

getPiecesForColor: Board -> PieceColor -> List (Piece, Int, Int)
getPiecesForColor board color = filter (\(t, x, y) -> pieceColorMatches color t)
(getPiecesFromBoard board)

promotePossiblePawns: Board -> Tile -> Position -> Board
promotePossiblePawns board tile position = case (position, tile) of
                                        ((0, _), Just (Piece White Pawn)) -> setTile
position (Just (Piece White Queen)) board
```

```
                                    ((7, _), Just (Piece Black Pawn)) -> setTile
position (Just (Piece Black Queen)) board
                                    otherwise -> board

canMoveWhitePawn: Board -> Position -> Position -> Bool
canMoveWhitePawn board from to = readTile to board == Nothing &&
                                 ((to `minus` from) == (-1, 0) ||
                                   case (isFirstMove (Piece White Pawn) from) of
                                       True -> (to `minus` from) == (-2, 0)
                                       False -> False
                                 )

canMoveBlackPawn: Board -> Position -> Position -> Bool
canMoveBlackPawn board from to = readTile to board == Nothing &&
                                 ((to `minus` from) == (1, 0) ||
                                   case (isFirstMove (Piece Black Pawn) from) of
                                       True -> (to `minus` from) == (2, 0)
                                       False -> False
                                 )
```

GameModel.elm

```
module GameModel where

import Utils exposing (Position)
import Piece exposing (PieceColor)
import Board exposing (Board, startBoard)
import Random exposing (Seed)

type Progress = InProgress | WhiteWon | BlackWon
type alias PlayerColor = PieceColor
type GameType = OneVSOne | OneVSComputer

type alias GameState = {
    board: Board,
    gameProgress: Progress,
    turn: PlayerColor,
    selected: Maybe Position,
    cursorAt: Position,
    gameType: GameType,
    mayhem: Bool,
    seed: Maybe Seed
}

defaultGame: GameState -- the default starting game state
defaultGame = {
    board = startBoard,
    gameProgress = InProgress,
    turn = Piece.White,
    selected = Nothing,
    cursorAt = (6,4),
    gameType = OneVSOne,
    mayhem = False,
    seed = Nothing
```

```
    }
```

InputModel.elm

```
module InputModel where

import Signal exposing (..)
import Keyboard

type Direction = Up | Down | Left | Right | None

type Update = Move Direction | NewGameButtonPressed Bool | GameTypeChanged Bool |
EnterPressed Bool | MayhemActivated Bool | Tick Float

type alias Input = { -- inputs that the game will depend of
    action: Update, -- the user actions
    currentTimestamp: Int -- the current time
}

-- Signal which represents the direction that the user has chosen.
-- Compatible with both the wasd and arrow keys.
playerDirection: Signal Direction
playerDirection = let toDirection ds =
                      if | ds == {x = 0, y = 1} -> Up
                         | ds == {x = 0, y = -1} -> Down
                         | ds == {x = 1, y = 0} -> Right
                         | ds == {x = -1, y = 0} -> Left
                         | otherwise -> None
                      in merge (toDirection <~ Keyboard.arrows) (toDirection <~
Keyboard.wasd)

enterPressed: Signal Bool
enterPressed = Keyboard.enter
```

Chess.elm

```
module Chess where

import InputModel exposing (..)
import GameModel exposing (GameState, defaultGame)
import GameManager exposing (checkValidGame)

import Signal exposing (..)
import Rendering exposing (display)
import Time exposing (every, second)
import Date exposing (year, hour, minute, second, fromTime)


-- These ports are directly connected to JavaScript, they are incoming ports. Their
counterpart is in chess.html.
-- Semantically, they should be declared on InputModel, but elm requires ports to be
declared on main elm file.
port newGameButton: Signal Bool
```

```
port gameTypeChangedButton: Signal Bool
port currentTimestamp: Signal Int
port mayhemActivated: Signal Bool

-- These are the actions that will trigger a new gameState.
actions: Signal Update
actions = mergeMany
            [ map Move playerDirection,
            map NewGameButtonPressed newGameButton,
            map GameTypeChanged gameTypeChangedButton,
            map EnterPressed enterPressed,
            map MayhemActivated mayhemActivated,
            map Tick clock
            ]

input = Input <~ actions ~ currentTimestamp

-- Folds the input into the gameState, starting with the defaultGame.
gameState: Signal GameState
gameState = foldp checkValidGame defaultGame input

-- Display the game.
main = display <~ gameState

-- The clock is used as an action because when mayhem is activated, we want computer to
play against itself without the need
-- of making a move, so we need to update gameState. This isn't the best solution, as an
event is trigger every second; In a future implementation, it would be better to trigger
the clock depending on GameTypeChanged.
clock: Signal Float
clock = Time.every Time.second
```

GameManager.elm

```
module GameManager where

import GameModel exposing (..)
import InputModel exposing (..)
import Utils exposing (..)
import Piece exposing (PieceColor)
import Tile exposing (Tile)
import Board exposing (Board, readTile, getPiecesForColor, promotePossiblePawns)
import Logic exposing (inCheck, canMakeMove, canMoveTile, loopUntilCanMovePiece)

import Random exposing (Seed, initialSeed)

checkValidGame: Input -> GameState -> GameState
checkValidGame input gameState =  if gameFinished gameState
                                    then gameState
                                    else (checkPreConditions input gameState) |> stepGame
input |> checkPostCondition

checkPreConditions: Input -> GameState -> GameState
```

```
checkPreConditions input gameState =
                              let
                                newGameState = { gameState | seed <- Just (initialSeed
input.currentTimestamp) }
                              in
                                case input.action of
                                MayhemActivated True -> switchMayhem newGameState
                                NewGameButtonPressed True ->  let
                                                                initialGame = startingGame
input
                                                              in
                                                                { initialGame | gameType <-
gameState.gameType }
                                GameTypeChanged True -> switchGameType newGameState
                                otherwise -> newGameState

stepGame: Input -> GameState -> GameState
stepGame input gameState =  if
                              | gameState.mayhem
                                -> simulateComputerPlayer input gameState
                              | gameState.gameType == OneVSComputer && gameState.turn ==
Piece.Black
                                -> simulateComputerPlayer input gameState
                              | otherwise
                                -> case input.action of
                                    Move direction -> moveCursor gameState direction
                                    EnterPressed True -> playerMakingMove input gameState
                                    otherwise -> gameState

--Color is already switched
checkPostCondition: GameState -> GameState
checkPostCondition gameState = if gameState.gameType == OneVSOne && not (gameState.mayhem)
                                 then if | gameState.turn == Piece.Black && inCheck
gameState.board Piece.Black
                                           -> { gameState | gameProgress <- colorLost
Piece.Black }
                                         | gameState.turn == Piece.White && inCheck
gameState.board Piece.White
                                           -> { gameState | gameProgress <- colorLost
Piece.White }
                                         | otherwise -> gameState
                               else  gameState

switchMayhem: GameState -> GameState
switchMayhem g = let x = not g.mayhem in { g | mayhem <- x }

gameFinished: GameState -> Bool
gameFinished gameState = gameState.gameProgress == WhiteWon || gameState.gameProgress ==
BlackWon

moveCursorToOrigin: GameState -> GameState
moveCursorToOrigin gameState = case gameState.turn of
                         Piece.Black -> if gameState.gameType /= OneVSComputer
                                        then { gameState | cursorAt <- (1,4) }
```

```
                                    else gameState
                            Piece.White -> { gameState | cursorAt <- (6,4) }

playerMakingMove: Input -> GameState -> GameState
playerMakingMove input gameState =  let
                                      selected = gameState.selected -- Position selected
                                      tileAt = readTile gameState.cursorAt gameState.board
-- tile where cursor is
                                    in case (selected, tileAt) of
                                    (Nothing, Nothing) -> gameState
                                    (Nothing, Just piece) -> { gameState | selected <- Just
(gameState.cursorAt) }
                                    (Just position, Nothing) -> attemptMove gameState
(readTile position gameState.board) position gameState.cursorAt (\gamestate -> { gamestate
| selected <- Nothing })
                                    (Just position, Just piece) -> attemptMove gameState
(readTile position gameState.board) position gameState.cursorAt (\gamestate -> { gamestate
| selected <- Just (gameState.cursorAt) })

startingGame: Input -> GameState
startingGame input = defaultGame

switchGameType: GameState -> GameState
switchGameType gameState = { gameState | gameType <- case gameState.gameType of
                                                        OneVSOne -> OneVSComputer
                                                        OneVSComputer -> OneVSOne
                           }

simulateComputerPlayer: Input -> GameState -> GameState
simulateComputerPlayer input gameState = makeRandomMove gameState

makeRandomMove: GameState -> GameState
makeRandomMove gameState = let
                             pieces = getPiecesForColor gameState.board gameState.turn
                           in case gameState.seed of
                             Nothing -> gameState
                             Just seed ->
                               let
                                 maybeMovement = loopUntilCanMovePiece gameState.board
gameState.turn seed pieces
                               in case maybeMovement of
                                 Nothing -> { gameState | gameProgress <- colorLost
gameState.turn }
                                 Just (from, to, newSeed) -> let newGameState = makeMove
gameState (readTile from gameState.board) from to in { newGameState | seed <- Just
(newSeed) }

colorLost: PlayerColor -> Progress
colorLost color = case color of
                  Piece.Black -> WhiteWon
                  Piece.White -> BlackWon
```

```
attemptMove: GameState -> Tile -> Position -> Position -> (GameState -> GameState) ->
GameState
attemptMove g t from to f = if canMoveTile g.board g.turn t from to
                          then f (makeMove g t from to)
                          else f g

makeMove: GameState -> Tile -> Position -> Position -> GameState
makeMove g tile from to =  moveCursorToOrigin { g | board <- promotePossiblePawns
(Board.makeMove g.board from to) tile to,
                          selected <- Nothing,
                          turn <- case g.turn of
                                  Piece.White -> Piece.Black
                                  Piece.Black -> Piece.White
                        }

moveCursor: GameState -> Direction -> GameState
moveCursor gameState direction =
            let
              newPosition = getNewPosition gameState.cursorAt (getIncrement direction)
            in
              if direction /= None && isInsideBoard newPosition
                then { gameState | cursorAt <- newPosition }
                else gameState
```

Logic.elm

```
module Logic where

import Piece exposing (..)
import Tile exposing (..)
import Board exposing (..)
import Utils exposing (..)

import Maybe exposing (Maybe, withDefault)
import Array exposing (Array, toList)
import Random exposing (Seed)
import List exposing (filter, any, all, head, drop, take, foldr, tail)

canMoveTile: Board -> PieceColor -> Tile -> Position -> Position -> Bool
canMoveTile board color tile from to = mapWithDefault (\piece -> canMakeMove board color
piece from to) tile False

canMakeMove: Board -> PieceColor -> Piece -> Position -> Position -> Bool
canMakeMove board color piece from to =
    let
      tileTo = readTile to board
    in
      pieceColorMatches color piece &&
      not (colorMatches color tileTo) &&
      pieceCanReachPosition board piece from to &&
      let
        newBoard = makeMove board from to
      in
        notInCheck newBoard color
```

```
pieceCanReachPosition: Board -> Piece -> Position -> Position -> Bool
pieceCanReachPosition board piece from to = notOtherPieceInPath board piece from to &&
(moveIsValid board piece from to || canAttack board piece from to)

notOtherPieceInPath: Board -> Piece -> Position -> Position -> Bool
notOtherPieceInPath board piece from to = let
                                    positions = piecePossibleMoves piece from to
                                  in
                                  case piece of
                                    Piece _ Knight -> True
                                    Piece _ King -> True
                                    otherwise -> all (\position -> readTile position
board == Nothing) positions

notInCheck: Board -> PieceColor -> Bool
notInCheck board color = not (inCheck board color)

inCheck: Board -> PieceColor -> Bool
inCheck board color = let
                        pieces = getPiecesFromBoard board
                        otherColorPieces = getPiecesForColor board (switchColor color)
                        king = head (filter (\(t, x, y) -> t == Piece color King) pieces)
                      in
                        case king of
                          Nothing -> False -- Should not enter here, but king is Maybe.
                          Just kingPosition -> any (\(piece, x, y) -> pieceCanReachPosition
board piece (x,y) (second kingPosition, thrd kingPosition)) otherColorPieces

canAttack: Board -> Piece -> Position -> Position -> Bool
canAttack board piece from to = case piece of
                            Piece White Pawn -> whitePawnCanAttack from to board
                            Piece Black Pawn -> blackPawnCanAttack from to board
                            otherwise -> False

moveIsValid: Board -> Piece -> Position -> Position -> Bool
moveIsValid board piece from to = let
                              difRow = fst to - fst from
                              difCol = snd to - snd from
                            in case piece of
                              Piece _ Rook ->  canMoveRook difCol difRow
                              Piece _ Knight -> canMoveKnight (abs difCol) (abs difRow)
                              Piece _ Bishop -> canMoveBishop (abs difCol) (abs difRow)
                              Piece _ Queen -> canMoveQueen (abs difCol) (abs difRow)
                              Piece White Pawn -> canMoveWhitePawn board from to
                              Piece Black Pawn -> canMoveBlackPawn board from to
                              Piece _ King -> canMoveKing (abs difRow) (abs difCol)

loopUntilCanMovePiece: Board -> PieceColor -> Seed -> List(Piece, Int, Int) ->
Maybe(Position, Position, Seed)
loopUntilCanMovePiece board color seed pieces = let
                                                  (shuffledPieces, newSeed) = shuffle
pieces seed
                                                in
```

```
                                                evaluatePossibleMovement board color
shuffledPieces newSeed

evaluatePossibleMovement: Board -> PieceColor -> List(Piece, Int, Int) -> Seed ->
Maybe(Position, Position, Seed)
evaluatePossibleMovement board color pieces seed = case head pieces of
                                        Nothing -> Nothing
                                        Just (piece,x,y) -> let
                                                              (randomPositions, newSeed)
= getRandomPositionsForPiece board piece (x, y) seed
                                                              move = firstThatSatisfies
(\randomPosition -> canMakeMove board color piece (x,y) randomPosition) (Just
randomPositions)
                                                            in case move of
                                                                 Nothing ->
evaluatePossibleMovement board color (getTail pieces) newSeed
                                                                 Just position -> Just
((x,y), position, newSeed)

getTail: List a -> List a
getTail list = withDefault [] (tail (list))

getRandomPositionsForPiece: Board -> Piece -> Position -> Seed -> (List Position, Seed)
getRandomPositionsForPiece board piece from seed = shuffle (generatePossiblePositions board
piece from) seed

generatePossiblePositions: Board -> Piece -> Position -> List Position
generatePossiblePositions board piece from = allPositions |> List.filter (moveIsValid board
piece from)
```

Rendering.elm

```
module Rendering where

import Utils exposing (..)
import GameModel exposing (..)
import Tile exposing (..)
import Board exposing (..)
import Logic exposing (..)
import Piece exposing (..)

import Color exposing (..)
import Text exposing (..)
import Graphics.Collage exposing (..)
import Graphics.Element exposing (..)
import List exposing (..)

tileSize: Float
tileSize = 70

tileMargin: Float
tileMargin = 8

darkgreen = rgb 77 153 60
```

```
tileColor: Tile -> Utils.Position -> GameState -> Color
tileColor tile position gameState =
        if  | gameState.mayhem == True
                -> displayNormalTile position
            | position == gameState.cursorAt && (gameState.turn == White ||
gameState.gameType /= OneVSComputer)
                -> red
            | gameState.selected /= Nothing
                -> case gameState.selected of
                    Just tileSelected -> if canMoveTile gameState.board gameState.turn
(readTile tileSelected gameState.board) tileSelected position then blue else
displayNormalTile position
            | otherwise
                -> displayNormalTile position

canMoveTile: Board -> PieceColor -> Tile -> Utils.Position -> Utils.Position -> Bool
canMoveTile board color tile from to = mapWithDefault (\piece -> canMakeMove board color
piece from to) tile False

displayNormalTile: Utils.Position -> Color
displayNormalTile (x,y) = case (isEven x, isEven y) of
                            (True, True) -> lightBrown
                            (False, False) -> lightBrown
                            otherwise -> darkBrown

wonTextColor: Tile -> Color -- the text color of a tile
wonTextColor tile = black

wonTextSize: Tile -> Float -- the text size of a tile
wonTextSize tile = 50

wonTextStyle: Tile -> Style -- the text style of a tile
wonTextStyle tile = {
                  typeface = [ "Helvetica Neue", "Arial", "sans-serif" ]
                , height = Just <| wonTextSize tile
                , color = wonTextColor tile
                , bold = True
                , italic = False
                , line = Nothing
                }

displayTile: Tile -> Utils.Position -> GameState -> Element -- display a tile
displayTile tile position gameState =
        let tileBackground = filled (tileColor tile position gameState)
        <| square tileSize
                in case tile of
                    Just (Piece c t) -> collage (round tileSize) (round tileSize)
                        [
                            tileBackground,
                            toForm (image 75 75 ("images/" ++ toString c ++ "_" ++ toString
t ++ ".jpg"))
                        ]
                    Nothing -> collage (round tileSize) (round tileSize)
```

```
                        [
                            tileBackground
                         ]

displayTileAtCoordinates: (Tile, Int, Int) -> GameState -> Form
displayTileAtCoordinates (t,i,j) gameState =
                        let
                            p = ((tileSize + tileMargin) * (toFloat j - (toFloat boardSize
- 1)/2),
                             (-1) * (tileSize + tileMargin) * (toFloat i - (toFloat
boardSize - 1)/2))
                        in
                            move p <| toForm <| displayTile t (i,j) gameState

boardWidth: Float -- the width of the entire game grid
boardWidth = (toFloat boardSize) * tileSize + (1 + toFloat boardSize) * tileMargin

displayGrid: GameState -> Element -- display a grid
displayGrid gameState = let
                    gridBox = filled (rgb 187 173 160) -- the grid background
                                <| square boardWidth
                    tiles = map (\position -> displayTileAtCoordinates position gameState)
                        <| tilesWithCoordinates gameState.board
    in collage (round boardWidth) (round boardWidth) ([gridBox] ++ tiles)

displayOverlay: Style -> Color -> String ->  Element -- display an overlay
                                                     -- with a message
displayOverlay s c t = collage (round boardWidth) (round boardWidth)
    [
      filled c <| square boardWidth -- background
    , toForm <| centered <| style s <| fromString t -- message
    ]

wonOverlayStyle: Style
wonOverlayStyle = wonTextStyle <| Just (Piece Black King)

wonOverlayColor: Color
wonOverlayColor = rgba 237 194 46 0.5

displayWonOverlay: String -> Element -- display a game won overlay
displayWonOverlay message = displayOverlay
                            wonOverlayStyle
                            wonOverlayColor
                            message

applyOverlay: Element -> Element -> Element
applyOverlay overlay board = collage (round boardWidth) (round boardWidth)
        [
          toForm <| board
        , toForm <| overlay
        ]

display: GameState -> Element -- display a gamestate
display gameState = displayGrid gameState
```

```
                |> case gameState.gameProgress of
                    WhiteWon -> applyOverlay (displayWonOverlay "White wins!")
                    BlackWon-> applyOverlay (displayWonOverlay "Black wins")
                    otherwise -> identity
```

Utils.elm

```
module Utils where

import InputModel exposing (..)

import Array exposing (Array, get)
import Maybe exposing (Maybe, map, withDefault)
import Random exposing (Seed, generate, int)
import List exposing (head, tail, concat, map, take, drop)

type alias Position = (Int, Int)

allPositions: List (Int, Int)
allPositions = concat (List.map (\x -> List.map (\y -> (x,y)) [0..(boardSize-1)])
[0..(boardSize-1)])

boardSize: Int
boardSize = 8

infixl 9 !
(!): Array a -> Int -> Maybe a -- the nth element of a list
arr ! n = get n arr

getElement: Maybe(List a) -> Int -> Maybe a
getElement maybeList n = case maybeList of
                          Nothing -> Nothing
                          Just list -> if n == 0 then head list else getElement (tail list)
(n - 1)

plus: Position -> Position -> Position
plus lhs rhs = (fst lhs + fst rhs, snd lhs + snd rhs)

minus: Position -> Position -> Position
minus lhs rhs = (fst lhs - fst rhs, snd lhs - snd rhs)

infixl 9 |||
(|||): Bool -> Bool -> Bool
a ||| b = (a || b) && not (a && b)

interval: Int -> Int -> List (Int)
interval x1 x2 = [min x1 x2 .. max x1 x2]

getRow: Position -> Int
getRow x = fst x

getColumn: Position -> Int
getColumn x = snd x
```

```
sameRow: Position -> Position -> Bool
sameRow x y = getRow x == getRow y

sameColumn: Position -> Position -> Bool
sameColumn x y = getColumn x == getColumn y

upperRightOf: Position -> Position -> Bool
upperRightOf x y = x `atRightOf` y && x `atUpOf` y

upperLeftOf: Position -> Position -> Bool
upperLeftOf x y = x `atLeftOf` y && x `atUpOf` y

downRightOf: Position -> Position -> Bool
downRightOf x y = x `atRightOf` y && x `atDownOf` y

downLeftOf: Position -> Position -> Bool
downLeftOf x y = x `atLeftOf` y && x `atDownOf` y

atRightOf: Position -> Position -> Bool
atRightOf x y = getColumn x > getColumn y

atLeftOf: Position -> Position -> Bool
atLeftOf x y = getColumn x < getColumn y

atUpOf: Position -> Position -> Bool
atUpOf x y = getRow x < getRow y

rowsBetween: Position -> Position -> Int
rowsBetween x y = getRow x - getRow y

atDownOf: Position -> Position -> Bool
atDownOf x y = getRow x > getRow y

sign: Int -> Int
sign x = if x > 0 then 1 else (-1)

isEven: Int -> Bool
isEven x = (x % 2 ) == 0

isOdd: Int -> Bool
isOdd x = not (isEven x)

mapWithDefault: (a -> b) -> Maybe a -> b -> b
mapWithDefault f m h = withDefault h (Maybe.map f m)

isInsideBoard: Position -> Bool
isInsideBoard (x,y) = x >= 0 && x < boardSize && y >= 0 && y < boardSize

-- This is implemented because elm is not lazy
firstThatSatisfies: (a -> Bool) -> Maybe (List a) -> Maybe a
firstThatSatisfies f maybeList = case maybeList of
                                    Nothing -> Nothing
                                    Just list -> let
                                                    element = head list
```

```
                                        in if mapWithDefault f element False
                                          then element
                                          else firstThatSatisfies f (tail list)

generateRandomPosition: Position -> Position -> Seed -> (Position, Seed)
generateRandomPosition from to seed = let
                                        (x, newSeed) = generateRandomInt (fst
from) (fst to) seed
                                        (y, newSeed2) = generateRandomInt (snd
from) (snd to) newSeed
                                      in
                                        ((x,y), newSeed)

generateRandomInt: Int -> Int -> Seed -> (Int, Seed)
generateRandomInt from to seed = generate (int from to) seed

shuffle: List a -> Seed -> (List a, Seed)
shuffle list seed = if List.isEmpty list
                    then ([], seed)
                    else
                    let
                      (randomIndex, newSeed) = generateRandomInt 0 (List.length list - 1)
seed
                      maybeElement = getElement (Just list) randomIndex
                    in
                      case maybeElement of
                        Nothing -> ([], newSeed)
                        Just (element) ->
                          let
                            shuffled = shuffle (without randomIndex list) newSeed
                          in ([element] ++ fst shuffled, snd shuffled)

without: Int -> List a -> List a
without i list =
  let before = take i list
      after = drop (i+1) list
  in
    before ++ after

getNewPosition: Position -> Position -> Position
getNewPosition (from, to) (incX, incY) = (from + incX, to + incY)

getIncrement: Direction -> Position
getIncrement direction = case direction of
                           Up -> ((-1), 0)
                           Down -> (1, 0)
                           Left -> (0, -1)
                           Right -> (0, 1)
                           None -> (0, 0)
```