Trabajo Práctico Final

Blob Wars

Programación Funcional

Alumno

Argentino Ducret 52194

Profesores

Martinez Lopez, Pablo Ernesto

Pennella, Valeria Verónica

Introducción

El objetivo del trabajo práctico es implementar el juego Blob Wars en el lenguaje de programación Elm, el cual se encuentra dentro del paradigma funcional.

Este juego es para dos jugadores, consta de un tablero 8x8 que inicialmente se parte con dos fichas o manchas para cada jugador ubicadas en las esquinas del tablero, el juego finaliza al completar el tablero o cuando un jugador pierde todas sus fichas.

El lenguaje Elm es un lenguaje con una curva de aprendizaje rápida para la mayoría de los conceptos básicos y esenciales, además cuenta con una arquitectura muy particular que vale la pena conocer y aplicar a otros proyectos que no necesariamente estén desarrollados en lenguajes funcionales puros.

Se comienza explicando las reglas de juegos, luego se desarrolla la estructura o arquitectura que el lenguaje Elm necesita para una correcta implementación, se detallaran los modelos adoptados para representar el juego, se analizaran algunas tecnicas de debug y por ultimo instrucciones de compilación.

Reglas de juego

El objetivo del juego es dominar el tablero y obtener la mayor cantidad de manchas. El juego finaliza cuando el tablero esté completo o un jugador se quede sin fichas y el que más manchas tenga será el ganador.

Hay dos colores azul y rojo que representan a los dos jugadores, inicialmente se colocan de una lado del tablero las dos fichas azules en las esquinas y del mismo modo se colocan las fichas rojas del otro lado del tablero. Siempre comienza a jugar el jugar de color azul.

Los posibles movimientos son seleccionar una mancha del color del jugador y moverla 1 o 2 casilleros en cualquier dirección (la manchas propias o del rival no bloquean el movimiento). Si el movimiento es de 1 casillero la mancha original seleccionada se queda en su lugar y genera una ficha nueva en el casillero seleccionado. En caso contrario, si el movimiento es de 2 casillas la mancha original se mueve al lugar destino.

La forma de conquistar o manchar la fichas enemigas es realizar un movimiento y automáticamente se manchas las fichas rivales que estén a radio 1 de la ficha que se movió.

Arquitectura del lenguaje Elm

En esta sección se detallara la arquitectura básica que propone este lenguaje funcional.

Para comenzar es importante notar que Elm es un lenguaje estáticamente tipado, esto quiere decir que en tiempo de compilación se detectan o infieren todos los tipos que se van a utilizar ya sean en variables, funciones, etc. Es decir que no es posible que tengamos algún tipo de error de casteo o crash al aplicar alguna función como suele suceder en lenguaje imperativos.

Además Elm es un lenguaje funcional puro, en otras palabras no se pueden realizar cambios de estados o mutar datos. Será necesario perder la costumbre de los lenguajes imperativos tanto de declarar variables, tener objetos con estados, entre otras prácticas.

Arquitectura basica

La lógica de los programas desarrollados en Elm se pueden distribuir en tres secciones principales y bien diferenciadas:

1. Model: En esta sección se declaran todos los tipos que se usarán y se inicializa el modelo que utiliza la arquitectura de Elm. Es muy importante tener bien definido el modelo para poder trabajar con este en las siguientes secciones. En pocas palabras, representa el estado de la aplicación.
2. Update: Es la función encargada de actualizar el modelo al recibir algún mensaje. Los mensajes son otro actor importante en esta arquitectura, normalmente los mensajes son enviados por las acciones que realiza el usuario en la UI.
3. View: Es una funciona que dado un modelo lo traduce a HTML o a una vista con la cual el usuario interactuara.

Existe también una arquitectura un poco más compleja que se podría usar en Elm, la cual agrega señales, subscripciones, comandos, puertos para comunicarse con codigo JS, entre otras cosas que no serán abarcadas en este trabajo.

Ejemplos de la arquitectura básica

Todo programa debe indicar las 3 partes básica antes mencionadas, de esta forma Elm internamente llamara a las funciones de Update y View con el modelo correspondiente.

```
main = Html.beginnerProgram
  {
    model   = model,
    view    = view,
    update  = update
  }
```

El modelo es un estructura definida por el programador la cual deberá tener todo lo necesario para poder representar el estado de la aplicación en todo momento, se lo puede pensar como una foto del programa en un cierto momento.

La función de update puede ser declarada como más convenga pero generalmente recibe como parámetros un mensaje y el modelo. Los mensajes deben ser definidos según las acciones que el usuario pueda realizar. Dados estos parámetros la función deberá evaluarlos y retornar el modelo actualizado.

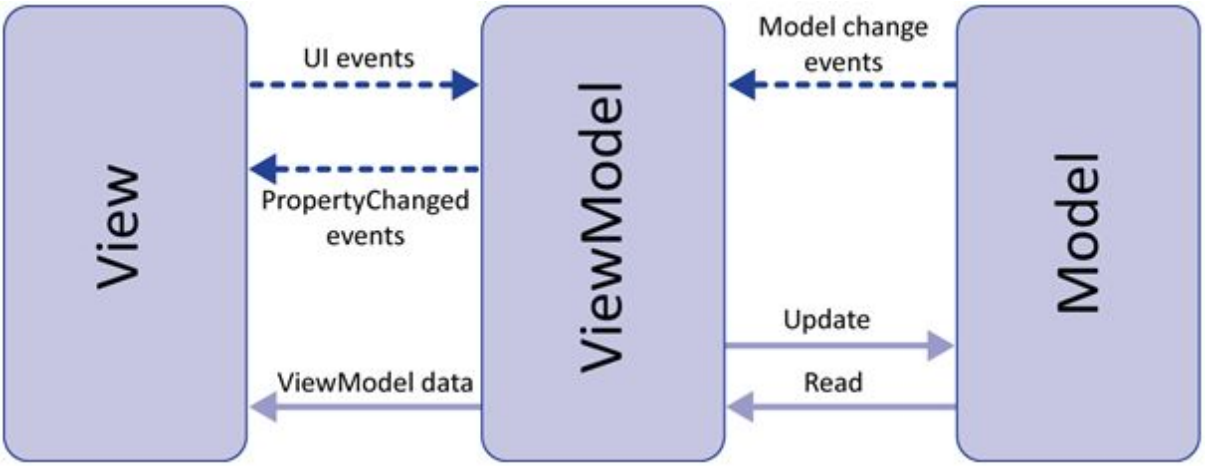
```
type Msg = Message1 | … | MessageN

update : Msg -> Model -> Model
update msg model =
  ...
```

Por último la función view, recibe como parámetro el modelo y lo traduce a una vista, particularmente en Elm lo transcribe a Html.

```
view : Model -> Html Msg
view model =
  ...
```

Experiencia personal

Hace dos años que desarrollo aplicaciones en Swift (iOS) aplicando el patrón MVVM y es muy común que en aplicaciones grandes donde hay varios niveles de profundidad al navegar se generen inconsistencias con el patrón, dificultad para coordinar eventos o acciones, se generan side effects no deseados o condiciones de carrera y se dificulta la comunicación entre view controllers padres e hijos.



En cambio si se siguiera un manejo de eventos o mensajes orientado al propuesto en Elm, se podrían evitar la mayoría de estos problemas.

Adoptando una estructura de mensajes de tipo árbol, es decir que los mensajes solo fluyen desde los padres hacia los hijos. Además, utilizando estado y datos inmutables se puede representar cada momento de la aplicación dado un mensaje. Tambien se podria guardar el estado en un stack para posteriormente analizarlo a modo de debug o agilizar tareas de QA.

Una de las librerías que comencé a utilizar para evitar los problemas antes mencionados es Portal la cual está inspirada en la arquitectura de Elm pero con sutiles cambios para poder adaptarla a aplicaciones para iOS. Con esta nueva forma de programar en Swift, se pueden observar todos los cambios de estados de la aplicación en un switch principal donde según qué mensaje se reciba se aplicarán ciertos cambios al estado o modelo y luego en la función de view se puede ver claramente que vista se va a visualizar para ese frame.

Buenas prácticas

1. **Siempre comenzar declarando los tipos a utilizar:** Por ejemplo al definir una función, primero pensar cuáles son los parámetros a utilizar y que retornara. Al hacer esto, ayuda a familiarizarse con el mundo funcional y a pensar antes de escribir una función.

2. **Usar typealias:** Abusar del uso de typealias puede ser algo bueno.

   Ejemplo:

```
type alias Row = Int
type alias Col = Int
type alias Size = Int
type alias Distance = Int
type alias Point = (Row, Col)
```

   Es mucho más intuitivo ver una definición de la siguiente forma:
   Size -> (Point -> Distance) -> Point
   Que en cambio usar:
   Int -> ((Int, Int) -> Int) -> Int

3. **Que los estados inválidos no sean posible:** Si bien a veces no es fácil o no es posible, no hay que generar casos extras que no van a representar un estado valido de la aplicación, siempre es mejor pensar mejor el modelo y como se va a representar la aplicación.

Repetición de código

Al tratarse de un juego de tablero es común repetir mucho código para recorrer el tablero y aplicar funciones como por ejemplo el uso de maps y recude o folds.

Por lo cual, se generó una serie de funciones para generalizar todo este comportamiento y así agilizar el desarrollo del resto de las funciones específicas del juego.

Map

```
type alias Transform a b = a -> b

matrixMap : Transform a b -> Matrix a -> Matrix b
matrixMap transform board =
   List.map (\tiles -> List.map transform tiles) board
```

Indexed map

```
type alias IndexedTransform a b = Row -> Col -> a -> b

matrixIndexedMap : IndexedTransform a b -> Matrix a -> Matrix b
matrixIndexedMap transform board =
    List.indexedMap (\row tiles -> List.indexedMap (transform row)
tiles) board
```

Reduce

```
type alias InnerReduce a b = a -> b -> b

type alias OuterReduce a = a -> a -> a

matrixReduce : InnerReduce a b -> OuterReduce b -> b -> Matrix a -> b
matrixReduce innerReduce outerReduce initialValue matrix =
  List.foldr (\row acum -> outerReduce (List.foldr innerReduce
initialValue row) acum) initialValue matrix
```

Modelo

La parte más importante es el modelo, siempre es conveniente dedicarle un tiempo a analizar y pensar esta sección antes de empezar a programar. Un error en el modelo y puede generar estados inconsistentes, o que el código sea mucho más engorroso simplemente por plantear mal el modelo.

En el caso de Blob wars, se eligió representar el modelo de la siguiente forma:

```
type alias Model =
   {
      board           : Board,
      selected        : Maybe Point,
      match           : Match,
      turn            : Turn,
      amountOfRed     : Amount,
      amountOfBlue    : Amount
   }

type alias Row = Int

type alias Col = Int

type alias Point = (Row, Col)

type alias Board = List(List(Tile))

type alias Amount = Int

type Tile = Empty
               | PossibleMovement
               | Selected Blob
               | Unselected Blob

type Blob = Blue
                | Red


type Match = BlueWin
                  | RedWin
                  | Tie
                  | InProgress

type Turn = BlueTurn
                | RedTurn
```

Como es un juego de tablero lo ideal es representarlo con una lista de listas de los posibles elementos a representar en el tablero, en este caso Tiles.
Tiles se definió utilizando union types, es decir que pueden ser Empty, PossibleMovement, Selected Blob o Unselected Blob. Donde Blob puede ser Red o Blue, los colores de los jugadores.

Además se agregaron al modelo otros elementos para poder representar la vista, como la cantidad de fichas de cada jugador, el turno (rojo o azul), el estado del juego (en progreso, empate, ganó azul o gano rojo).

Por último, es necesario saber cual es la posición de la última ficha seleccionada para realizar el movimiento en el caso correspondiente, ya que el único dato que tenemos es la posición donde se hizo click.

Funcion de Update

Es útil pensar a la aplicación como una máquina de estados la cual se analizará en la función de update. Por ejemplo en nuestro caso solo soportamos dos posibles mensajes:

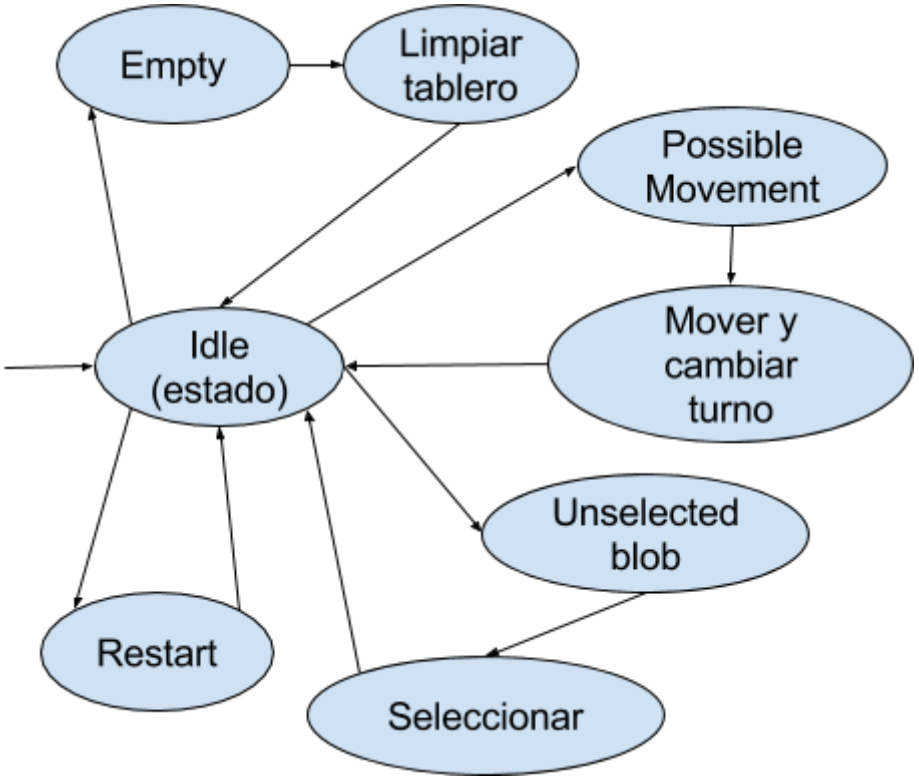
1. Click Point
2. Restart

Con el primero indicamos en qué casillero hizo click el usuario y con el segundo indica que hizo click en el botón de reiniciar el juego.

En el caso del mensaje de restart solo hace falta volver al estado inicial, en el caso del mensaje Click se debe analizar en qué Tile se hizo click:

- Unselected blob: Se hizo click en una mancha sin seleccionar es decir que hay que seleccionarla y mostrar los casilleros a donde se puede mover.
- Empty: Solo hace falta limpiar el tablero. (Deseleccionar las manchas y cambiar los PossibleMovement por Empty)
- PossibleMovement: Es importante aclarar que si se llegó a este estado es porque anteriormente se seleccionó una mancha y se mostraron las fichas de PossibleMovement, es decir que podemos realizar el movimiento de la ficha y cambiar de turno sin hacer ninguna validación. (Por esto es importante no representar casos inválidos)
- El resto de los casos no generan ningún cambio en el modelo.

La máquina de estado se la puede pensar de la siguiente forma:



Se puede ver que en este caso y con el modelo propuesto es muy fácil hacer la estructura de la función de update y aplicar la actualización que corresponda.

Ahora bien pasando esta máquina de estado que pensamos previamente a la función de Update obtuvimos el siguiente código:

```
type Msg = Click Point | Restart

update : Msg -> Model -> Model
update msg model =
   case Debug.log "msg" msg of
      Click point ->
         case getTile point model.board of
            Just tile ->
               case Debug.log "selected tile" tile of
                  Empty              -> model |> clear
                  PossibleMovement   -> model
                                           |> handleBlobMovement point
                                           |> clear
                                           |> updateAmountOfblobs
                                           |> checkGameStatus
                  Unselected blob    -> model
                                           |> clear
                                           |> handleBlobSelection point
                  _                  -> model
            Nothing -> Debug.log "This should never happend!!!" model
      Restart -> initialModel
```

Si comparamos con la máquina de estado anteriormente planteada, podemos ver que se representan exactamente los estados descritos con la salvedad de que la función total getTile podría retornar Nothing (esto no debería suceder, a no ser que haya algún error en el código por ejemplo que los casilleros tengan mal asignado su mensaje de Click Point).

Si el mensaje recibido es Restart simplemente retornamos nuevamente el estado inicial (con el cual inicializamos el modelo). En otro caso, como bien se explicó anteriormente se analiza en qué Tile se hizo click, analizemos todos los tres posibles casos:

1. Empty: Al recibir que se hizo click en un casillero vacío, se tengan o no blobs seleccionadas, se debe deshacer todas las selecciones que haya hecho el jugador para esto al modelo le aplicamos la función clear.

```
clear : Model -> Model
clear model =
   { model |
     board = model.board
                  |> matrixMap possibleMovementToEmpty
                  |> matrixMap selectedToUnselected,
     selected = Nothing
   }
```

La función clear recibe el modelo propiamente dicho, cambiando el valor de selected (última blob seleccionada, esto se mantiene para saber qué blob mover en caso de hacer click en un casillero de tipo PossibleMovement) a Nothing. Además, mapea el tablero con las siguientes funciones:

- possibleMovementToEmpty: Recibe un Tile y si esta es un PossibleMovement retorna otra Tile de tipo Empty sino retorna el Tile recibido por parámetro.

```
possibleMovementToEmpty : Tile -> Tile
possibleMovementToEmpty tile =
   case tile of
      PossibleMovement -> Empty
      _ -> tile
```

- selectedToUnselected: Similar a possibleMovementToEmpty cambia un Tile de tipo Selected a Unselected.

```
selectedToUnselected : Tile -> Tile
selectedToUnselected tile =
   case tile of
      Selected Blue -> Unselected Blue
      Selected Red -> Unselected Red
      _ -> tile
```

Como se puede ver, al tener una implementación genérica de matrixMap dada cualquier lista de listas o tablero podemos de forma muy fácil y en pocas líneas mapear sus valores.

2. PossibleMovement: Si se hizo click en un casillero marcado como movimiento posible quiere decir que en un estado previo se hizo click en un blob marcándolo para moverla (campo selected del modelo), los pasos a seguir en este estado es mover la mancha correspondiente, limpiar el tablero (ya que pueden quedar casilleros del tipo PossibleMovement y el blob original seleccionado), actualizar la cantidad de manchas que hay de cada jugar en el tablero y verificar si el juego termino. Las funciones utilizadas son las siguientes:

- handleBlobMovement: La cual recibe un point a donde mover el blob seleccionado y el modelo para actualizarlo.

Dado el tablero primero se mancha un área, es decir que invierte el color de las blobs en un radio 2 con centro el punto indicado, posteriormente se hace un swap entre el casillero destino y el original y finalmente se hace un update en el casillero donde se hizo la selección del blob que se movió dejando el original si el movimiento fue de 1 casillero o sino se deja el casillero vacío en el caso de un movimiento de 2 casilleros.

```
handleBlobMovement : Point -> Model -> Model
handleBlobMovement point model =
  let
    selectedPoint =
      case model.selected of
        Just p -> p
        Nothing -> Debug.log "This should never happend!!!" (0, 0)
    tile = unwrapTile (getTile selectedPoint model.board)
  in
    case model.selected of
      Nothing -> model |> clear
      Just lastSelectedpoint ->
        {  model |
           board = model.board
                     |> stainArea point model.turn
                     |> swap lastSelectedpoint point
                     |> updateLastSelection lastSelectedpoint point
tile,

           selected = Nothing,
           turn = changeTurn model.turn
        }

stainArea : Point -> Turn -> Board -> Board
stainArea point turn board =
  let
    stain row col tile =
      if calculateDistance point (row, col) == 1 then
        stainTile turn tile
      else
        tile
  in
    matrixIndexedMap stain board

updateLastSelection : Point -> Point -> Tile -> Board -> Board
updateLastSelection p1 p2 originalTile board =
  let
    distance = calculateDistance p1 p2
    update row col tile =
      if (row, col) == p1 then
```

```
        if distance == 2 then
          Empty
        else
          originalTile
      else
        tile
  in
    matrixIndexedMap update board
```

- clear : Funcion explicada en el caso anterior.

- updateAmountOfBlobs: Actualiza en el modelo las cantidad de manchas para esto primero reduce el tablero contando las manchas de un color.

```
updateAmountOfblobs : Model -> Model
updateAmountOfblobs model =
    {  model |
       amountOfRed = countBlobs Red model.board,
       amountOfBlue = countBlobs Blue model.board
    }


countBlobs : Blob -> Board -> Amount
countBlobs blob board =
  let
    check blobType = if blobType == blob then 1 else 0
    innerCount tile acum =
      case tile of
        Unselected blobType -> acum + check blobType
        Selected blobType -> acum + check blobType
        _ -> acum
    outerCount amount acum = amount + acum
  in
    matrixReduce innerCount outerCount 0 board
```

- checkGameStatus: Actualiza en el modelo, si el juego está en progreso o si hubo empate o algún ganador. Simplemente verifica si algún jugador se quedó sin manchas o si al tablero le quedan espacios vacíos si esta condición se cumple actualiza el campo match por algún ganador o empate según corresponda.

```
checkGameStatus : Model -> Model
checkGameStatus model =
  if isBoardCompleted model.board then
    { model |
      match = updateWinner model
```

```
    }
  else if model.amountOfRed == 0 then
    { model |
      match = BlueWin
    }
  else if model.amountOfBlue == 0 then
    { model |
      match = RedWin
    }
  else
    model

isBoardCompleted : Board -> Bool
isBoardCompleted board =
  let
    innerReduce tile completed =
      case tile of
        Empty -> False
        _ -> completed
    outerReduce rowCompleted matrixCompleted = rowCompleted &&
matrixCompleted
  in
    matrixReduce innerReduce outerReduce True board

updateWinner : Model -> Match
updateWinner model =
  if model.amountOfRed < model.amountOfBlue then
    BlueWin
  else if model.amountOfRed > model.amountOfBlue then
    RedWin
  else
    Tie
```

3. Unselected blob: En este caso lo único que hay que hacer es limpiar el tablero, seleccionar la mancha y mostrar los movimientos posibles para que el usuario pueda realizar un movimiento con el siguiente click que haga o cambie de selección. Las funciones involucradas son las siguientes:

- clear: Anteriormente explicada.

- handleBlobSelection: Esta función es la encargada de seleccionar un blob para ello delega el trabajo a la función updateModelOnSelection pero antes verifica que el tile seleccionado no es Nothing y que el turno sea el correcto en caso contrario retorna el mismo modelo sin cambios. La función updateModelOnSelection verifica si el modelo tiene una mancha previamente seleccionada si este es el caso la deselecciona y selecciona la mancha correcta mostrando sus posibles movimientos y actualiza esta información en el modelo del mismo modo actúa si no hay

previamente una mancha seleccionada pero en este caso no hace falta deseleccionar ninguna mancha. El código es el siguiente:

```
handleBlobSelection : Point -> Model -> Model
handleBlobSelection point model =
  let
    tile = getTile point model.board
  in
    case (tile, model.turn) of
      (Just (Unselected Red), RedTurn)  -> updateModelOnSelection point
model
      (Just (Unselected Blue), BlueTurn) -> updateModelOnSelection point
model
      (_, _)             -> model

updateModelOnSelection : Point -> Model -> Model
updateModelOnSelection point model =
  let
      selectBoard board =
         board
            |> updateBlob point selectTile
            |> updateMovements point showMovement

      unselectboard lastSelectedpoint board =
         board
            |> updateBlob lastSelectedpoint unselectTile
            |> updateMovements lastSelectedpoint unshowMovement
  in
    case model.selected of
      Nothing ->
        {  model |
           board = selectBoard model.board,
           selected = Just point
        }
      Just lastSelectedpoint ->
        {  model |
           board = model.board
                     |> unselectboard lastSelectedpoint
                     |> selectBoard,
           selected = Just point
        }
```

Las funciones utilizadas en el código de updateModelOnSelection son las siguientes:

- updateMovements: La cual verifica si los casilleros están a rango dos del punto recibido por parámetro y le aplica una transformación (las que utilizo son para mostrar o ocultar los movimientos posibles)

```
updateMovements : Point -> (Tile -> Tile) -> Board -> Board
updateMovements point transform board =
   let
      update row col tile =
         if isOnMoveRange point (row, col) then
            transform tile
         else
            tile
   in
      matrixIndexedMap update board

isOnMoveRange : Point -> Point -> Bool
isOnMoveRange p1 p2 =
   let
      distance = calculateDistance p1 p2
   in
      distance == 1 || distance == 2
```

- updateBlob: Dado un punto y una transformación actualiza el casillero correspondiente a ese punto con la transformación provista.

```
updateBlob : Point -> (Tile -> Tile) -> Board -> Board
updateBlob point transform board =
   let
      update row col tile =
        if point == (row, col) then
            transform tile
        else
           tile
   in
      matrixIndexedMap update board
```

Función view

Esta función recibe el modelo y lo traduce a html, para ello se verifica en que estado esta el juego. Si el juego termino se muestra la página indicando que el hay un ganador o que terminó en empate, en caso contrario se muestra la vista indicando quien tiene que jugar.

```
view : Model -> Html Msg
view model =
  let
    restartButton = div [class "restartButton"] [button [onClick
Restart] [text "Restart"]]
  in
   case model.match of
     BlueWin ->
        defaultViewWithContent ([div [class "header"] [div [class
"left"] [h2 [] [text "Blue wins!"]], amountToHtml model.amountOfBlue
model.amountOfRed], table [] (boardToHtml model.board), restartButton])
     RedWin ->
        defaultViewWithContent ([div [class "header"] [div [class
"left"] [h2 [] [text "Red wins!"]], amountToHtml model.amountOfBlue
model.amountOfRed], table [] (boardToHtml model.board), restartButton])
     Tie ->
        defaultViewWithContent ([div [class "header"] [div [class
"left"] [h2 [] [text "Tie!"]], amountToHtml model.amountOfBlue
model.amountOfRed], table [] (boardToHtml model.board), restartButton])
     InProgress ->
        defaultViewWithContent ([div [class "header"] [turnToHtml
model.turn, amountToHtml model.amountOfBlue model.amountOfRed], table []
(boardToHtml model.board), restartButton])
```

Cada parte del modelo tiene una función para traducir a formato html y que de esta forma sea más sencillo y legible el código. Por ejemplo la función que traduce el tablero es la siguiente:

```
boardToHtml : List(List(Tile)) -> List(Html Msg)
boardToHtml board =
  let
    update row col tile =
      case tile of
         Empty            -> td [onClick (Click (row, col)), class "tile
empty"] [text ""]
         PossibleMovement -> td [onClick (Click (row  , col)), class
"tile possibleMovement"] [text ""]
         Unselected blob ->
            case blob of
               Red        -> td [onClick (Click (row, col)), class "tile
unselected"] [img [src "./Assets/red_blob.png"] []]
               Blue       -> td [onClick (Click (row, col)), class "tile
unselected"] [img [src "./Assets/blue_blob.png"] []]
         Selected blob ->
            case blob of
               Red        -> td [onClick (Click (row, col)), class "tile
selected"] [img [src "./Assets/red_blob_selected.png"] []]
               Blue       -> td [onClick (Click (row, col)), class "tile
selected"] [img [src "./Assets/blue_blob_selected.png"] []]
  in
    List.map (\tiles -> tr [] tiles) (matrixIndexedMap update board)
```

Es importante notar que los elementos td tiene un acción onClick la cual envía el mensaje click(row, col) el cual será recibido por la función update para actualizar el modelo.

El resto de las funciones se pueden encontrar en el código del trabajo práctico.

Problemas en el desarrollo

El principal problema que surgió durante el desarrollo del juego, fue plantear el modelo de una forma compleja y no hacer funciones genéricas que eviten repetir código.

Anteriormente el modelo planteado fue el siguiente:

```
type alias Point = (Int, Int)

type alias Model =
  { board : List(List(Tile))
  , lastAction : Point
  , winner : Winner
  , turn : Turn
  }

type Tile = Empty (Int, Int) | RedBlob (Int, Int) Bool | BlueBlob (Int, Int) Bool |
PossibleMovement (Int, Int)

type Winner = BlueWin | RedWin | MatchInProgress

type Turn = BlueTurn | RedTurn
```

Como se puede ver cada casillero tiene un identificador (Int, Int) para poder distinguirlo en todo momento lo cual no hace falta ya que se puede usar la función IndexMap. Además semánticamente es incorrecto ya que un casillero no debería porque saber en qué posición del tablero se encuentra.

Por otro lado, los tipos BlueBlob y RedBlob además de un identificador reciben un valor booleano indicando si están seleccionadas o no. Semánticamente no está mal plantearlo de esta forma pero es mucho más prolijo distinguir entre dos tipos Selected y Unselected que reciban un ficha roja o azul.

Se puede apreciar que la función de update tenía muchos más casos y cabe destacar que esta función de update está extraída de uno de los primeros commits del repositorio donde se aloja el código del juego por lo tanto esta función si se hubiera seguido por este camino debería ser más compleja y grande ya que la lógica del juego no estaba terminada.

```
type Msg = Click Point
```

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Click (x, y) ->
      case getTile x y model.board of
        Just tile ->
          case tile of
            BlueBlob (x, y) False -> handleBlobSelection x y model
            RedBlob (x, y) False -> handleBlobSelection x y model
            Empty (x, y) -> { model | board = model.board |> clearPossibleMovements }
            PossibleMovement (x, y) -> handleMovement x y model
            BlueBlob (x, y) True -> { model | board = model.board |> clearPossibleMovements
}
            RedBlob (x, y) True -> { model | board = model.board |> clearPossibleMovements }
```

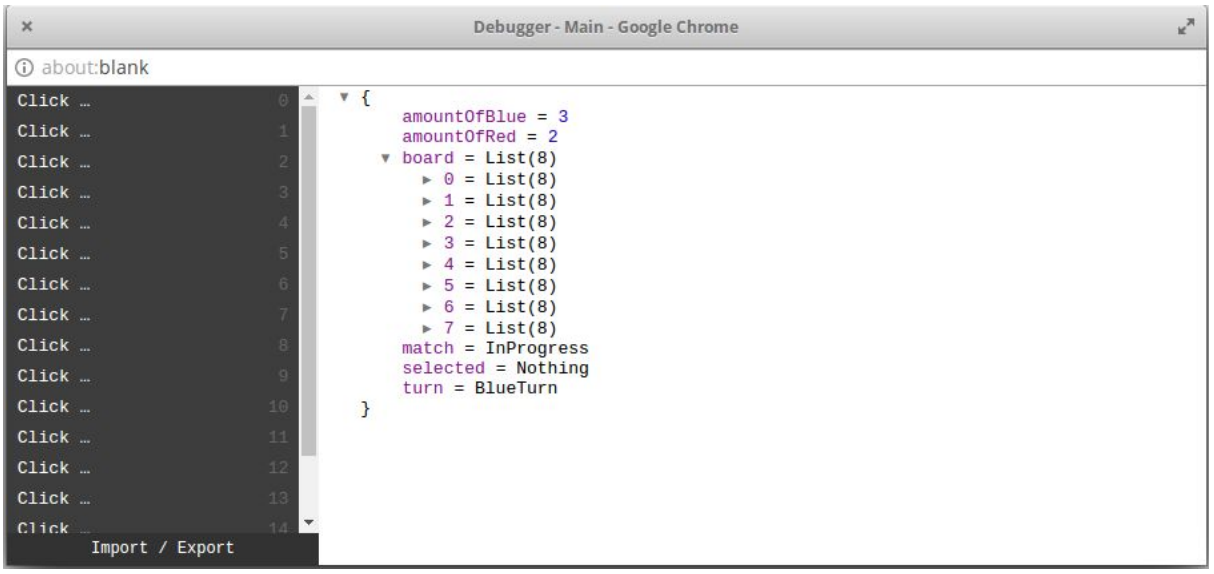
Además, al no haber usado typealiases era muy complicado recordar que significado cada Int o Bool. En cambio con el modelo actual y el abuso del uso de typealias se puede tener una idea más acertada de que significa un parámetro de cualquier función sin tener que contextualizarla o leer su código.

El otro punto fuerte que forzó a cambiar el modelo por el actual, es la falta de uso de funciones genéricas. Por cada funcionalidad nueva se estaba utilizando una función específica lo cual hacía inmantenible el modelo y por lo tanto el juego presentaba varios bugs que se hacían imposible de debuguear y detectar con facilidad.

Debug

Una de las herramientas más útiles que tienen Elm es su debugger, el cual permite tener la lista de todos los estados que pasó nuestra aplicación con las opciones de exportarlo e importarlo en otro momento para evitarnos volver a generar el mismo estado solo para debuguear un caso en particular.

Para esto es necesario utilizar el comando Elm-Reactor y abrir en un explorador localhost:8000 seleccionando el archivo principal de nuestro código.



Como podemos ver en la imagen, en la sección lateral izquierda se encuentra la lista de todos los estados que fuimos pasando pudiendo seleccionarlos y cambiar el estado de la aplicación a el seleccionado para poder visualizarlo y ver los valores del modelo en ese momento.

Además, podemos ver las opciones de importar y exportar, lo cual nos permite guardar un archivo con todos los estados y cargarlos en otro explorador por ejemplo para ver si esa funcionalidad se comporta de forma correcta en distintos exploradores.

Build

Para compilar la aplicación es necesario ejecutar el siguiente comando:

```
elm-make --output target/elm.js src/Main.elm
```

O correr el escript ./build que se encuentra en la raíz del repositorio.

Este comando compila el archivo Main.elm y sus dependencias generando un archivo de salida en formato JS.

Se decidió compilar a JS, ya que se utilizó CSS y para esto es necesario escribir nuestro propia página home.html que se vincula a la hoja de estilo y carga por javascript la aplicación contenida en elm.js. Además, de cargar los recursos necesarios.

Ejemplo de home.html:

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="UTF-8">
  <title>Blob wars</title>
  <link href="https://fonts.googleapis.com/css?family=Coiny" rel="stylesheet">
  <link rel="stylesheet" type="text/css" href="style.css" media="screen"/>
</head>

<body id="body" style="-moz-user-select: none; -webkit-user-select: none;
-ms-user-select:none; user-select:none;-o-user-select:none;" unselectable="on"
onselectstart="return false;" onmousedown="return false;">
</body>

<script type="text/javascript" src="target/elm.js"></script>
<script type="text/javascript">
  var node = document.getElementById('body');
  var app = Elm.Main.embed(node);
</script>

</html>
```

Por último abrir home.html en un browser.

Heroku

El juego se encuentra aloja en un aplicación en heroku se la puede probar en el siguiente link:

http://blob-wars.herokuapp.com/

Vale la pena mencionar que al compilar se genera un archivo JS y se utiliza un home.html para mostrar el juego tal como se mencionó en la sección anterior, por lo tanto para poder integrar este juego con heroku fue necesario utilizar un archivo php wrapper con la siguiente línea de código:

```
<?php include_once("home.html"); ?>
```

De esta forma Heroku reconoce la aplicacion como una aplicacion php y no hace falta realizar ningún otro cambio.

Conclusión

La detección e indicación de errores en tiempo de compilación es muy amigable y fácil de corregir, además de la robustez que esto genera. Ya que Elm es estáticamente tipado, evita errores en tiempo de ejecución facilitando el desarrollo.

Se noto que es sencillo detectar repetición de código similar y abstraerse en funciones más generales de esta forma haciendo más fácil la construcción de otra funciones específicas del juego.

En este caso el modelo fue chico por lo cual ayuda a entender en cada momento lo que está sucediendo, además de que no se generan side effect por ser un lenguaje de programación funcional puro evitando así muchos problemas que como se mencionó en secciones anteriores en iOS son muy frecuentes.

Por último, la arquitectura propuesta por Elm genera un nivel de abstracción excelente para trabajar con programas chicos o juegos sencillos. Pudiendo así plantear una solución inicial en poco tiempo y solo es necesario enfocarse en la programación de funciones específicas del juego sin preocuparnos por la interacción con la UI.