Lib.Model.Game

To begin, note that we are using `Text` in place of `String` for the obvious reasons. Additionally, to support proper blending of colours into the game sprites – to differentiate the teams – we require the use of the `Colour` package.

The `DuplicateRecordFields` language extension is enabled as many of the components of the model have conflicting names, but I feel they are most easily manageable when laid out in a single file as it is done here.

```
{-# LANGUAGE DuplicateRecordFields #-}

module Lib.Model.Game where
  import Data.Text (Text)
  import Data.Colour (Colour)
```

The `Game` is what holds everything together, and serves as the model that is used to represent the current state of the game at any given time. As in the usual fashion, this is an immutable data structure, which when applied an `Action` becomes the next state of our game. In that sense, an `Action` can be simply thought of as a mapping from one state to another.

```
  data Game = Game
    { settings :: Settings
    , scene    :: Scene
    , graphics :: [Sprite]
    }

  newtype Action = Action (Game → IO Game)
```

The `Settings` are pretty self explanatory. They can be set and should affect the player's experience accordingly.

```
  data Settings = Settings
    -- TODO
    { combatAnimations   :: Bool
    , movementAnimations :: Bool
    , autoEnd            :: Bool
    }
```

At a very high level, a game consists of just a few `Scene`s. Each scene has an almost entirely distinct set of relevant updaters to manage its own internal state, so they are broken up and a currently visible scene is stored at the highest level of the `Game` structure.

In this case, the `Menu` scenes are rather similar so they hold a shared record format, the `Menu`, while a `Battle` scene is the more interesting one in which the gameplay actually takes place.

```
data Scene
  = MainMenu Menu
  | PauseMenu Menu Room
  | Battle
    { players      :: [Player]
    , board        :: Board
    , turnCount    :: Int
    }
```

A `Menu` can be thought of, generally, as a set of named `options`, each of which perform a different `Action`. The currently selected option is determined by the `selection` and `submenu` (as menus may have many levels).

```
data Menu = Menu
  { options   :: [(String, Action)]
  , selection :: Int
  , submenu   :: Maybe Menu
  }
```

A `Player` represents a particular team in a battle. There are just two types of `Player`:

`Human` A human controlled player, choosing `Action`s to apply based on the player's inputs.

`CPU` A computer controlled player, choosing `Action`s by following a prescribed `Strategy`.

In either case,

```
data Player
  = Human
```

```
    { name   :: Text
    , colour :: Colour Double
    , units  :: [Unit]
    }
  | CPU
    { strategy :: Strategy
    , colour   :: Colour Double
    , units    :: [Unit]
    }

data Strategy = Strategy -- TODO

data Unit = Unit
  { role      :: Role
  , name      :: Text
  , stats     :: Stats
  , equipment :: Maybe Equipment
  , skills    :: [Skill]
  , owner     :: GameRef Player
  }

data Equipment = Equipment -- TODO

data Stats = Stats
  { mhp :: Int
  , chp :: Int
  , atk :: Int
  , mag :: Int
  , def :: Int
  , res :: Int
  , spd :: Int
  , mov :: Int
  , lck :: Int
  , skl :: Int
  }

data Skill = Skill -- TODO

data Role
  = Tank
  | Infantry
  | Archer
  | Cavalry
  | Flyer
  | CavalryArcher
  | CavalryTank
  | FlyerArcher
  -- TODO

newtype Board = Board
```

```
  { grid :: Grid Tile
  }

data Grid a = Grid
  { width  :: Int
  , height :: Int
  , cells  :: [a]
  }

data Tile = Tile
  { terrain :: Terrain
  , unit    :: Maybe (GameRef Unit)
  }

data Terrain
  = Plain
  | Mountain
  | Forest
  | Swamp
  | River
  | Water
  | Hill
  | Road
  -- TODO

data MapData = MapData
  { board          :: Board
  , startPositions :: [GameRef Tile]
  }

data Sprite = Sprite -- TODO
newtype GameRef a = GameRef (Game → a)
```