Lib.Model

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 where
  import qualified SDL
  import Data.Shape
  import Data.Text (Text)
  import Data.Grid (Grid)
  import Data.Colour (Colour)
  import Data.Set (Set)
  import Lib.RC
  import Foreign.C.Types (CInt)
```

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
    , environment :: Environment
    , saveData    :: SaveData
    , room        :: Room
    , quit        :: Bool
    }

  type Action = Game → IO Game
```

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

The `Environment` is similar in that it simply holds a bunch of information about the game, but these are not set by the user and are instead calculated by

the game much as the rest of the model is.

The `SaveData` is again the same idea, but is intended for information about a particular playthrough of the story mode, holding information like what point in the story has been reached, and what units are available and their stats, among other things. Writing this to a file should be sufficient to save and restore most of the player's game state.

```
data Settings = Settings
  { combatAnimations   :: Bool -- whether combat should be animated
  , movementAnimations :: Bool -- whether movement should be animated
  , autoEnd            :: Bool -- whether the turn should end
      automatically when no actions remain
  , dangerZoneDefault  :: Bool -- whether to show the danger zone by
      default
  }

data Environment = Environment
  { renderOffset :: Point Int -- suggests that everything on the screen
       should be rendered shifted by some distance
  , renderScale  :: Dimension Float -- suggests that everything on the
      screen should be scaled by some amount
  }

data SaveData = SaveData
  { stage :: Int
  , units :: [Unit]
  }
```

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

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

`Cutscenes` and `Dialogs` just run through a predefined set of steps, transitioning to the provided room on completion.

```
data Room
  = MainMenu Menu
  | PauseMenu Menu Room
  | Battlefield Battle
  | Cutscene [Sprite] Room
```

```
    | Dialog [DialogStep] Room

data Battle = Battle
  { players      :: [Player]
  , board        :: Board
  , turnCount    :: Int
  }
data DialogStep = Say Text Text
```

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   :: [(Game → Text, 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 `Actions` to apply based on the player's inputs.

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

In either case, a player chooses a colour to differentiate their units on the battlefield, and has a set of `Units` available to them.

```
data Player
  = Human
    { name   :: Text
    , colour :: Colour Double
    }
  | CPU
    { strategy :: Strategy
    , name     :: Text
    , colour   :: Colour Double
    }
```

```haskell
data Strategy = Strategy -- TODO
```

The `Unit` is probably the most complicated part of the whole model. Each unit represents a single unit on the battlefield, capable of moving around, attacking things, and interacting with others. To determine all the specifics of each unit, they are made up of a number of other components.

The first is the role, which defines what kind of unit they are. "Class" would have been a better name for them, but sadly `class` is a keyword in Haskell, so we'll have to settle for role.

Next is the name, which is pretty self explanatory and exists solely for the player's benefit.

The stats are what determines the unit's abilities in battle and other areas. There are many individual stats which make up the `Stats` record, all of which will be explained elsewhere.

Next is the unit's equipment. That is, what they are holding or wearing. Equipment affects units' stats, as well as their skills.

The units skills help differentiate units, giving them their own strategic values beyond raw stats. There are lots of skills available, so these are listed and described separately.

Finally, a unit has a sprite. Though the sprite exists only for rendering purposes, the unit needs to be able to perform updates on the sprite so that it provides an adequate representation of the unit's state to the player.

```haskell
data Unit = Unit
  { race      :: Race
  , name      :: Text
  , stats     :: Stats
  , equipment :: [EquipmentSlot]
  , skills    :: [Skill]
  , sprite    :: Sprite
  }

data Race
  = Centaur
  -- will this be a stealth game or a mythology game...
  | Thief
  | Assassin

data Stats = Stats
  { mhp :: Int
  , chp :: Int
  , atk :: Int
  , mag :: Int
  , def :: Int
```

```
  , res :: Int
  , spd :: Int
  , lck :: Int
  , skl :: Int
  , mov :: Int
  , snk :: Int
  , vis :: Int
  }

data EquipmentSlot
  = OneOf [EquipmentSlot]
  | Sword
  | Spear
  | Axe
  | Hammer
  | Bow
  | Knife
  | Shield
  | Body
  | Legs
  | Head

data Skill
  = Trample
  | Steal
  | SneakAttack
```

The `Board` is a representation of the actual battlefield. It is composed of a grid of tiles.

A `Tile`, then, represents one space on the `Board`. Each space has a `Terrain`, which affects the units that are passing over it, and sometimes a `Unit` when there should be one at this location.

The actual terrain types are varied and each have different effects which will be explained elsewhere.

```
data Board = Board
  { grid         :: Grid Tile
  }

data TileHighlight = Select | Danger | FogOfWar

data Tile = Tile
  { terrain   :: Terrain
  , unit      :: Maybe Unit
  , highlight :: Set TileHighlight
```

```
  }

data Terrain
  = Plain
  | Mountain
  | Peak
  | Stone
  | Lava
  | Cliff
  | Forest
  | Hill
  | Road
  | Floor
  | Wall
  | ShallowWater
  | DeepWater
  | River
  | Swamp
  | Bridge
```

A `Sprite` exists solely for rendering purposes. Though some elements of the game can be rendered based simply on the state of the `Room`, the more complex items require the use of a `Sprite` – things such as animations and special effects.

```
newtype Sprite = Sprite SpriteEffect

data SpriteBase
  = Static (Key SDL.Texture) (Rectangle CInt)
  | Animation (Key SDL.Texture) [Rectangle CInt] Int
  | AnimationCycle (Key SDL.Texture) [Rectangle CInt] Int

data SpriteEffect
  = Invisible
  | Base SpriteBase
  | Position (Point CInt) SpriteEffect
  | Scale (Dimension CInt) SpriteEffect
  | Movement (Point CInt) (Point CInt) Int Int SpriteEffect
  | ColourBlend (Colour Double) SpriteEffect -- TODO: does this require
      a blend mode
  | Particles (Point CInt) [Point CInt] SpriteEffect
  | Sequence [Sprite]
```

A `GameRef` simply provides a view into the `Game` model allowing a particular element to be quickly retrieved. This provides a sort of weak reference mecha-

nism specific to this model, which may or may not actually be useful when it comes time to implement this stuff. If needed, this can be updated to be a Lens or something.

```
newtype GameRef a = GameRef (Game → a)
```