Lab 10. Algebraic Data Types

Types are an essential component in Haskell. They start with capital letters, like `Int`. We have multiple ways of customizing data types.

Type Aliases

The `type` keyword is used to introduce **type aliases** (new names for **already existing** types). Its main role is to improve the clarity of type annotations by renaming complex types to simpler ones or providing additional context.

```haskell
type Matrix = [[Int]]

type Name = String

type PhoneNumber = String

type PhoneBook = Map Name PhoneNumber
```

Creating Data Types

We can also create **new** types using the `data` keyword. Each type has one or more constructors (separated by `|` in the type's definition). Each constructor can take parameters, and the type itself may take type parameters (enabling the creation of generic types).

```haskell
data BinaryDigit = Zero | One

data Maybe a = Just a | Nothing

data List a = Cons a (List a) | Void

data Both a b = Both a b
```

To create objects of the given types, we use the constructors as if they were functions:

```haskell
bDigitOne :: BinaryDigit

bDigitOne = One

maybeFive :: Maybe Int

maybeFive = Just 5

myIntList :: List Int
myIntList = Cons 1 (Cons 2 Void)
```

Pattern matching can be used on any types defined with `data`.

```
listHead :: List a -> a
listHead (Cons x _) = x
listHead Void = undefined
```

Records

Additionally, we can use `records` to give names to specific paraneters of constructors:

```
data Dog = Dog{ name :: String
              , breed :: String
              , age :: String
              }
```

Using this syntax allows us to create objects like this:

```
myDog = Dog{name="Spike", breed="Golden Retriever", age = 5}
```

Additionally, it defines a 'getter' function for each named field:

```
name myDog -- "Spike"
breed myDog -- "Golden Retriever"
age myDog -- 5
```

The record syntax can be used for types with multiple constructors too. In this case, constructors can also share parameter names as long as the types of parameters with the same name match. Example:

```
data Expr = Atom Int | Add {left:: Expr, right:: Expr} | Subtract {left:: Expr,
right: Expr} | ... -- here, 'left' from 'Add' needs to have the same type as 'left'
from 'Subtract'
```

Newtype

Another, more specialised, way of creating new types is the `newtype` keyword. It is used to create single-constructor, single-parameter types with some memory usage and access optimisations.

```
newtype WithIndex a = WithIndex (a, Int)
```

implementation details about newtype

Type Classes

Type classes represent a similar concept to Java interfaces. They are used to group together types which have a certain behaviour. For example, all types which can be converted to a `String` and printed to the console belong to the `Show` type class.

Do not confuse type classes with the classes from the Object-Oriented paradigm. Type classes are a category of **types**, while OOP classes are a category of **objects**.

We can define a class as such:

```
class MyShow a where
    myShow :: a -> String
```

To enroll a type in this class we create an instance for it:

```
data BinaryDigit = One | Zero


instance MyShow BinaryDigit where
    myShow One = "1"
    myShow Zero = "0"
```

Type classes allow us to place restrictions on parameter types in function definitions, other type class definitions and instaces:

```
showTable :: Show a => [[a]] -> String -- this function can show any table-like list-of-lists as long as the type of the elements is showable itself
showTable table = ...  -- we can freely use 'show' on each "cell" of the table


class  (Eq a) => Ord a  where -- any type 'a' belonging to this class also needs to belong in the 'Eq' class
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a


instance (Show a) => Show [a] where -- we define an instance for lists of all showable types
```

```
    show = ...
```

After the lab, you can take a look at some more cool stuff related to data types in Haskell and functional types in general:

Unboxed types

Lenses

10.1. Tournament

We wish to model a tournament scoring system for 1v1 games such as chess (we will not be referring to the logic of the game itself, only the scorekeeping system). The tournament will have two stages:

- Group stage: Players are placed into $2n2n$ groups. Each player plays against all other players in the same group. The top m players will advance to the elimination stage.
- Elimination stage: Players are paired up to play against each other. For each matchup, the loser gets eliminated, the winner remains. The process is repeated until a single player remains.

10.1.1. Define the datatypes we will need:

```
data MatchResult = ??? -- the result of a match could be a Win, Loss or Draw

data Player = Player {???} -- we care about 3 things: the player's name, elo
(floating point which measures the player's strength), and a list of results of past
games (call it matchHistory)

data Tree a = ??? deriving Show -- A binary tree we will use to represent the
elimination stage of the tournament
```

10.1.2. We want to be able to show a MatchResult and a Player. For this, we could do deriving Show to let GHC derive the implementations, but let's do it ourselves.

```
instance Show MatchResult where

    ???


instance Show Player where

    ???
```

10.1.3. We will want to update player match history after each match.

```
addResult :: MatchResult -> Player -> Player

addResult = ???
```

10.1.4. During the group stage of the tournament, we will require a point-based scoring system. A loss is worth 0 points, a win is worth 1, and a draw 0.5.

```haskell
points :: MatchResult -> Float

points = ???
```

10.1.5. To determine the best players from each group, we will need to determine how many points each player scored.

```haskell
score :: Player -> Float

score = ???
```

10.1.6. For the group stage we will need to sort the players to establish a ranking. The easiest way to do this is to define a order relationship for players, based solely on score (2 players are equal if their scores are equal, similar for greater than and less than relationships)

```haskell
instance Eq Player where

  (==) = ???


instance Ord Player where

  (<=) = ???

  -- Haskell can infer implementations for all other operations if the implementation
for (<=) is given.
```

10.1.7. For simplicity, we will 'simulate' matches between players by looking at their elo. The player with more elo always wins.

```haskell
-- return the players, updated (used for group stages)

playGame :: Player -> Player -> (Player, Player)

playGame = ???

-- ex : playGame (Player "A" 1 []) (Player "B" 2 []) == (Player "A" 1 [Loss], Player
"B" 2 [Win])




-- return the winner, and the players, all updated (used for elimination stage). In
case of a draw, return any of the players instead of the winner.

playGameWithWinner :: Player -> Player -> (Player, (Player, Player))

playGameWithWinner = ???

-- ex : playGameWithWinner (Player "A" 1 []) (Player "B" 2 []) == ((Player "A" 1
[Loss], Player "B" 2 [Win]))
```

10.1.8. You are given a player and a list of players, simulate a match between the player and each of the other ones. Return the player and the list of players updated with the results of the matches.

```haskell
playAll :: Player -> [Player] -> (Player, [Player])

playAll = ???
```

10.1.9. Now, to simulate the group stage:

```haskell
{-

Simulate a single group. Each player will play against all other players exactly
once.

Return a list of all players, updated to reflect the results of the matches. Hint:
use playAll

-}

playGroup :: [Player] -> [Player]

playGroup players = ???


{-

Select the best 'm' players from the given group. (Assume the players already have
their match history updated).

Return 2 lists: one containing the players which were selected and the other
containing all other players.


hint: sort

-}

selectPlayers :: [Player] -> Int -> ([Player], [Player])

selectPlayers players m = ???


{-

Given a list of groups, simulate their matches and select the best 'm' players from
each, and return 2 lists: the selected players and the rest.

-}

playGroups :: [[Player]] -> Int -> ([Player], [Player])

playGroups groups m = ???
```

10.1.10. Now, we play the elimination phase (you can assume that the number of players is a power of two, so we have a full binary tree):

Given a list of players, turn each of them into a binary tree node. Then, until the list contains a single node, group all nodes in the list in pairs, simulate a match between the players at their roots, and replace each pair with a binary tree containing the winning player in the root and the two original trees as children. Only the top (closest to the root) appearence of each player is required to have up-to-date match history.

```
playElimination :: [Player] -> Tree Player

playElimination = ???
```

10.1.11. While the `Tree` representation may be useful, we will require the final ranking to be in a list format. For each player, keep only the top entry (which will be up-to date in terms of match history). (you can consider all player names to be unique).

```
eliminationResults :: Tree Player -> [Player]

eliminationResults = ???
```

10.1.12. Finally, it's time to put everything together:

- split the players into `m` groups (you choose how this is done)
- simulate the group stage, selecting `n` players from each group
- simulate the elimination stage
- return a ranking of all players, sorted in order of descending scores

note: `n` and `m` should be powers of 2 to properly create the elimination phase.

```
runTournament :: [Player] -> Int -> Int -> [Player]

runTournament players n m = ???
```

Example Players

To help you in testing, here is a list of randomly-generated players:

```
players :: [Player]

players = [

    Player {name = "Jill Todd", elo = 69.32222, matchHistory = []},

    Player {name = "Cara Wong", elo = 68.451675, matchHistory = []},

    Player {name = "Travis Dunlap", elo = 49.667397, matchHistory = []},

    Player {name = "Adam Mills", elo = 65.36233, matchHistory = []},

    Player {name = "Josephine Barton", elo = 14.974056, matchHistory = []},

    Player {name = "Erica Mendez", elo = 27.466717, matchHistory = []},

    Player {name = "Derrick Simmons", elo = 11.790775, matchHistory = []},

    Player {name = "Paula Hatch", elo = 80.039635, matchHistory = []},
    Player {name = "Patricia Powers", elo = 61.08892, matchHistory = []},
    Player {name = "Luke Neal", elo = 65.933014, matchHistory = []},
    Player {name = "Jackie Stephenson", elo = 86.00121, matchHistory = []},
    Player {name = "Bernice Nixon", elo = 2.8692048, matchHistory = []},
    Player {name = "Brent Cobb", elo = 39.80139, matchHistory = []},
    Player {name = "Bobbie Sanderson", elo = 81.07552, matchHistory = []},
    Player {name = "Zachary Conner", elo = 63.88572, matchHistory = []},
    Player {name = "Shawn Landry", elo = 7.68082, matchHistory = []},
    Player {name = "Mabel Gentry", elo = 88.13421, matchHistory = []},
    Player {name = "Enrique Ali", elo = 9.568502, matchHistory = []},
    Player {name = "Clara McLaughlin", elo = 60.83427, matchHistory = []},
    Player {name = "Jacqueline Connell", elo = 60.091232, matchHistory = []},
    Player {name = "Jared Morgan", elo = 49.84152, matchHistory = []},
    Player {name = "Lorraine Castaneda", elo = 34.701054, matchHistory = []},
    Player {name = "Robin Hurd", elo = 78.33226, matchHistory = []},
    Player {name = "Vince Dunlap", elo = 63.634525, matchHistory = []},
    Player {name = "Elaine Winter", elo = 34.86934, matchHistory = []},
    Player {name = "Bennie Godfrey", elo = 73.81608, matchHistory = []},
    Player {name = "Gale Britton", elo = 16.05768, matchHistory = []},
    Player {name = "Jeanne Mathis", elo = 34.603416, matchHistory = []},
    Player {name = "Aida Greenwood", elo = 8.308169, matchHistory = []},
    Player {name = "Christian Witt", elo = 80.397675, matchHistory = []},
    Player {name = "Cecelia Dyer", elo = 80.657974, matchHistory = []},
    Player {name = "Edwin Gallagher", elo = 14.976497, matchHistory = []}]
```