COMP5450 - Assessment 2 - Tic-Tac-Toe (Brief v1.1.0)

Formatting and submission instructions

- Download `DecisionTree.hs` and `Base.hs` from the Moodle. Read `Base.hs` carefully alongside this brief. **You must not modify these files, nor submit them.**
- You may use any function from the `Prelude` but **must not import any other libraries** apart from `Control.Concurrent` and the `Base` and `DecisionTree` modules provided here.
- Complete your work in a file named by your user id, e.g., `dao7.hs`.
- All code must be valid Haskell. Any code that produces errors will receive a penalty. Incomplete code must be commented out with an explanation.
- Please put comments in your code to show which question you are answering with each piece of code. This will help the markers.
- For readability, please limit your line lengths to 100 characters maximum as far as possible.
- **The work must be your own. Do not share your code with others, collaborate, or use generative AI.**

Introduction

This assessment is about developing an interactive game of tic-tac-toe (also called 'noughts and crosses') which allows a combination of human and/or AI players, built as a concurrent program. The AI opponent will use a decision tree to make its moves (we will build off Assessment 1).

The game will be played in the terminal, with any human players inputting moves by typing into standard input. A central game server will co-ordinate the players and the game board, providing information to standard output. The game will be a tournament with a number of rounds played until a player decides to stop.

Provided base library The `Base.hs` file provides a basic framework for the game, including:

1. `Player` data type representing the two players;
2. `Board` type synonym defining the game state which is a list of `Maybe Player` where `Nothing` represents an unfilled position, `Just p` represents a position in which player `p` has made a move. A valid game board will have 9 such elements;
3. `Result` data type representing the outcome of a move;
4. `Coordination` data type representing the co-ordination of the tournament;
5. `flipPlayer` for switching between players.

You should import `Base` into your solution.

1 Helper functions (30 marks)

1. Define a function `showBoard` that convert a `Board` into a `String`, displaying the current state of the game, where player one is denoted `X` and player two is `O`. (7 marks)

   For example: `showBoard [Just One, Nothing, Just Two, Nothing, Nothing, Nothing, Nothing, Nothing, Just One]` should return:

   ```
   X _ O
   _ _ _
   _ _ X
   ```

2. Positions on the board will be referred to internally by `Int` values from 0 to 8 inclusive, where 0 is the top-left position, 2 is the top-right position, 6 is the bottom-left position, and 8 is the bottom-right position. (4 marks)

   Define a function that converts a pair of integers into a position in the board, but only if those integers are each in the range 0 to 2 inclusive:

   ```
   toPos :: (Int, Int) -> Maybe Int
   ```

   For example, `toPos (1, 2)` should return `Just 5` and `toPos (3, 2)` should return `Nothing`.

3. Define a function: (3 marks)

   ```
   lookupBoard :: Board -> Int -> Maybe Player
   ```

   That looks up the state of the board at the given position.

   For example, `lookupBoard [Just One, Nothing, Just Two, Nothing, Nothing, Nothing, Nothing, Nothing, Just One] 2` should return `Just Two`.

4. Define a function: (6 marks)

   ```
   addToGameBoard :: Board -> Int -> Player -> Maybe Board
   ```

   which takes a game board and a position and a player and returns a new board with the player added to the position, or returns `Nothing` if the position is already occupied.

   For example:

   ```
   addToGameBoard [Just One, Nothing, Just Two, Nothing,
                   Nothing, Nothing, Nothing, Nothing, Just One] 3 Two
          == Just [Just One, Nothing, Just Two, Just Two,
                   Nothing, Nothing, Nothing, Nothing, Just One]
   ```

5. Define a function: (8 marks)

   ```
   checkWin :: Board -> Maybe Player
   ```

   which checks if either player has won the game, returning `Just` of the player if a player has won, and `Nothing` otherwise. A player wins if they have three in a row horizontally, vertically, or diagonally.

   It is not possible to have more than one winner when playing the game correctly, so you do not need to check for this.

6. Define an IO computation `hline :: IO ()` which prints out a line of 20 hyphens. (2 marks)

2 Game server and co-ordination (30 marks)

A game server will co-ordinate the game between the different players via concurrent programming. The server is responsible for managing the game state, providing output messages, and ensuring the game is played correctly. We will use channels to connect the server and players. Each player will have two channels specific to them:

- A *coordination channel* of type `Chan Coordination` which is used to determine whether to play another game again (`Again`) or whether to stop the tournament (`Stop`);
- A *moves channel* of type `Chan Int` used by the player to send a move to the server.

There will be a further *results channel*, of type `Chan Result`, which is used to communicate results from the server to the players. Thus, in total there will be five channels used.

7. Define a helper function of the following type: (2 marks)

   ```
   select :: Player -> Chan Int -> Chan Int -> Chan Int
   ```

   which returns the second input if the `Player` is `One` and returns the third input if the `Player` is `Two`. This will be useful for alternating which player to receive a move from.

8. Define a helper function that writes a message twice to a channel, of type: (3 marks)

   ```
   writeChanTwice :: Chan a -> a -> IO ()
   ```

9. Define the main `gameServer` function of type: (12 marks)

   ```
   gameServer :: Player -> Board
               -> Chan Int -> Chan Int -> Chan Result -> IO (Maybe Player)
   ```

   with inputs: (1) the player whose turn it is, (2) the current game board state, (3) the moves channel for player one, (4) the moves channel for player two, (5) the results channel. The `gameServer` has IO side effects and computes an outcome `Maybe Player` where `Nothing` represents a draw and `Just p` represents a win for player `p`.

   The game server should do the following:

   (a) Print a horizontal line (`hline`);
   (b) Print a message asking the current player for a move;
   (c) Receive a move from the current player;
   (d) Print a message describing the attempted move;
   (e) Add the move to the game board (`addToGameBoard`):
      i. If the move is invalid, print a notification, write on the results channel twice that the game `Continue`s with the existing game board, and flip whose turn it is (`flipPlayer`) in a recursive call (yes, this is a bit brutal; it's up to the players to choose a position that isn't already occupied!);
      ii. If the move is valid, show the updated board, then check for a win (`checkWin`):
         A. If a player has won, output a message and then report the win on the results channel twice (so both players can receive it), and return this outcome;
         B. If there is no winner and there are possible moves left on the board, report to the players that the game continues (with the new board state) and continue with the game (recursively), flipping whose turn it is;
         C. If there is no winner and there are no moves left, show a 'draw' message, report the draw on the results channel twice, and return the outcome.

   An example play can be seen at the end of this brief which provides examples of the messages to output. You do not need to replicate exactly their words but at least the same meaning.

10. Define a function which is used to initialize the game server and co-ordinate play across multiple rounds of the game, of type: (8 marks)

```
gameServerStart :: Player
             -> (Chan Coordination, Chan Coordination)
             -> (Chan Int, Chan Int)
             -> (Int, Int)
             -> Chan Result -> IO ()
```

with the inputs: (1) the player who will start the next game; (2) a pair of co-ordination channels for each player; (3) a pair of moves channels for each player; (4) a pair of the current scores in the tournament for each player; (5) the results channel.

The function `gameServerStart` should:

(a) Run `gameServer` with an empty board and all its input dependencies;
(b) Inspect the outcome returned from the server;
(c) Report the latest scores for both players;
(d) Print a message asking whether player one wants to play again;
(e) Receive on the coordination channel for player one;
(f) If a new game is not requested (`Stop`), notify player two on its coordination channel (send `Stop`); if a new game is requested (`Again`), notify player two accordingly and then recurse, flipping the start player.

11. A player is represented by a function that takes as input the player designation, a co-ordination channel, a moves channel, and a result channel, and performs some IO computation, i.e., players have type: (5 marks)

```
  Player -> Chan Coordination -> Chan Int -> Chan Result -> IO ()
```

Define a higher-order function `startGame` that takes two players (of the above type) as input and returns `IO ()`. This function should create all the required channels for both players, should start both players as new concurrent processes (`forkIO`) and should then start the game server via `gameServerStart`.

3 Human Player (20 marks)

12. Define a function `parseInput` that takes two string inputs: (4 marks)

```
parseInput :: String -> String -> Maybe Int
```

The first string should be a row number written as `1`, `2`, or `3`, and the second should be a column written `A`, `B`, or `C` (or in lowercase). The function should parse valid inputs into a game board position (you can use `toPos`), returning `Nothing` on invalid inputs.

For example, `parseInput "1" "A"` should return `Just 0` and `parseInput "3" "c"` should return `Just 8` but `parseInput "4" "a"` should return `Nothing`.

13. Define a human player process, i.e., of type: (16 marks)

```
humanPlayer :: Player -> Chan Coordination -> Chan Int -> Chan Result -> IO ()
```

The human player should read two input strings from the user separately, a row `1-3`, and a column `A-C`, in order to calculate its move. It should send its move to the server and then receive from the results channel. If the game is ended (win or draw) the player should then do some co-ordinating whether to keep playing (see below). If the game continues then the player should receive again on the results channel to find out the outcome of its opponent's play. If the game continues then recurse, otherwise if the game is over its time to co-ordinate whether to play again.

At the end of a game, if the human is player one, then input a string to determine a `Coordination` message to send to the server, with `Y` meaning play `Again` and `N` meaning `Stop`. If the player wants to play again then `humanPlayer` should repeat its behaviour otherwise it should finish.

Otherwise, if the human is player two, it should wait to receive a co-ordination message and act accordingly (either stopping or playing again).

Hints: Follow the protocol of the server and make sure the player acts in a 'dual' way. It may also be helpful to split `humanPlayer` into two functions, one that handles playing the game, and one that handles playing the tournament, i.e., with the co-ordination involved for stopping play or not.

If you want to test extensively but haven't completed Part 4 yet, you could make a dummy AI opponent that is like the human player but with no input and always trying to play in the same position (i.e., corner).

4 AI Player (20 marks)

The last part of this assessment is to develop an AI player that uses the decision trees from Assessment 1 to make its moves, adding the results of successive games into a training data set, and retraining a decision tree after each game to improve its play.

Provided decision tree implementation The `DecisionTree.hs` file on the Moodle provides a model solution to Assessment 1 but with some slight adaptation. **If you would like, you may adapt your Assessment 1 according to the changes given in this file and use that instead, otherwise you can just use what is provided by importing `DecisionTree`.**

The interface you will need to use comprises the same data types and type definitions as in Assessment 1, but with a slight change to `infer` and a new function `addRow`:

- `addRow :: TrainingDataSet -> (Row, Label) -> TrainingDataSet`

  Adds a row to a training data set;

- `infer :: DecisionTree -> Header -> Row -> Maybe (Label, Float)`

  From a decision tree, a header for the data set, and a row, infer the label with its confidence values (a float);

- `learnTree :: TrainingDataSet -> (TrainingDataSet -> Attribute) -> DecisionTree`

  Learn a decision tree from a training data set;

- `bestGain :: TrainingDataSet -> Attribute`

  Choose the next attribute based on maximising information gain

14. Define an AI player, with the same type as the human player. The AI player should keep track of: the current game board, a `TrainingDataSet` containing a record of the games played so far, and a decision tree trained on its current knowledge of past games. Attributes in the data-set/decision-tree are just game board positions (as a string). We can thus define the header as `map show [0..9]`. Values of the attributes are just the string representing the `Maybe Player` value at each position in the game board. The label can be interpreted as `Yes` to mean the AI player won and `No` to mean it lost or there was a draw. (20 marks)

Your AI player should decide which move to take by querying its decision tree with a game board corresponding to making each of the possible remaining moves and finding which has the maximum likelihood of leading to a win or the minimum likelihood of leading to a loss or draw; use `infer`, which now returns a probability (confidence) with potential game boards, and then rank the moves, taking the best one.

After each game, extend the AI players' training data set with the outcome of the last game and relearn a decision tree (using the provided 'best gain' attribute selector) to use for the next game.

At the end of every game the AI player must also wait for a co-ordination message to decide whether to play again or not.

A Example game play

```
--------------------
Player One, enter your row (1-3) and then column (A-C):
1
a
Player One attempted to move to 0
X _ _
_ _ _
_ _ _
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 8
X _ _
_ _ _
_ _ O
--------------------
Player One, enter your row (1-3) and then column (A-C):
2
b
Player One attempted to move to 4
X _ _
_ X _
_ _ O
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 7
X _ _
_ X _
_ O O
--------------------
Player One, enter your row (1-3) and then column (A-C):
2
b
Player One attempted to move to 4
Invalid move!
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 6
X _ _
_ X _
O O O
```

```
Player Two wins!
--------------------
Score Board: Player One = 0 | Player Two = 1
--------------------
Play again? Player One needs to say Y/N?
Y
Again!
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 8
_ _ _
_ _ _
_ _ O
--------------------
Player One, enter your row (1-3) and then column (A-C):
1
a
Player One attempted to move to 0
X _ _
_ _ _
_ _ O
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 7
X _ _
_ _ _
_ O O
--------------------
Player One, enter your row (1-3) and then column (A-C):
1
b
Player One attempted to move to 1
X X _
_ _ _
_ O O
--------------------
Player Two, enter your row (1-3) and then column (A-C):
Player Two attempted to move to 6
X X _
_ _ _
O O O

Player Two wins!
--------------------
Score Board: Player One = 0 | Player Two = 2
--------------------
Play again? Player One needs to say Y/N?
N
End of tournament!
```