Announcements

"...there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated there are no *obvious* deficiencies. The first method is far more difficult."

— Tony Hoare, 1980 ACM Turing Award Lecture

Review

- `type` defines a type name as an abbreviation for other types
- `data` defines new data structures (and a type) and constructors / deconstuctors
- `IO t` is the input/output monad
- `do` can be used to sequence input/output operations

Games

- Players make actions
- Games take actions and update state of game, perhaps finishing.
- A player, given state of game and what actions are available, selects an action
- A game: function from action and state into a result.
  A result is either:
  - End of game with result = value for player, (e.g., +1 for win, 0 for draw/tie, -1 for loss) and a new starting state, or
  - Game continues with a new state

Count Game

- players take turns choosing numbers, e.g, in range [1..9]
- if the sum of the numbers chosen exceeds some break value, the player loses

What information does a state contain?

- current count (sum of all values)

What Actions are available?

- List of numbers not chosen.

See CountGame.hs Play.hs

Game Abstraction

- `type Player = State -> Action`
  A player is given a state of the game and selects a move.
- `type Game = Action -> State -> Result`
  A game takes an action and the state and returns a result
- Result is either
  - the end of the game with a reward and starting state or
  - a continue with a new state

```
data Result = EndOfGame Double State
            | ContinueGame State
```

- As part of the state is internal state and the available moves:

```
data State = State InternalState [Action]
```

- Generic players should not make any assumptions about the form of the internal state or the actions.

Interacting with user (Play.hs)

Two mutually recursive functions:

```
person_play :: Game -> Result -> Player ->
                    TournammentState -> IO TournammentState
```

- The person's play: takes `Game`, the `Result` of the previous play (the computer's move, except initially), the computer `Player`, and updates the tournament state
- If the result (of previous move) is to continue, asks the person for an action, computes result, then it's the computer's turn
- If the result (of previous move) is the end of the game, the value of the result is the value for the computer, and so needs to be negated for the person. Start again.

```
computer_play :: Game -> Result -> Player ->
                    TournammentState -> IO TournammentState
```

- Similar except, it calls the Player to get the next action. No negation needed.

Minimax

For two-player zero-sum games – the value for one player is the negation of the value of the other player – the optimal strategy can be computed using:

- At the end of the game, the result provides the value for the player that ended the game.
- When an action is chosen, the value for that player is the negation of value for the opposing player of the resulting state.
- An agent gets to choose the action with the maximum value for them.
- Use functional definition of a game to simulate game
- Select move with best evaluation function
  ..... then it's the opponents turn to select their best move
  ..... until end of game

Minimax

- `type Game = Action -> State -> Result`
  The game can be asked hypothetical questions about the result of a move. (Because it is functional.)
- A game has a value for each player at the end of the game.
- Assumes a two-player zero-sum game:
  The value for a player is the negative of the value of the opponent.
- `minimax:: Game -> State -> (Action, Double)`
  `valueact :: Game -> State -> Action -> Double`
  `value:: Game -> Result -> Double`
- Minimax takes game and a state and returns (action,value) for the best move for agent playing,
  assuming a move is available
- `type Player = State -> Action`
- `mm_player:: Game -> Player`
  `mm_player game state = fst ( minimax game state)`
- See `Minimax.hs` (run the test cases at bottom)