Announcements

- Project proposals due by at start of next week. Talk to a TA!

"Pascal [Java] is for building pyramids .... Lisp [Haskell] is for building organisms – .... The organizing principles used are the same in both cases, except for one extraordinarily important difference: . . . Lisp programs inflate libraries with functions whose utility transcends the application that produced them. In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation.

– Alan J. Perlis, Foreword to "Structure and Interpretation of Computer Programs", 1985, 1996

Review

- `type` defines a type name as an abbreviation for other types
- `data` defines new 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
- `instance` puts types into classes

How to write a (Haskell) Program

- To solve a complex problem, break it into simpler problems.
- Motivate/design top-down
- Build bottom-up.
- Write a clear specification (API / intended interpretation) for each component; program to that specification.
- Ensure each part is modular, debuggable, with clear meaning.
- Test early and test often. Try to break your program. Try to prove your program is correct.
- Test every function when defined. Every component of a function should be already tested and debugged before use.
- Give the type signature for every function (so the compiler does not suggest bugs in tested code).
- Generalize components so they are as reusable as possible. Make sure you can find a previously written appropriate function when it is the one you want.
- Write functional components as much as possible. Try to minimize IO components.

Games (MagicSum.hs, Play.hs)

Turn-taking 2-player games with win/lose/draw at the end:

- Players make actions
- Games take actions and update state of game, perhaps finishing.
- A player, given state of game, and a list of legal actions, 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

Game Abstraction (See MagicSum.hs Play.hs)

- A player is given a state of the game and selects a move.
  `type Player = State -> Action`
- A game takes an action and the state and returns a result
  `type Game = Action -> State -> 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]
```

Magic Sum Game

- players take turns choosing different numbers in range [0..9]
- first player to have 3 numbers that sum to 15 wins
- draw/tie if run out of numbers to play

What common game is this equivalent to?

Magic square:

2	7	6
9	5	1
4	3	8

Magic sum game is isomorphic to tic-tac-toe.

Magic Sum Game

- players take turns choosing different numbers in range [0..9]
- first player to have 3 numbers that sum to 15 wins
- draw/tie if run out of numbers to play

What information does a state contain?

- (Numbers chosen by self, numbers chosen by opponent)

What Actions are available?

- List of numbers not chosen.

Clicker Question

```
type Action = Int
data State = State ([Action],[Action]) [Action]
         deriving (Eq, Show)
data Result = EndOfGame Double State
            | ContinueGame State
         deriving (Eq, Show)
```

What is not true:

A `State` is both a type and a constructor

B We can compare States with ==

C `EndOfGame` is a function of type `Double -> State -> Result`

D `State([1],[])[4] == ContinueGame(State([1],[])[4])` returns False

E `ContinueGame (State a b)` can be used on the left side of an equality

Clicker Question

```
type Action = Int
data State = State ([Action],[Action]) [Action]
         deriving (Eq, Show)
data Result = EndOfGame Double State
            | ContinueGame State
         deriving (Eq, Show)
```

What happens if the third line is removed?

A Nothing as long as we don't compare or show states

B It gives a compile-time error as Result then cannot be in Eq or Show

C It gives a run-time error if a state is compared or shown

D The proof of Show and Eq will loop forever

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)

Minimax

- 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