Announcements

- Midterm #2 Monday after midterm break — see the course web site for more details (same format as Midterm 1, including you can write up to 24 hours early)
- Watch Piazza for booking project demos
- *One must learn by doing the thing; for though you think you know it, you have no certainty until you try.*

  Sophocles ($\approx$ 497- 406 BCE)

  The doer alone learneth.

  Friedrich Nietzsche (1844 – 1900)

Review: Haskell since midterm

- `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
- newtype is like data but with more restrictions (and no runtime overhead)
- Type constructors and type variables

Last week

- Abstraction for games, so we can write interfaces and solvers for any games that fit the abstraction
- Representation of magic-sum game and count game
- A simple human interface for the abstraction
- `mm_player`: a player that searches through all possible games and returns a best move. (Using minimax).
- Make minimax more efficient (Caching / Memoization)
- Abstract data types
- Threading state and memoization
- Trees and functions as implementations for dictionaries

Today:

- Defining classes and higher level abstractions

Building a game abstraction

What do we need to represent:

- Magic sum game and other "fully observable" games
- Blackjack (or other card game)
- Adventure game where agent can move around, collect rewards, get penalties (without necessarily turn-taking with an opponent)
- Agents that can have state (e.g., agents that learn)
- Multiple games at the same time (e.g, simultaneously play magic sum and count games)

Questions

- What did we need to put the game abstraction at the top of the Magic sum game?
- What is wrong with having

  ```
  type Player = State -> Action
  ```

See: `Games2.hs`

An interface for games (Games2.hs)

```
-- gs=game_state  act=action
data State gs act = State gs [act]
         deriving (Ord, Eq, Show)

data Result gs act =
              EndOfGame String Double (State gs act)
            | ContinueGame Double (State gs act)
         deriving (Eq, Show)

type Game gs act = act -> State gs act -> Result gs act

-- gs=game_state  act=action ps=player_state
type Player gs act ps = Game gs  act -> Result gs act
                              -> ps -> (act, ps)
```

Clicker Question

```
data State gs act = State gs [act]
data Result gs act =
              EndOfGame String Double (State gs act)
            | ContinueGame Double (State gs act)
```

What is not true?

- A State is both a type constructor and a function
- B The State function takes a gs and a list of act
- C `EndOfGame` is a function that takes 3 arguments
- D `Result` is a function that takes 2 arguments

Clicker Question

```
data State gs act = State gs [act]
data Result gs act =
              EndOfGame String Double (State gs act)
            | ContinueGame Double (State gs act)
----------------------------------*************
```

The `(State gs act)` above the stars

- A Gives an error because act should be in square brackets
- B Refers to the type constructor for State
- C Refers to the function State
- D Doesn't need the `gs act` arguments

Clicker Question

```
data Result gs act =
               EndOfGame String Double (State gs act)
             | ContinueGame Double (State gs act)
         deriving (Eq, Show)
```

What is not true:

- A gs is a type variable
- B `EndOfGame "Fun" 7 bla`
  is of type `Result Int Int` as long as bla is of type
  `State Int Int`.
- C At compile time gs needs to be resolved into an actual type
- D `ContinueGame` is a function that takes 2 arguments
- E A function to return the reward associated with a result can have the type
  `reward :: Result -> Double`

Clicker Question

If we were to have:

```
type Game mt st init
    = Action mt st init -> Result mt st init
```

What is true:

- A Everything of type `Game` is a function that takes one argument
- C Everything of type `Game` is a function that takes three arguments
- D We cannot tell what something of type `Game` is from this declaration

Redefining show for card games (Games2.hs)

- What if we also want to include blackjack?
  Actions: Flip, Hold
  State: current count and the deck
- The state for blackjack includes the deck, but the player shouldn't see or have access to the deck!!
  It shouldn't be able to simulate next card.
- Don't export the constructor:

```
data Rands = RandsC [Double]     -- random numbers
instance Show Rands where
   show d = "_"
instance Eq Rands where
   x == y = True
instance Ord Rands where
   x <= y = True
```

- Deck is an infinite list of numbers.
- Generating random numbers with a seed from system can only be done in IO.

Dictionaries without data structures

- A dictionary is a function from keys into values.
  Functions can be used to implement dictionaries

  ```
  type Dict k v = (k -> Maybe v)
  ```

- How can we implement

  ```
  emptyDict:: Dict k v   -- the empty dictionary
  getval :: (Eq k) => k -> Dict k v -> Maybe v
  insertval :: (Eq k) => k -> v -> Dict k v -> Dict k v
  show :: (Dict k v) -> String
  ```

- To enable show:

  ```
  newtype Dict k v = FunDict (k -> Maybe v)
  ```

  except then we can't implement show for dictionaries

  ```
  instance Show (Dict k t) where
     show d = "Function_dictionary"
  ```

  (see FunDict2.hs, FunDict.hs)

Functional programming in other languages (pythonDemo.py)

- Other languages have adopted features from functional programming languages.
- E.g. Python has functions as first-class objects, lambda, list comprehensions (as well as set comprehensions, dictionary comprehensions), and even some lazy computation.

```
a1 = lambda x:x+1
a17 = a1(7)
odd = lambda n: n % 2 ==1
def even(n): return n % 2 == 0
l1 = [x*x for x in range(10) if odd(x)] # list
s1 = {x*x for x in range(10) if odd(x)} # set
d1 = {x*x:x for x in range(10) if odd(x)}  # dictionary
g1 = (x*x for x in range(10) if odd(x)) # generator
```

- Sometimes they do weird things (because of side effects).