Announcements

- Midterm #2 Monday after break — same format as midterm 1

- *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)

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

Overview

Last classes:

- 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

Today:

- More on games and abstract data types

Making Caching Useful

- Caching doesn't prune any nodes in magic-sum game! Why?
- Represent each state in canonical form:
  unique representation for each state. (sorted lists)
- with import MagicSum and TreeDict (top of Minimax_mem):

```
*Minimax_mem> minimax magicsum magicsum_start emptyDict
((9,0.0),dict)
*Minimax_mem> mema = (snd it)
*Minimax_mem> stats  mema
"Number of elements=294778, Depth=103"
```

- with import MagicSum_ord and TreeDict (at the top of Minimax_mem):

```
*Minimax_mem> minimax magicsum magicsum_start emptyDict
((9,0.0),dict)
*Minimax_mem> mema = (snd it)
*Minimax_mem> stats  mema
"Number of elements=4520, Depth=52"
```

Balancing Trees

- `"Number of elements=294778, Depth=103"`
  `"Number of elements=4520, Depth=52"`
- What is suspicious about this?
  The trees are are very unbalanced. The first dictionary should be able to be represented with a tree of depth 19, and the second one with a tree of depth 13.
- Is there a simple way to keep the tree approximately balanced?
- use $(hash\ k, k)$ as the key in the tree, as long as $hask\ k$ randomizes the ordering.

Clicker Question

Using $(hash\ k, k)$ as the key in the tree

A has to be done as a special case each time because *hash* needs to be defined for each type, and Haskell needs a type for the *hash* function

B could be done in DictTree just by calling *hash*

C could be done if we define a class for types that include a *hash* function, and only use *hash* for types in the class

D requires support in a low level language like C++, because hash functions could only improve performance if defined efficiently in C++.

Building a hashing dictionary

- Define a class for types that implement *hash*
- Make the type *State* be in that class
- Define a hashing tree dictionary that uses hash, but does not change the definition of *TreeDict*

Defining classes (Hash.hs)

- Define a class for types that implement *hash*

```
class Hash t where
    hash :: t -> Int
```

  A type in the *Hash* class implements *hash*.

- Define hash functions for Ints e.g.:

```
instance Hash Int where
    hash n = floor(numHashVals *
                   fractionalPart(arbMun *fromIntegral n))
```

- Define a hash function for lists (as long as the base type is hashable):

```
instance Hash t => Hash [t] where
    hash [] = 1741
    hash (h:t) = hash ( hash h + hash t)
```

Clicker Questions

For the two definitions of Hash for lists:

i)
```
instance Hash t => Hash [t] where
    hash [] = 1741
    hash (h:t) = hash ( hash h + hash t)
```

ii)
```
instance Hash t => Hash [t] where
    hash lst = hash (sum [hash e | e <- lst])
```

Which one always maps permutations to the same value?

A Both (i) and (ii)

B (i) but not (ii)

C (ii) but not (i)

D neither

Defining a tree dictionary with hashing

- How can we build a hashing tree dictionary, without changing TreeDict?
- See HashTreeDict.hs
- Note that Haskell has a standard class Hashable that act like our Hash.

Incorporate Hashing into game playing

- Import HashTreeDict into Minimax
- See `Minimax_mem_hash.hs`
- What else do we need to do?
- See `MagicSum_ord_hash.hs`

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`