Announcements

I have made this longer than usual because I have not had time to make it shorter.

Blaise Pascal, 1657

I have already made this paper too long, for which I must crave pardon, not having now time to make it shorter.

Benjamin Franklin, 1750

From `https://quoteinvestigator.com/2012/04/28/shorter-letter/`

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)

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).

Today:

- Make minimax more efficient
- Threading state
- Memoization
- Different dictionary implementations

Games

- Players observe state and make actions
- Games take actions and update state of game, perhaps finishing.

```
type Game = Action -> State -> Result

type Player = State -> Action

data State = State InternalState [Action]
         deriving (Ord, Eq, Show)

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

See MagicSum.hs Play.hs CountGame.hs

Minimax

- `type Game = Action -> State -> Result`
  `type Player = State -> Action`
  `mm_player:: Game -> Player`
  The game can be asked hypothetical questions about the result of a move. (Because it is functional.)
- In any state (if there is a move available), the agent chooses the action with the highest value after playing the action.
- The value is either:
  - the value for the end of the game, or
  - the negation of the value for the opponent (who now plays)
- `minimax:: Game -> State -> (Action, Double)`
- Minimax takes a game and a state and returns (action,value) for the best move (assuming there are moves available)
- `value:: Game -> Result -> Double`
- `mm_player game state = fst ( minimax game state)`
- See `Minimax.hs` (run the test cases)

Improving Minimax by caching results

- Minimax could cache the values of states it has evaluated
- A dictionary can be used to remember values
- A dictionary maps a key to a value

  `Dict k v`

  is a dictionary with key type `k` and value type `v`
- Dict Interface:

```
emptyDict :: Dict k v
getval :: (Ord k) => k -> Dict k v -> Maybe v
insertval :: (Ord k) => k -> v -> Dict k v
                             -> Dict k v
stats :: Dict t1 t2 -> [Char]
```

  "abstract data type"
- Minimax can use

  `Dict state (action,value)`

Minimax with memory (Minimax_mem.hs)

- Minimax without memory:

```
minimax:: Game -> State -> (Action, Double)
valueact :: Game -> State -> Action -> Double
value:: Game -> Result -> Double
```

- What type should memory be? Either:

```
type Mem = Dict State (Action, Double)
type Mem = Dict (State, Action) Double
```

- Memory can be threaded through the program:

```
minimax:: Game -> State ->
                    Mem -> ((Action, Double), Mem)
valueact :: Game -> State -> Action ->
                    Mem -> (Double,Mem)
value:: Game -> Result ->
                    Mem -> (Double,Mem)
```

  The can all use, pass the memory to functions they call, and update memory as appropriate.

Threading state through value function

value function without memory:

```
value:: Game -> Result -> Double
value _  (EndOfGame val _) = val
value game (ContinueGame st) =
        let (_,val) = minimax game st
           in -val
```

value function with memory (does not update dictionary)

```
type Mem = Dict State (Action, Double)
value:: Game -> Result -> Mem -> (Double, Mem)

value _  (EndOfGame val _) mem = (val, mem)
value game (ContinueGame st) mem =
      let ((_,val), mem1) = minimax game st mem
         in  (-val, mem1)
```

Threading state through minimax function

```
minimax:: Game -> State -> (Action, Double)
minimax game st  =
      argmax (valueact game st) avail
      where State _ avail = st
```

With memory:

```
type Mem = Dict State (Action, Double)
minimax:: Game -> State -> Mem -> ((Action, Double), Mem)
minimax game st mem =
   case getval st mem of
      Just act_val  -> (act_val,mem)
      Nothing ->
        let (act_val,mem1) =
              argmax_mem (valueact game st) avail mem
        in (act_val, (insertval st act_val mem1))
    where State _ avail = st
```

Argmax with memory

```
argmax :: Ord v => (e -> v) -> [e] -> (e,v)
argmax f [e] = (e, f e)
argmax f (h:t)
    | fh > ft = (h,fh)
    | otherwise = (bt, ft)
   where  (bt,ft) = argmax f t
          fh = f h
```

argmax with memory

```
argmax_mem::Ord v=>(e-> m->(v,m)) -> [e] -> m -> ((e,v),m)
argmax_mem f [e] mem = ((e, v), mem1)
      where (v, mem1) = f e mem
argmax_mem f (h:t) mem
    | fh > ft = ((h,fh),mem2)
    | otherwise = ((bt, ft),mem2)
   where ((bt,ft),mem1) = argmax_mem f t mem
         (fh,mem2) = f h mem1
```