Advanced Functional Programming

Lecture 1 Sudoku

Basic type declarations

```
type Grid     = Matrix Value
type Matrix a = [Row a]
type Row a    = [a]
type Value    = Char

-- Alternatively Grid could be defined as(Not optimal verbose is life)
type Grid     = [[Char]]
```

On the basic definitions we want to define all our types as we will be working with functions that will be taken those types as parameters.

Basic definitions

This can be seen as the init functions.

```
boxsize :: Int
boxsize = 3

values :: [Value]
values = [1..9]

empty :: Value -> Bool
empty = (== '.')

single :: [a] -> Bool
single [_] = True
single  _  = Flase
```

Making a empty grid with sexy functions

```
blank :: Grid
blank = replicate n(replicate n '.')
       where n = boxsize ^ 2
-- This makes an array of arrays based on box size square
```

Validator function

```
valid :: Grid -> Bool
valid g = all nodeups (rows g) &&
```

```
         all nodeups (cols g) &&
         all nodeups (boxs g)
-- rows, cols and boxs check if one square is valid in a row a column and a box respectivel
rows, cols, boxs :: Matrix a -> [Row a]
rows = id
cols = tranpose
boxs = unpack . map cols . pack
       where
         pack   = split . map split
         split  = chop boxsize
         unpack = map concat . concat
chop :: Int -> [a] -> [[a]]
chop n [] = []
chop n xs = take n xs : chop n (drop n xs)

nodeups :: Eq a => [a] -> Bool
nodeups []     = True
nodeups (x:xs) = not (elem x xs) && nodeups xs
```

Lecture 2 More Sudoku

Function composition not that hard so learn it and use it

The dot(.) notation can be use to easily pass a function as a parameter of another function and create chained functions

```
solve :: Grid -> [Grid]
solve  = filter valid . collapse . choices
-- Both do the same but the first version is sexier arguments can be hidden because curry s
solve g = filter valid(collapse(choices g))
```

Choices function

```
-- This uses the parametric type of Matrix to allow it to take not only a single value but
type Choices = [Value]
choices :: Grid -> Matrix Choices
choices g = map(map choice) g
  where
    choice v =
      if v == '.' then
        [1..9]
      else
          [v]
```

The solver definitions

These are the functions that actually solve the puzzle

```
-- Cartesian product
cp :: [[a]] -> [[a]]
cp []       = [[]]
cp (xs:xss) = [y:ys | y <- xs, ys <- cp xss]
-- Optimization(lazy brute forcing)
collapse :: Matrix[a] -> [Matrix a]
collapse  = cp . map cp
-- This is the same as
-- collapse g = cp(map cp g)
-- Apply cp to all inputs and then apply cp to the final result
```

Optimizations

These optimizations will drastically improve the brute force algorithm by adding lazy evaluation(based on pre existing numbers on the grid)

```
lazySolver :: Grid -> [Grid]
lazySolver  = filter valid . collapse . prune . choice
--Still slow af fam
prune :: Matrix Choices -> Matrix Choices
prune  = pryneBy boxs . prubeBy cols . pruneBy roows
        where pruneBy f = f . map reduce . f
        --niiiiice code right there

--removes the obivious cases(in case brute force worked)
recude :: Row Choices -> Row Choices
recude xss = [xs 'minux' singles | xs <- xss]
            where singles = concat (filter single xss)
minus :: Choices -> Choices -> Choices
xs 'minus' ys  = if single xs then xs else xs \\ ys
-- look at that next level pattern matching

-- Super lazy solver
llSolver :: Grid -> [Grid]
llSolver  = filter valid . collapse . fix . prune . choice
```

Cw1 - Game A.I.

Game trees:

To use a game tree you will need to define a maximum depth, and expand every single possible outcome.

1. Produce the game trees
2. Label each leaf with the winner or with B if the game is a draw
3. Work up the grid(By copying the child's classification to their parents)
4. Decide which is the best possible next step

Checking

While checking for winners make one check row function and reapply that to matrix operations of the same board.

A simple evaluator

This is a simple evaluator which evaluates expressions with a division operator. This is done recursively and uses the Maybe monad to achieve error checking(for division on 0)

```
--Simple div evaluator
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Maybe Int
eval(Val n) = Just n
eval(Div x y) =
  case eval n of
    Nothing -> Nothing
    Just n -> case eval y of
      Nothing -> Nothing
      Just m -> safediv n m

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just(x 'Div' y)

--More error checking with fancy stuff
--I will call this fancy error propagating monad(aka bind), this evaluates Maybe types and
--(>>=)
eval(Val n) = Just n
eval(Div x y) = eval x >>= (\n)
```

```
                eval y >>= (\m)
-- Now the op thing is that is applied automatically to do notation. Now watch this
eval :: Expr -> Maybe Int
eval(Val n) = Just n
eval(Div x y) = do n <- eval x
                   m <- eval y
                   safediv n m
```