Code (Tuesday Week 7)

Labelling Trees

Haskell

```haskell
import Control.Monad.State

data Tree a = Branch a (Tree a) (Tree a) | Leaf
     deriving (Show, Eq)

-- label in infix order, starting at 1.
label :: Tree () -> Tree Int
label t = snd (go t 1)
 where
   go :: Tree () -> Int -> (Int, Tree Int)
   go Leaf c = (c, Leaf)
   go (Branch x l r) c1 = let
         (c2, l') = go l c1
         x' = c2
         (c3, r') = go r (c2 + 1)
    in (c3, Branch x' l' r')


type Counter a = State Int a

count :: Counter Int
count = do
    x <- get
    put (x + 1)
    pure x

runCounter :: Counter a -> a
runCounter c = evalState c 1
-- evalState :: (State s a) -> s -> a

label' :: Tree () -> Tree Int
label' t = runCounter (go t)
  where
    go :: Tree () -> Counter (Tree Int)
    go Leaf = pure Leaf
    go (Branch x l r) = do
        l' <- go l
        x' <- count
        r' <- go r
        pure (Branch x' l' r')
```

```
    -- pure :: a -> Counter a
```

IO Monad

Drawing Triangles

```
printTriangle :: Int -> IO ()
printTriangle 0 = pure ()
printTriangle n = do
    putStrLn (replicate n '*')
    printTriangle (n - 1)

main = printTriangle 9
```

Maze Game

```
import Data.List
import System.IO

mazeSize :: Int
mazeSize = 10

data Tile = Wall | Floor deriving (Show, Eq)


type Point = (Int, Int)


lookupMap :: [Tile] -> Point -> Tile
lookupMap ts (x,y) = ts !! (y * mazeSize + x)

addX :: Int -> Point -> Point
addX dx (x,y) = (x + dx, y)

addY :: Int -> Point -> Point
addY dy (x,y) = (x, y + dy)

data Game = G { player :: Point
              , map    :: [Tile]
              }

invariant :: Game -> Bool
invariant (G (x,y) ts) = x >= 0 && x < mazeSize
```

```
                    && y >= 0 && y < mazeSize
                    && lookupMap ts (x,y) /= Wall

moveLeft :: Game -> Game
moveLeft (G p m)
  = let g' = G (addX (-1) p) m
     in if invariant g' then g' else G p m

moveRight :: Game -> Game
moveRight (G p m)
  = let g' = G (addX 1 p) m
     in if invariant g' then g' else G p m

moveUp :: Game -> Game
moveUp (G p m)
  = let g' = G (addY (-1) p) m
     in if invariant g' then g' else G p m

moveDown :: Game -> Game
moveDown (G p m)
  = let g' = G (addY 1 p) m
     in if invariant g' then g' else G p m

won :: Game -> Bool
won (G p m) = p == (mazeSize-1,mazeSize-1)

main :: IO ()
main = do
    str <- readFile "input.txt"
    let initial = G (0,0) (stringToMap str)
    gameLoop initial
  where
    gameLoop :: Game -> IO ()
    gameLoop state
        | won state = putStrLn "You win!"
        | otherwise = do
            display state
            c <- getChar'
            case c of
                'w' -> gameLoop (moveUp state)
                'a' -> gameLoop (moveLeft state)
                's' -> gameLoop (moveDown state)
                'd' -> gameLoop (moveRight state)
                'q' -> pure ()
                _   -> gameLoop state

stringToMap :: String -> [Tile]
stringToMap [] = []
stringToMap ('#':xs) = Wall : stringToMap xs
```

```
stringToMap (' ':xs) = Floor : stringToMap xs
stringToMap (c:xs)   = stringToMap xs

display :: Game -> IO ()
display (G (px,py) m) = printer (0,0) m
  where
    printer (x,y) (t:ts) = do
        if (x,y) == (px,py) then putChar '@'
        else if t == Wall then putChar '#'
        else putChar ' '

        if (x == mazeSize - 1) then do
            putChar '\n'
            printer (0,y+1) ts
        else printer (x+1,y) ts
    printer (x,y) [] = putChar '\n'

getChar' :: IO Char
getChar' = do
    b <- hGetBuffering stdin
    e <- hGetEcho stdin
    hSetBuffering stdin NoBuffering
    hSetEcho stdin False
    x <- getChar
    hSetBuffering stdin b
    hSetEcho stdin e
    pure x
```

The level used as input was as follows. The parser will ignore the pipe characters, which are placed merely for visual clarity:

```
  #       |
# # ##### |
#   #     |
##### ####|
      #   |
 ###### # |
        ##|
### ##### |
 #        |
## #####  |
```

QuickChecking IO

IORefs Average

```
import Data.IORef
import Test.QuickCheck.Monadic
import Test.QuickCheck

averageListIO :: [Int] -> IO Int
averageListIO ls = do
    sum <- newIORef 0
    count <- newIORef 0
    let loop :: [Int] -> IO ()
        loop [] = pure ()
        loop (x:xs) = do
            s <- readIORef sum
            writeIORef sum (s + x)
            c <- readIORef count
            writeIORef count (c + 1)
            loop xs
    loop ls
    s <- readIORef sum
    c <- readIORef count
    pure (s `div` c)

prop_average :: [Int] -> Property
prop_average ls = monadicIO $ do
    pre (length ls > 0)
    avg <- run (averageListIO ls)
    assert (avg == (sum ls `div` length ls))
```

GNU Factor

```
import Test.QuickCheck
import Test.QuickCheck.Modifiers
import Test.QuickCheck.Monadic
import System.Process

-- readProcess :: FilePath -> [String] -> String -> IO String

test_gnuFactor :: Positive Integer -> Property
test_gnuFactor (Positive n) = monadicIO $ do
    str <- run (readProcess "gfactor" [show n] "")
    let factors = map read (tail (words str))
    assert (product factors == n)
```