Code and Proofs (Week 2 Wed)

Fractal Trees

```haskell
fracTree :: Picture
fracTree = tree 10 (Point 400 800) (Vector 0 (-100)) red

tree :: Int -> Point -> Vector -> Colour -> Picture
tree depth base direction colour
  | depth == 0 = drawLine line
  | otherwise
    =  drawLine line
    ++ tree (depth - 1) nextBase left nextColour -- left tree
    ++ tree (depth - 1) nextBase right nextColour -- left tree
  where
    drawLine :: Line -> Picture
    drawLine (Line start end) =
      [ Path [start, end] colour Solid ]

    line = Line base nextBase
    nextBase = offset direction base

    left = rotate (-pi /12) $ scale 0.8 $ direction
    right = rotate (pi /12) $ scale 0.8 $ direction

    nextColour =
      colour { redC = (redC colour) - 24, blueC = (blueC colour) + 24 }

-- Offset a point by a vector
offset :: Vector -> Point -> Point
offset (Vector vx vy) (Point px py)
  = Point (px + vx) (py + vy)

-- Scale a vector
scale :: Float -> Vector -> Vector
scale factor (Vector x y) = Vector (factor * x) (factor * y)

-- Rotate a vector (in radians)
rotate :: Float -> Vector -> Vector
rotate radians (Vector x y) = Vector xRotated yRotated
  where
    -- As polar
```

```
    radius = sqrt $ (x * x) + (y * y)
    theta =
      if radius == 0
        then 0
        else if y >= 0
          then acos $ x / radius
          else - (acos $ x / radius)
    -- Rotate theta
    rotated = theta + radians
    -- Back to cartesian
    xRotated = radius * (cos rotated)
    yRotated = radius * (sin rotated)
```

Proofs

Natural Numbers

Definitions

```
data Nat = Zero
         | Succ Nat

add :: Nat -> Nat -> Nat
add Zero     b = b               -- 1
add (Succ a) b = add a (Succ b) -- 2

one = Succ Zero                  -- 3
two = Succ (Succ Zero)           -- 4
```

Proof of 1 + 1 = 2

```
two
  = add one one
  = add (Succ Zero) (Succ Zero)   -- 3
  = add Zero (Succ (Succ Zero))   -- 2
  = Succ (Succ Zero)              -- 1
  = two                           -- 4<
```

Lists

Append

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys              -- 1
(x:xs) ++ ys = x : xs ++ ys    -- 2
```

Proof of associativity of append (from Semigroup)

Our proof goal is:

```
((xs ++ ys) ++ zs) = (xs ++ (ys ++ zs))
```

Base case:

```
[] ++ (ys ++ zs)
  = ([] ++ ys) ++ zs
  = ys ++ zs          -- 1
  = [] ++ (ys ++ zs)  -- 1<
```

Recursive case

```
(x:xs) ++ (ys ++ zs)
  = ((x:xs) ++ ys) ++ zs
  = (x : (xs ++ ys)) ++ zs   -- 2
  = x : ((xs ++ ys) ++ zs)   -- 2
  = x : (xs ++ (ys ++ zs))   -- I.H.
  = (x:xs) ++ (ys ++ zs)     -- 2<
```

Proof of identity for append (from Monoid)

Left identity

```
xs
  = [] ++ xs
  = xs        -- 1
```

Right identity: We want to show:

```
xs ++ [] == xs
```

Base case:

```
[]
  = [] ++ []
  = []          -- 1
```

Inductive case:

```
(x:xs)
  = (x:xs) ++ []
  = x : (xs ++ [])  -- 2
  = x : xs          -- I.H.
```

Reverse

```
reverse :: [a] -> [a]
reverse []     = []                       -- A
reverse (x:xs) = (reverse xs) ++ [x]     -- B
```

Lemma Proof

We want to prove:

```
reverse (ys ++ [x]) = x:(reverse ys) -- C
```

Base case:

```
(x:(reverse []))
  = reverse ([] ++ [x])
  = reverse ([x])        -- 1
  = reverse (x:[])       -- Defn. of []
  = reverse [] ++ [x]    -- B
  = [] ++ [x]            -- A
  = [x]                  -- 1
  = (x:[])               -- Defn. of []
  = (x:reverse [])       -- A<
```

Recursive case:

```
(x:reverse (y:ys))
  = reverse ((y:ys) ++ [x])
  = reverse (y : (ys ++ [x])) -- 2
  = reverse (ys ++ [x]) ++ [y] -- B
  = (x:(reverse ys)) ++ [y] -- C (I.H.)
  = x : (reverse ys ++ [y]) -- 2
  = x : reverse (y:ys) -- B<
```

Reverse involution proof

We want to show:

```
xs = reverse (reverse xs)
```

Base case:

```
[]
  = reverse (reverse []) -- A
  = reverse [] -- A
  = []
```

Recursive case:

```
(x:xs)
  = reverse (reverse (x:xs))
  = reverse ((reverse xs) ++ [x]) -- B
  = x:(reverse (reverse xs)) -- C
  = x:xs -- I.H.<
```