Assignment 4

Question 1

1.1)

```
pAverage :: (Fractional a) => [a] -> a
pAverage ls = sum ls / fromRational (toRational (length ls))
```

1.2)

```
hAverage :: (Fractional a) => a -> a -> [a] -> a
hAverage _   avg []     = avg
hAverage len avg (x:xs) =
  hAverage (len + 1) (((avg * len) + x) / (len + 1)) xs
```

1.3)

```
average :: (Fractional a) => [a] -> a
average = hAverage 0 0
```

1.4)

assume head', addLen and avg given below,

```
head' [] = 0

head' (x:xs)
= x

addLen len []
= len

addLen len (x:xs)
= len + 1

avg
= (sum / len)
```

iteration invariant,

```
h_average len avg xs
= ((avg * len) + head' xs) / addLen len xs
```

1.5)

Proof h_average is satisfied by Iteration Invariant,

```
LHS(11) : h_average len avg []
          = ((avg * len) + head' []) / (addLen len [])
                              [ apply assump. avg    ]
          = (((sum / len) * len) + head' []) / (addLen len [])
                              [ apply assump. addLen ]
          = (((sum / len) * len) + head' []) / len
                              [ apply assump. head'  ]
          = (((sum / len) * len) + 0) / len
                              [ simplify             ]
          = (sum + 0) / len
                              [ simplify             ]
          = sum / len
                              [ unapply assump. avg  ]
          = avg : RHS(11)

LHS(12) : h_average len avg (x:xs)
          = ((avg * len) + head' (x:xs)) / (addLen len (x:xs))
                              [ apply assump. avg    ]
          = (((sum / len) * len) + head' (x:xs)) / (addLen len (x:xs))
                              [ apply assump. addLen ]
          = (((sum / len) * len) + head' (x:xs)) / (len + 1)
                              [ apply assump. head'  ]
          = (((sum / len) * len) + x) / (len + 1)
                              [ unapply assump. avg  ]
          = ((avg * len) + x) / (len + 1)
                              [ as desired           ]
          = h_average (len + 1) (((avg * len) + x) / (len + 1)) xs : RHS(13)
```

1.6)

QuickCheck

1.6.a)

```haskell
prop1 :: [Float] -> Bool
prop1 xs =
  let l = length xs
  in  (average [left, right] == ((left + right) / 2))
      where
        left = average (take l xs)
        right = average (drop (l-1) xs)
        l = length xs
{-
*Q1 Test.QuickCheck> quickCheck $ prop1
+++ OK, passed 100 tests.
-}
```

1.6.b)

```haskell
prop2 :: [Float] -> Property
prop2 xs =
  l > 0 ==> avgXS == ((avgXS' * l') + last xs) / convert l
  where
    l    = length xs
    l'   =  convert (l - 1)
    xs' = take (l-1) xs
    avgXS   = average xs
    avgXS' = average xs'
    convert = fromRational . toRational
{-
*Q1 Test.QuickCheck> quickCheck $ prop2
+++ OK, passed 100 tests; 23 discarded.
*Q1 Test.QuickCheck>
-}
```

Question 2

```haskell
type State = Int
newtype ST a = S (State -> (a,State))

apply :: ST a -> State -> (a, State)
apply (S st) x = st x

instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  fmap g st = st >>= (pure . g)

instance Applicative ST where
  -- pure :: a -> ST a
  pure x = S (\s -> (x,s))

  --(<*>) :: ST (a -> b) -> ST a -> ST b
  stf <*> stx = stf >>= (\f -> fmap f stx)

instance Monad ST where
  --  (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f = S (\s -> let (x,s') = apply st s
                      in apply (f x) s')
```

Question 3

2.1)

Assuming `fmap = <$>`,

```
(<$>) :: (a -> b) -> [a] -> [b]
_ <$> []     = []
f <$> (x:xs) = (f $ x) : (f <$> xs)
```

2.2)

```
pure :: a -> [a]
pure a = [a]

(<*>) :: [a -> b] -> [a] -> [b]
[] <*> _ = []
_ <*> [] = []
(f:fs) <*> (x:xs) = (f <$> pure x) ++ (fs <*> xs)
```

2.3)

Assuming `x = [x']`,

Note : `(\g -> g y) = ($ y)`,

```
x <*> pure y                          [ by assumption    ]
= [x'] <*> pure y                     [ apply pure       ]
= [x'] <*> [y]                        [ apply <*>        ]
= (x' <$> pure y) ++ ([] <*> [])      [ apply pure       ]
= (x' <$> [y]) ++ ([] <*> [])         [ apply <*>        ]
= (x' <$> [y]) ++ []                  [ apply ++         ]
= (x' <$> [y])                        [ apply <$>        ]
= (x' $ y) : (x' <$> [])              [ apply <$>        ]
= (x' $ y) : []                       [ apply (:)        ]
= [(x' $ y)]                          [ unapply ($ y)    ]
= [(($ y) x')]                        [ unapply <$>      ]
= [($ y)] <$> [x']                    [ unapply <*>      ]
= pure ($ y) <*> [x']                 [ note             ]
= pure (\g -> g y) <*> [x']           [ by assumption    ]
= pure (\g -> g y) <*> x
```

Question 4

4.1)

Base Case Assume, `(g . h) x = (g (h x))`.

Show hold for `(Leaf x)`,

```
fmap (g . h) (Leaf x)
= Leaf $ (g . h) x              [ apply fmap def.  ]
= Leaf $ (g (h x))              [ apply (.) def.   ]
= fmap g (Leaf $ h x)           [ unapply fmap g   ]
= fmap h (fmap g (Leaf $ h x)) [ unapply fmap h   ]
= fmap g . fmap h (Leaf $ x)    [ unapply (.) def. ]
```

I.H Assume holds for `lt` and `rt` in `(Node lt x rt)`

For lt,

```
fmap (g . h) lt = fmap g . fmap h lt
```

and rt,

```
fmap (g . h) rt = fmap g . fmap h rt
```

I.S Prove for `fmap (g . h) (Node lt x rt) = fmap g . fmap h (Node lt x rt)`,

```
fmap (g . h) (Node lt x rt)
= Node (fmap (g . h) lt) ((g . h) x) (fmap (g . h) rt)        [ apply fmap def.  ]
= Node (fmap g . fmap h lt) ((g . h) x) (fmap g . fmap h rt)  [ by I.H           ]
= Node (fmap g (fmap h lt)) (g (h x)) (fmap g (fmap h rt))    [ apply (.) def.   ]
= fmap g (Node (fmap h lt) (h x) (fmap h rt))                 [ unapply fmap g   ]
= fmap g (fmap h (Node lt x rt))                              [ unapply fmap h   ]
= fmap g . fmap h (Node lt x rt)                              [ unapply (.) def. ]
```

Question 5

5.1)

```
map :: (a -> b) -> [a] -> [b]
map f = unfold null (f . head) tail
```

5.2)

```
iterate :: (a -> a) -> a -> [a]
iterate f b = unfold null head (pure . f . head) [b]
```