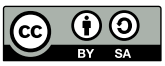
# Assignment 8




2019-12-11 15:44
08d925d

due *Tue 5 Nov*

```haskell
module A08 where

-- Here is a definition of a pseudo-random number generator that works
-- by passing around its state.

data Seed = Seed { unSeed :: Int }
  deriving (Eq, Show)

type Gen a = Seed -> (a, Seed)

rand :: Gen Int
rand (Seed s) = (s', Seed s')
  where
    s' = (s * 16807) `mod` 0x7FFFFFFF

-- Now let's make an enumerated type for representing coin flips. A
-- coin flip can be head or tails. This is 'isomorphic' to a Boolean
-- value -- we could just represent heads as True and tails as False (or
-- vice-versa), but for clarity it's sometimes helpful to declare such
-- things as types.

data Coin = Heads | Tails
  deriving (Eq, Show, Bounded, Enum)

-- Make a function to convert any integer to a coin flip. You want the
-- coin to be fair (or as close as possible), so use even or odd to
-- determine whether you produce Heads or Tails.

coinFromInt :: Int -> Coin
coinFromInt = error "TODO"

-- Now create a random generator for coin flips. You can call rand
-- and/or coinFromInt within this code.

flipCoin :: Gen Coin
flipCoin = error "TODO"
```

```haskell
-- This function creates a pair of random integers, by calling rand
-- twice. But note how it has to thread the PRNG state through both
-- calls. Given s0, we get s1 from the first rand, then give s1 to the
-- next rand and get s2. If we mess that up, and end up reusing a state
-- that has already been used, then duplicate results will appear.

randPair :: Gen (Int, Int)
randPair s0 = ((i1, i2), s2)
  where (i1,s1) = rand s0
        (i2,s2) = rand s1

-- Now, generalize randPair so that instead of calling rand twice, it
-- can call *any two generators* given as arguments. For example, you
-- should be able to do (genPair rand rand) to reproduce what randPair
-- can do. Or you can do (genPair flipCoin rand) or (genPair flipCoin
-- flipCoin).

genPair :: Gen a -> Gen b -> Gen (a,b)
genPair = error "TODO"

-- Generators are functors -- see if you can implement fmap!

fmapGen :: (a -> b) -> Gen a -> Gen b
fmapGen = error "TODO"

-- Finally, let's try to iterate a generator to produce a list of
-- results. The integer specifies the length of the list, so when
-- that's zero the result can be empty.

iterGen :: Gen a -> Int -> Gen [a]
iterGen = error "TODO"
```