Motivation

```haskell
tossDice :: Rand Int
tossDice = do
    d1 <- dice
    d2 <- dice
    return $ d1 + d2
```

```
[11,8,10,7,11,5,8,4,6,7]
```

```haskell
tossDicePrime :: Rand [Int]
tossDicePrime = weighted $ do
    d <- tossDice
    score $ if prime d then 1 else 0
    return $ d
```

```
[11,11,3,7,5,7,5,7,11,11]
```

```haskell
data Rand x where
    Ret :: x -> Rand x
    Sample01 :: (Float -> Rand x) -> Rand x
    Score :: Float -> Rand x -> Rand x
    Ap :: Rand (a -> x) -> Rand a -> Rand x
instance Functor Rand where
  fmap f (Ret x) = Ret (f x)
  fmap f (Sample01 r2mx) = Sample01 (\r -> fmap f (r2mx r))
  fmap f (Score s mx) = Score s (fmap f mx)
  fmap f (Ap m2x ma) = Ap ((f .) <$> m2x) ma
instance Applicative Rand where
  pure = Ret
  pa2b <*> pa = Ap pa2b pa
instance Monad Rand where
  return = Ret
  (Ret x) >>= x2my = x2my x
  (Sample01 r2mx) >>= x2my = Sample01 (\r -> r2mx r >>= x2my)
  (Score s mx) >>= x2my = Score s (mx >>= x2my)
  (Ap m2x ma) >>= x2my =
    m2x >>= \a2x -> ma >>= \a -> x2my (a2x a)
```

```haskell
-- | Run the computation _unweighted_.
-- | Ignores scores.
sample :: RandomGen g => g -> Rand a -> (a, g)
sample g (Ret a) = (a, g)
sample g (Sample01 f2my) =
  let (f, g') = random g in sample g' (f2my f)
sample g (Score f mx) = sample g mx -- Ignore score
sample g (Ap m2x ma) =
  let (a2x, g1) = sample g m2x
      (a, g2) = sample g1 ma
   in (a2x a, g2)
```

MCMC methods

```haskell
-- | Trace all random choices made when generating this value
data Trace a =
  Trace { tval :: a,
          tscore :: Float,
          trs :: [Float]
        }
-- | Lift a pure value into a Trace value
mkTrace :: a -> Trace a
mkTrace a = Trace a 1.0 []
-- | multiply a score to a trace
scoreTrace :: Float -> Trace a -> Trace a
scoreTrace f Trace{..} = Trace{tscore = tscore * f, ..}
-- | Prepend randomness
recordRandomness :: Float -> Trace a -> Trace a
recordRandomness r Trace{..} = Trace { trs = trs ++ [r], ..}
```

```haskell
-- | Trace a random computation.
-- We know what randomness is used
traceR :: Rand x -> Rand (Trace x)
traceR (Ret x) = Ret (mkTrace x)
traceR (Sample01 mx) = do
  r <- sample01
  trx <- traceR $ mx r
  return $ recordRandomness r $ trx
traceR (Score s mx) = do
  trx <- traceR $ mx
  return $ scoreTrace s $ trx
traceR (Ap rf rx) = do
  trf <- traceR rf
  trx <- traceR rx
  return $ Trace { tval = (tval trf) (tval trx),
          tscore = tscore trf * tscore trx,
          trs = if length (trs trf) > length (trs trx)
                  then trs trf
                  else trs trx
  }
```

– — Return a trace-adjusted MH computation

```haskell
mhStep :: Rand (Trace x) -- ^ proposal
       -> Trace x -- ^ current position
       -> Rand (Trace x)
mhStep r trace = do
  -- | Return the original randomness, perturbed
  rands' <- perturbRandomness (trs trace)
  -- | Run the original computation with the perturbation
  trace' <- feedRandomness rands' r
  let ratio = traceAcceptance trace' / traceAcceptance trace
  r <- sample01
  return $ if r < ratio then trace' else trace

traceAcceptance :: Trace x -> Float
traceAcceptance tx =
  tscore tx * fromIntegral (length (trs tx))

perturbRandomness :: [Float] -> Rand [Float]
perturbRandomness rands = do
  ix <- choose [0..(length rands-1)] -- ^ Random index
  r <- sample01 -- ^ random val
  -- | Replace random index w/ random val.
  return $ replaceListAt ix r rands
```

```haskell
-- | Find a starting position that does not have probability 0
findNonZeroTrace :: Rand (Trace x) -> Rand (Trace x)
findNonZeroTrace tracedR = do
  trace <- tracedR
  if tscore trace /= 0
  then return $ trace
  else findNonZeroTrace tracedR

-- | run the computatation after taking weights into account
weighted :: MCMC x => Rand x -> Rand [x]
weighted r =
  let tracedR = traceR r
      -- go :: Rand (Trace x) -> Rand (Trace [x])
      go tx = do
        tx' <- repeatM 10 (mhStep tracedR) $ tx
        liftA2 (:) (return tx) (go tx')
  in do
      seed <- findNonZeroTrace $ tracedR
      tracedRs <- go seed
      return $ map tval tracedRs
```

Payoff!

```haskell
predictCoinBias :: [Int] -> Rand [Float]
predictCoinBias flips = weighted $ do
  b <- sample01
  forM_ flips $ \f -> do
    -- | Maximum a posterior
    score $ if f == 1 then b else (1 - b)
  return $ b

predictCoinBiasNoData :: Rand [Float]
predictCoinBiasNoData = predictCoinBias []
```

```haskell
predictCoinBias0 :: Rand [Float]
predictCoinBias0 = predictCoinBias [0]
```

```haskell
predictCoinBias01 :: Rand [Float]
predictCoinBias01 = predictCoinBias [0, 1]
```

More fun stuff: sample from arbitrary distributions

```
sampleSinSq :: Rand [Float]
sampleSinSq = weighted $ do
  x <- (6 *) <$> sample01
  score $ (sin x) * (sin x)
  return $ x
```