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 va
data Trace a = Trace { tval :: a, tscore :: Float, trs ::
-- | 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], .
```