

Our primitives

```
-- | Convert a pure value into a Rand value
```

```
return :: a -> Rand a
```

```
-- | Get a random number
```

```
uniform01 :: Rand Float
```

```
-- | Take 'n' samples from a random variable
```

```
sample :: Int -> Rand a -> [a]
```

```
-- | take a Float, do *something*, and return nothing
```

```
score :: Float -> Rand ()
```

First example – The same as `System.Random`

```
-- | dice
dice :: Rand Int
dice = do
  u <- uniform01
  return $ floor (7*u)

-- | sum of dice
tossDice :: Rand Int
tossDice = do
  d1 <- dice
  d2 <- dice
  return $ d1 + d2

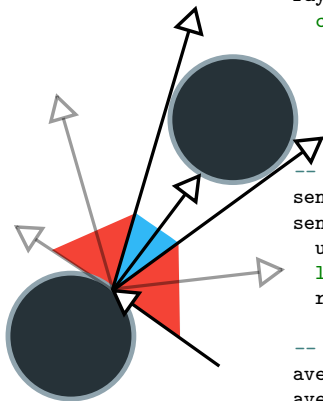
main :: IO ()
main = print $ sample 10 tossDice
```

Output:

[7,6,7,10,5,8,12,8,9,6]



Raytracing (Default)



```
-- / recursively raytrace
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
      color' <- averageRays loc
      return $ mixColor surface color'
    None -> return backgroundColor

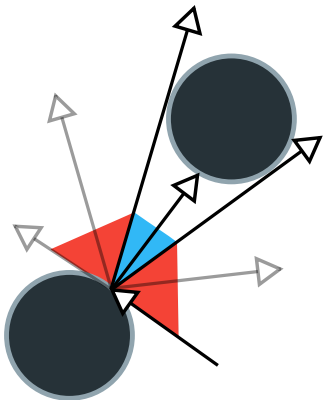
-- / Send a random ray
sendRandRay :: Position -> Rand Color
sendRandRay p =
  u <- uniform01
  let angle = 360 * u
  raytrace (makeRay p angle)

-- / Average rays sent from a location
averageRays :: Position -> Rand Color
averageRays p = do
  -- / computationally wasteful
  colors <- replicateM 100 (sendRandRay p)
  return $ averageColors colors

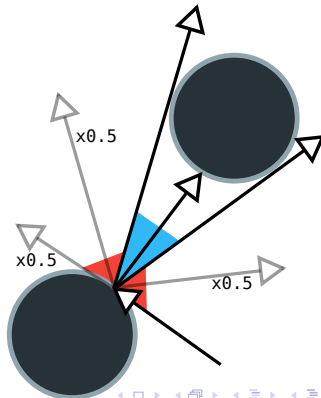
-- / Default background color.
backgroundColor = white
```

Raytracing (Scored)

```
raytrace :: Ray -> Rand Color
raytrace r = do
  case getCollision r of
    Some (surface, loc) ->
      color' <- averageRays loc
      return $ mixColor surface color'
    None -> return backgroundColor
```



```
raytrace' :: Ray -> Rand Color
raytrace' r = do
  case getCollision r of
    Some (surface, loc) ->
      color' <- averageRays loc
      return $ mixColor surface color'
    None -> do
      score 0.5 -- New!
      return backgroundColor
```



Program optimisation (Uniform)

```
-- | Randomly change programs and return their performance
equivRandomProgram :: Program -> Rand (Performance, Program)
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else return (0, p') -- A program that does not work has 0 perf.

-- | Take the random samples and pick the good performing ones
optimise :: Program -> Program
optimise p =
  let ps' = sample 100 (equivRandomProgram p)
  in snd $ maximumBy (compare . fst) ps'
```

Program optimisation (Scored)

```
equivRandomProgram' :: Program -> Rand (Performance, Program)
equivRandomProgram' p = do
  (perf, p) <- equivRandomProgram p
  let perf =
    if semanticsEqual p p'
    then performance p'
    else 0
  score perf -- ^ Correct programs are more likely
  return (perf, p')
```

```
equivRandomProgram :: Program -> Rand (Performance, Program)
equivRandomProgram p = do
  p' <- modifyProgram p
  if semanticsEqual p p'
  then return (performance p', p')
  else return (0, p') -- A program that does not work has 0 perf.
```

<http://stoke.stanford.edu/>

<https://github.com/bollu/blaze/blob/master/notebooks/tutorial.ipynb>

Learning from prior experience

```
-- / Naive understanding, in the beginning of the process
prior :: Rand a
prior = ...

-- / Learn as you go!
learn :: Rand a
learn = do
  value <- prior
  score (usefulness value)
  return value
```

Motivation

```
-- | fair dice
dice :: Rand Int
dice = choose [1, 2, 3, 4, 5, 6]
```

```
tossDice :: Rand Int
tossDice = do
  d1 <- dice
  d2 <- dice
  return $ d1 + d2
[7,6,7,10,5,8,12,8,9,6]
```

If the dice roll is prime,

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



```
data Rand x where
```

```
Ret :: x -> Rand x
```

```
SampleUniform01 :: (Double -> Rand x) -> Rand x
```

```
Score :: Double -> Rand x -> Rand x
```

```
-- | Run the computation _unweighted_.  
-- | Ignores scores.  
sample :: RandomGen g => g -> Rand a -> (a, g)  
sample g (Ret a) = (a, g)  
sample g (SampleUniform01 f2my) =  
    let (f, g') = random g in sample g' (f2my f)  
sample g (Score f mx) = sample g mx -- Ignore score
```

MCMC methods

```

-- | Trace all random choices made when generating this va
data Trace a =
  Trace { tval :: a, -- ^ The value itself
         tscore :: Double, -- ^ The total score
         trs :: [Double] -- ^ The ranom numbers used
       }

-- | 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 :: Double -> Trace a -> Trace a
scoreTrace f Trace{..} = Trace{tscore = tscore * f, ..}

-- | Prepend randomness
recordRandomness :: Double -> Trace a -> Trace a
recordRandomness r Trace{..} = Trace { trs = r:trs, ..}

-- | Trace a random computation.
-- We know what randomness is used
traceR :: Rand x -> Rand (Trace x)
traceR (Ret x) = Ret (mkTrace x)

```

- — Return a trace-adjusted MH computation

```

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 -> Double
traceAcceptance tx =
  tscore tx * fromIntegral (length (trs tx))

perturbRandomness :: [Double] -> Rand [Double]
perturbRandomness rands = do
  ix <- choose [0..(length rands-1)] -- ^ Random index
  r <- sample01 -- ^ random val

```

```

-- / 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 computation after taking weights into account
weighted :: MCMC x => Int -> Rand x -> Rand [x]
weighted 0 _ = return []
weighted n r =
  let tracedR = traceR r
      -- go :: Int -> Rand (Trace x) -> Rand (Trace [x])
      go 0 _ = return []
      go n tx = do
        tx' <- repeatM 10 (mhStep tracedR) $ tx
        txs <- go (n-1) tx'
        return (tx:txs)
  in do

```

Payoff!

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

predictCoinBiasNoData :: Rand Double
predictCoinBiasNoData = predictCoinBias []

████████████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████████████████████████████████████

predictCoinBias0 :: Rand Double
predictCoinBias0 = predictCoinBias [0]

████████████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████████████████████████████████████

predictCoinBias01 :: Rand Double
predictCoinBias01 = predictCoinBias [0, 1]

████████████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████████████████████████████████████
```


More fun stuff: sample from arbitrary distributions

```
sampleSinSq :: Rand Double
sampleSinSq = do
  x <- (6 *) <$> sample01
  score $ (sin x) * (sin x)
  return $ x
```



Transformations discovered by STOKE

```
// constant folding: 2 + 3 -> 5
*** original: (nparams: 0 | [IPush 2,IPush 3,IAdd])***
[IPush 5] | score: 2.5

// strength reduction: 2 * x -> x + x
*** original: (nparams: 1 | [IPush 2,IMul])***
[IDup,IAdd] | score: 2.25

// algebraic rewrite: x & x == x
*** original: (nparams: 1 | progInsts = [IDup,IAnd])***
[] | score: 3.0
```