Lecture 10 – Do Notation

- Do notation simplifies presentation of monadic computation.

```
do
  stmt1
  stmt2
  stmt3
```

- A statement can be a let declaration, a left-arrow bind, or an action (e.g. a return).

```
<action> ::= do <stmts>
          |  return <exp>
          |  guard <exp>   -- <exp> :: boolean
          |  ...

<stmts> ::= <stmt> | <stmt> <stmts>

<stmt> ::= let x = <exp>
        |  x <- <action>
        |  <action>
```

- guard returns an instance of Alternative such as MonadPlus.

Do Block

- A do-block is a (generic) Monad.

```
echo :: IO ()

echo = do
         x <- getLine
         print $ "You typed: " ++ x
```

- Left arrow translates to bind operator >>=.

```
echo' :: IO ()

echo' = getLine >>= \x ->
                       print $ "You typed: " ++ x


-- test
-- "You typed: test"
```

Do Block

- A sequence of actions translates to ‘then’ operator >>.

```haskell
prompt :: IO ()

prompt = do
            print "Type line1."
            print "Then type line2."

prompt
-- "Type line1."
-- "Then type line2."


-- equivalent definition
prompt' = print "Type line1." >>
          print "Then type line2."
```

Do Block

- Sequence discards previous results.

```
getLine2 :: IO String
getLine2 = do
             getLine  -- this line is discarded
             getLine

getLine2 >>= print -- means "getLine2 >>= \x -> print x"

test          -- input line 1
test again    -- input line 2
-- "test again"
```

Do Block

- The end of a do-block must be a monad such as a return.

```
get2Lines :: IO (String, String)

get2Lines = do
                 x <- getLine
                 y <- getLine
                 return (x, y)
```

Do Block

- Do-block is a monad.

```
echo2Lines :: IO ()

echo2Lines = do
               (x,y) <- get2Lines
               print $ "Line1 is: " ++ x
               print $ "Line2 is: " ++ y

echo2Lines

Learn do notation               -- line 1 input
Improve productivity            -- line 2 input
-- "Line1 is: Learn do notation"
-- "Line2 is: Improve productivity"
```

Do notation for List Monad

- Problem: given $n \geq 0$, find all pairs of positive integers i and j, where $1 = j < i < n$ such that $i + j$ is prime.

```
primePairs = \n ->
  do
    i <- [1..n-1]
    j <- [1..i-1]
    guard $ isPrime $ i+j
    return (j, i)

isPrime n = foldl f True [2..limit]
   where
     f c e = c && n `mod` e /= 0
     limit = floor $ sqrt $ fromIntegral n
```

List Monad using bind

- For good measure, below is the equivalent implementation using bind and translation of *guard*.

```
primePairs = \n ->
    [1..n-1] >>= \i ->
    [1..i-1] >>= \j ->
    (if isPrime $ i+j then pure () else mzero) >>
    return (j, i)

isPrime n = foldl f True [2..limit]
   where
     f c e = c && n `mod` e /= 0
     limit = floor $ sqrt $ fromIntegral n
```

List comprehension

- Problem: given $n \geq 0$, find all pairs of positive integers i and j, where $1 = j < i < n$ such that $i + j$ is prime.

```
primePairs = \n ->
  do
    i <- [1..n-1]
    j <- [1..i-1]
    guard $ isPrime $ i+j
    return (j, i)

primePairs' = \n ->
  [ (j, i) | i <- [1..n-1],
             j <- [1..i-1],
             isPrime $ i+j ]
```

List comprehension

- Problem: make an infinite list of prime numbers.

```
makePrime = filter isPrime [2..]

makePrime' = do x <- [2..]
                guard $ isPrime x
                return x

makePrime'' = [ x | x <- [2..],
                    isPrime x ]
```

FFT

- FFT shares a twiddle context

```
fft :: Signal -> Signal -> Signal
fft w x
  | length' x <= 1 = x
  | otherwise =
     let
         (even, odd) = split' x
         (w', _) = split' w    -- even half of twiddle
         e = fft w' even       -- fft with even signal
         o = fft w' odd        -- fft with odd signal
         p = w * o
     in
         (e + p) <> (e - p)

  where split' (Vec x) = let (e,o) = split x in (Vec e, Vec o)
        split [] = ([], [])
        split [a] = ([a], [])
        split (a:b:c) = let (x,y) = split c in (a:x, b:y)
```

Inverse FFT

- IFFT also needs a twiddle context

```
ifft :: Signal -> Signal -> Signal

ifft w x = let
             -- conjugate input signal
             v = fft w $ fmap conjugate x

             -- conjugate output signal
             v' = fmap conjugate v

           in
             -- scale output signal
             fmap (/n) v'

           where n = fromIntegral $ length' x
```

Low Pass Filter

- Low pass filter uses the same twiddle for fft and ifft.

```
low_pass :: Int -> Signal -> Signal

low_pass freq x =
    let y = \w ->
                fft w x          -- frequency domain signal
                f' = f * mask freq n
                ifft w f'        -- time domain signal
    in y w

    where
        n = length' x
        w = twiddle (fromIntegral n) 1
```

Reader for FFT

- use do block to create a Reader Monad

```
fft :: Signal -> Reader Signal Signal
fft x
  | length' x <= 1 = return x            -- return as a reader
  | otherwise = do
     let (even, odd) = split' x
     w <- ask                            -- get twiddle context
     let (w', _) = split' w
     e <- local (\_ -> w') (fft even)   -- fft with new context
     o <- local (\_ -> w') (fft odd)    -- fft with new context
     let p = w * o
     return $ (e + p) <> (e - p)        -- return as a reader

  where split' (Vec x) = let (e,o) = split x in (Vec e, Vec o)
        split [] = ([], [])
        split [a] = ([a], [])
        split (a:b:c) = let (x,y) = split c in (a:x, b:y)
```

Reader for Inverse FFT

- use do block to create a Reader Monad

```haskell
ifft :: Signal -> Reader Signal Signal

ifft x = do
          -- extract input signal from reader & conjugate
          v <- fft $ fmap conjugate x

          -- conjugate output signal
          let v' = fmap conjugate v

          -- scale output signal & return it as a Reader
          return $ fmap (/n) v'

      where n = fromIntegral $ length' x
```

Reader for Low Pass Filter

- use runReader to supply the initial context

```haskell
low_pass :: Int -> Signal -> Signal

low_pass freq x =
    let
        y = do f <- fft x  -- frequency domain signal
               let f' = f * mask freq n
               ifft f'     -- time domain signal
    in
        runReader y w

    where
        n = length' x
        w = twiddle (fromIntegral n) 1
```