# CIS 1904: Haskell

Monads

# What is a pure language?

- No side effects:
  - Printing
  - Reading from memory
  - Anything except evaluating a term down to a simpler term
    - e.g. $2 + 2 + 2 \rightarrow 4 + 2 \rightarrow 6$
- Easy to reason about
- Helps avoid classes of bugs (esp. concurrency bugs)
- We will see later how Haskell manages this
  - (It's monads)

# Monads

- A construct for modelling effectful code, with functions for *sequencing* actions
- A typeclass for type constructors of kind `* -> *`

# IO

```
lenInputLine :: IO Int
lenInputLine = do
    input <- getLine
    putStrLn (“Length: ” ++ show (length input))
    return (length input)
```

This sequences three actions:

1. Reading an input line from stdin
2. Writing the length of the input line to stdout
3. Creating the value given by `length input`

# IO

```
lenInputLine :: IO Int
lenInputLine =
    getLine >>=
    (\input ->
        putStrLn ("Length" ++ show (length input)) >>
        return (length input))
```

Note: the newlines are just for legibility in this case, they do not affect the code.

# Monads

```
class Applicative m => Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a

    m >> k = m >>= \_ -> k
```

`(>>=)` sequences effectful code

`return` gives us the representation of a pure expression within our effect model

Note: `return` is NOT the same concept as a return statement in other languages.

# Maybe

```
instance  Monad Maybe  where
    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    (Just x) >>= f  = f x
    Nothing  >>= _  = Nothing

    return :: a -> Maybe a
    return x = Just x
```

The effect we are modelling here is program failure, e.g., from a divide by 0.

# Maybe

```
safeDiv :: Double -> Double -> Maybe Double
saveDiv _ 0 = Nothing
safeDiv x y = x `div` y
```

# Maybe

```
safeDiv :: Double -> Double -> Maybe Double
saveDiv _ 0 = Nothing
safeDiv x y = x `div` y

-- x1/x2 + x3/x4
divAdd :: Double -> Double -> Double -> DoubleMaybe Double
divAdd x1 x2 x3 x4 = case safeDiv x1 x2 of
    Nothing -> Nothing
    Just r1 -> case safeDiv x3 x4 of
        Nothing -> Nothing
        Just r2 -> Just (r1 + r2)
```

# Maybe

```
divAdd :: Double -> Double -> Double -> DoubleMaybe Double
divAdd x1 x2 x3 x4 = do
    r1 <- safeDiv x1 x2
    r2 <- safeDiv x3 x4
    return r1 + r2
```

# Maybe

```
divAdd :: Double -> Double -> Double -> DoubleMaybe Double
divAdd x1 x2 x3 x4 = do
    r1 <- safeDiv x1 x2
    r2 <- safeDiv x3 x4
    return r1 + r2
```

# Maybe

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

addFirsts :: [Int] -> [Int] -> Maybe Int
addFirsts xs ys = case safeHead xs of
    Nothing -> Nothing
    Just x -> case safeHead ys of
        Nothing -> Nothing
        Just y -> Just (x + y)
```

# Maybe

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x : xs) = Just x

addFirsts :: [Int] -> [Int] -> Maybe Int
addFirsts xs ys = do
    x <- safeHead xs
    y <- safeHead ys
    return (x + y)
```

# Maybe

```
addFirsts :: [Int] -> [Int] -> Maybe Int
addFirsts xs ys = do
    x <- safeHead xs
    y <- safeHead ys
    return (x + y)


addFirsts [1,2] [] = ?
```

# Maybe

```
addFirsts :: [Int] -> [Int] -> Maybe Int
addFirsts xs ys = do
    x <- safeHead xs
    y <- safeHead ys
    return (x + y)
```

```
addFirsts [1,2] [3] = ?
```

# Maybe

```
addFirsts :: [Int] -> [Int] -> Maybe Int
addFirsts xs ys = do
    x <- safeHead xs
    y <- safeHead ys
    return (x + y)


addFirsts [1,2] [3,4] = ?
```

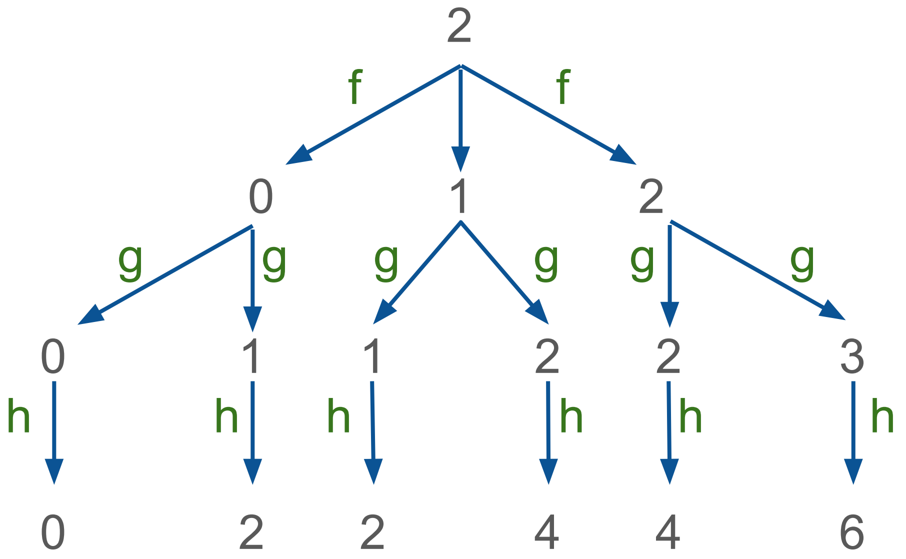
# List

Lists can be thought of as a model for *nondeterminism*.

f(x) = sample(0,x)

g(x) = if coinFlip()
then x
else x + 1

h(x) = 2 * x




[2]

[0,1,2]

[0,1,1,2,2,3]

[0,2,2,4,4,6]

# List

Lists can be thought of as a model for *nondeterminism*.

```
f x = [0..x]

g x = [x, x+1]

h x = [2 * x]
```

```
[2]

?? f [2] = [0,1,2]

?? g [0,1,2] = [0,1,1,2,2,3]

?? h [0,1,1,2,2,3] = [0,2,2,4,4,6]
```

# List

```
instance  Monad []  where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f     = concatMap f xs

    return :: a -> [a]
    return x = ??
```

# List

```
instance  Monad []  where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f    = concatMap f xs

    return :: a -> [a]
    return x = [x]
```

# List

```
f x = [0..x]
g x = [x, x+1]
h x = [2 * x]

foo x = do
    y <- f x
    z <- g y
    h z
```

# List

```
f x = [0..x]
g x = [x, x+1]
h x = [2 * x]

foo x = do                  2
    y <- f x            [0,1,2]
    z <- g y            [0,1,1,2,2,3]
    h z                     [0,2,2,4,4,6]
```

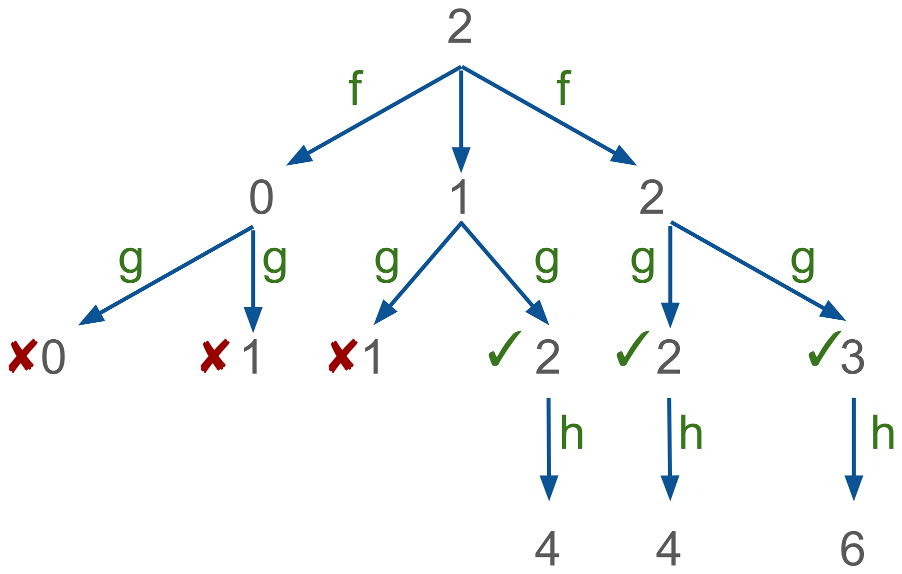
# Guard

```
f x = [0..x]
g x = [x, x+1]
h x = [2 * x]

foo x = do                   2
    y <- f x             [0,1,2]
    z <- g y             [0,1,1,2,2,3]
    guard (z > 1)            [2,2,3]
    h z                      [4,4,6]
```

Along each control-flow path, `guard` either ends computation or lets it continue.

# Guard

```
f x = [0..x]
g x = [x, x+1]
h x = [2 * x]

foo x = do                2
     y <- f x         [0,1,2]
     z <- g y         [0,1,1,2,2,3]
     guard (z > 1)        [2,2,3]
     h z                  [4,4,6]
```




Along each control-flow path, `guard` either ends computation or lets it continue.

# Guard

```
addFirstsEven xs ys = do
    x <- safeHead xs
    y <- safeHead ys
    guard (even x)
    return (x + y)
```

Along each control-flow path, `guard` either ends computation or lets it continue.

# List

```haskell
foo = [ x `div` 2 | x <- [0..10] , even x]

foo = do
    x <- [0..10]
    guard (even x)
    return (x `div` 2)

foo = [0..10] >>= \x -> guard (even x) >> return (x `div` 2)
```

What does `foo` return?

# List

```
foo = [ x `div` 2 | x <- [0..10] , even x]

foo = do
    x <- [0..10]
    guard (even x)
    return (x `div` 2)

foo = [0..10] >>= \x -> guard (even x) >> return (x `div` 2)
```

What does `foo` return? `[0,1,2,3,4,5]`

# List

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (mx : mxs) =
  mx >>= (\y -> (sequence mxs >>= (\ys -> return (y : ys))))
```

Note: `>>=` is right-associative, so these parentheses are redundant.

# List

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (mx : mxs) =
  mx >>= (\y -> (sequence mxs >>= (\ys -> return (y : ys))))
```

For `Maybe`, this means the whole list fails if any element failed.
For `IO`, this means we do the IO actions one after the other.
For `List`, this is all the ways to choose one value from each list:

```
sequence [“abc”, “de”] = ["ad","ae","bd","be","cd","ce"]
sequence [“abc”, “de”, “”] = []
```