函数式编程简介

PARKER LIU

The program is magic words

```
void qsort(int *A, int len) {
  if (len < 2) return;

  int pivot = A[len / 2];

  int i, j;
  for (i = 0, j = len - 1; ; i++, j--) {
    while (A[i] < pivot) i++;
    while (A[j] > pivot) j--;

    if (i >= j) break;

    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
  }

  qsort(A, i);
  qsort(A + i, len - i);
}
```

```c
void qsort(int *A, int len) {
  if (len < 2) return;

  int pivot = A[len / 2];

  int i, j;
  for (i = 0, j = len - 1; ; i++, j--) {
    while (A[i] < pivot) i++;
    while (A[j] > pivot) j--;

    if (i >= j) break;

    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
  }

  qsort(A, i);
  qsort(A + i, len - i);
}
```

```haskell
qsort [] = []
qsort (x:xs) = qsort left
            ++ (x : qsort right)
  where (left, right) = partition (< x) xs
```

```c
char* sieve(int n, int *c)
{
  char* sieve;
  int i, j, m;

  if(n < 2)
    return NULL;

  *c = n-1; /* primes count */
  m = (int) sqrt((double) n);

  /* calloc initializes to zero */
  sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1;
  sieve[1] = 1;
  for(i = 2; i <= m; i++)
    if(!sieve[i])
      for (j = i*i; j <= n; j += i)
        if(!sieve[j]){
          sieve[j] = 1;
          --(*c);
        }
  return sieve;
}
```

```c
char* sieve(int n, int *c)
{
  char* sieve;
  int i, j, m;

  if(n < 2)
    return NULL;

  *c = n-1; /* primes count */
  m = (int) sqrt((double) n);

  /* calloc initializes to zero */
  sieve = calloc(n+1,sizeof(char));
  sieve[0] = 1;
  sieve[1] = 1;
  for(i = 2; i <= m; i++)
    if(!sieve[i])
      for (j = i*i; j <= n; j += i)
        if(!sieve[j]){
          sieve[j] = 1;
          --(*c);
        }
  return sieve;
}
```

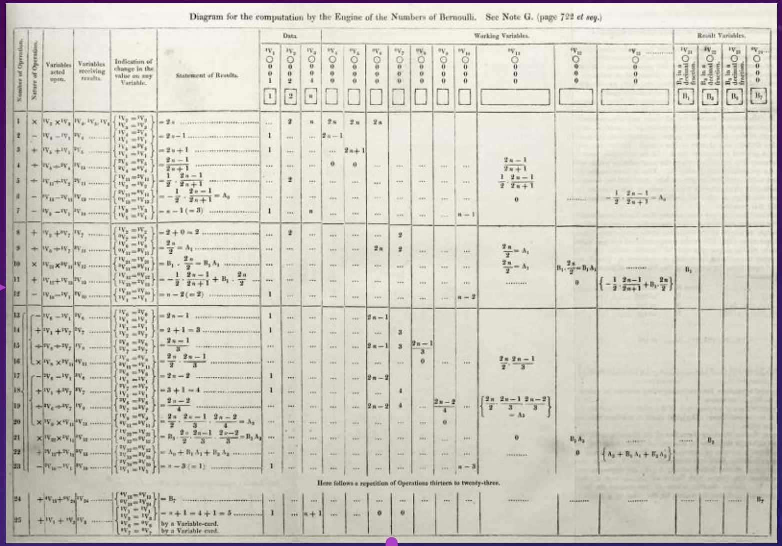
```haskell
primes = filterPrime [2..]
 where
  filterPrime (p:xs) = p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

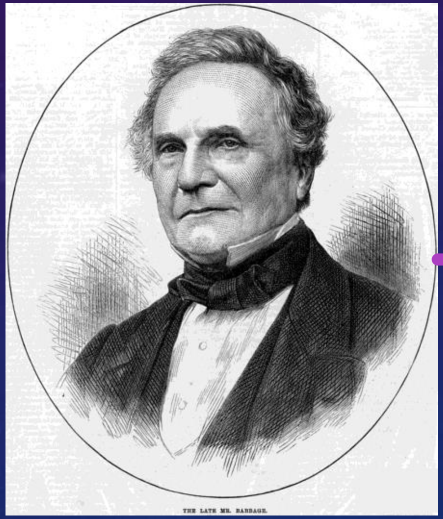
```
r = requests.get('http://chrisdone.com/ontime.csv.zip', stream=True)
with open("flights.csv", 'wb') as f:
  for chunk in r.iter_content(chunk_size=1024):
    if chunk:
      f.write(chunk)

zf = zipfile.ZipFile("flights.csv.zip") filename = zf.filelist[0].filename fp = zf.extract(filename) df = pd.read_csv(fp, parse_dates="FL_DATE").rename(columns=str.lower)
```

```python
r = requests.get('http://chrisdone.com/ontime.csv.zip', stream=True)
with open("flights.csv", 'wb') as f:
 for chunk in r.iter_content(chunk_size=1024):
  if chunk:
   f.write(chunk)

zf = zipfile.ZipFile("flights.csv.zip") filename = zf.filelist[0].filename fp = zf.extract(filename) df = pd.read_csv(fp, parse_dates="FL_DATE").rename(columns=str.lower)
```

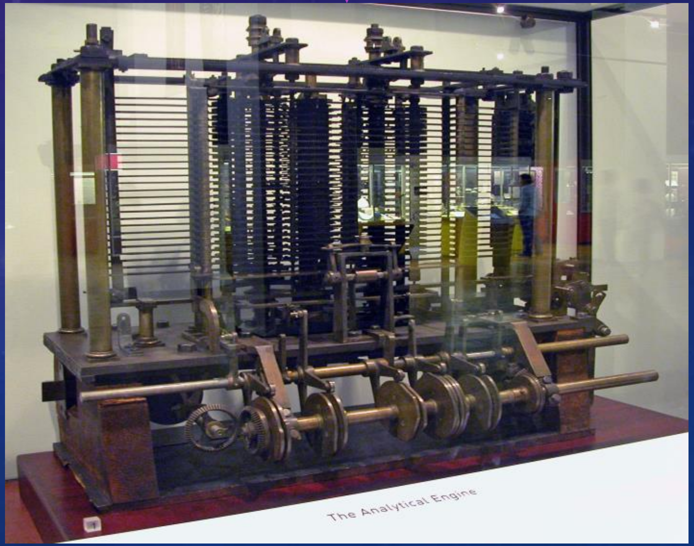
```haskell
main = runResourceT $
 httpSource "http://chrisdone.com/ontime.csv.zip" responseBody
 .| zipEntryConduit "ontime.csv“
 .| fromCsvConduit @("fl_date" := Day, "tail_num" := String) (set #downcase True csv)
 .| dropConduit 10
 .| takeConduit 5
 .> tableSink
```

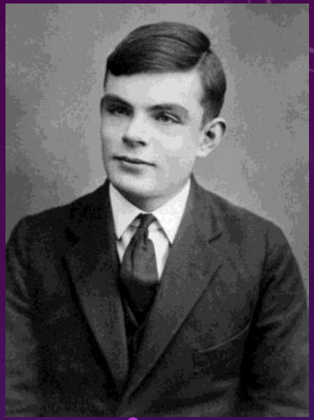
Ada Lovelace





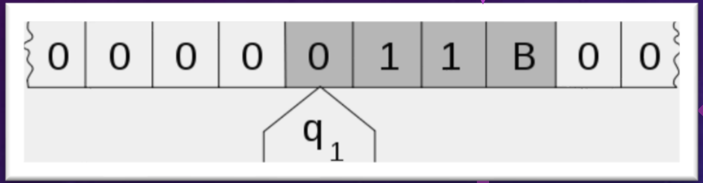
Charles Babbage





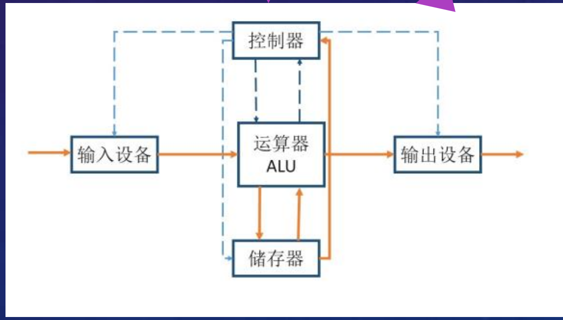
Alan Turing

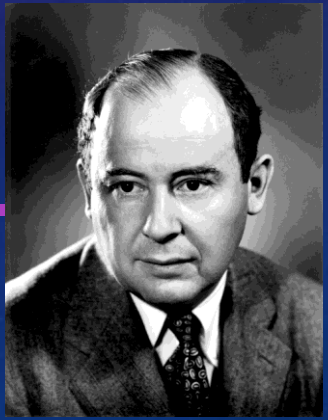
Alonzo Church



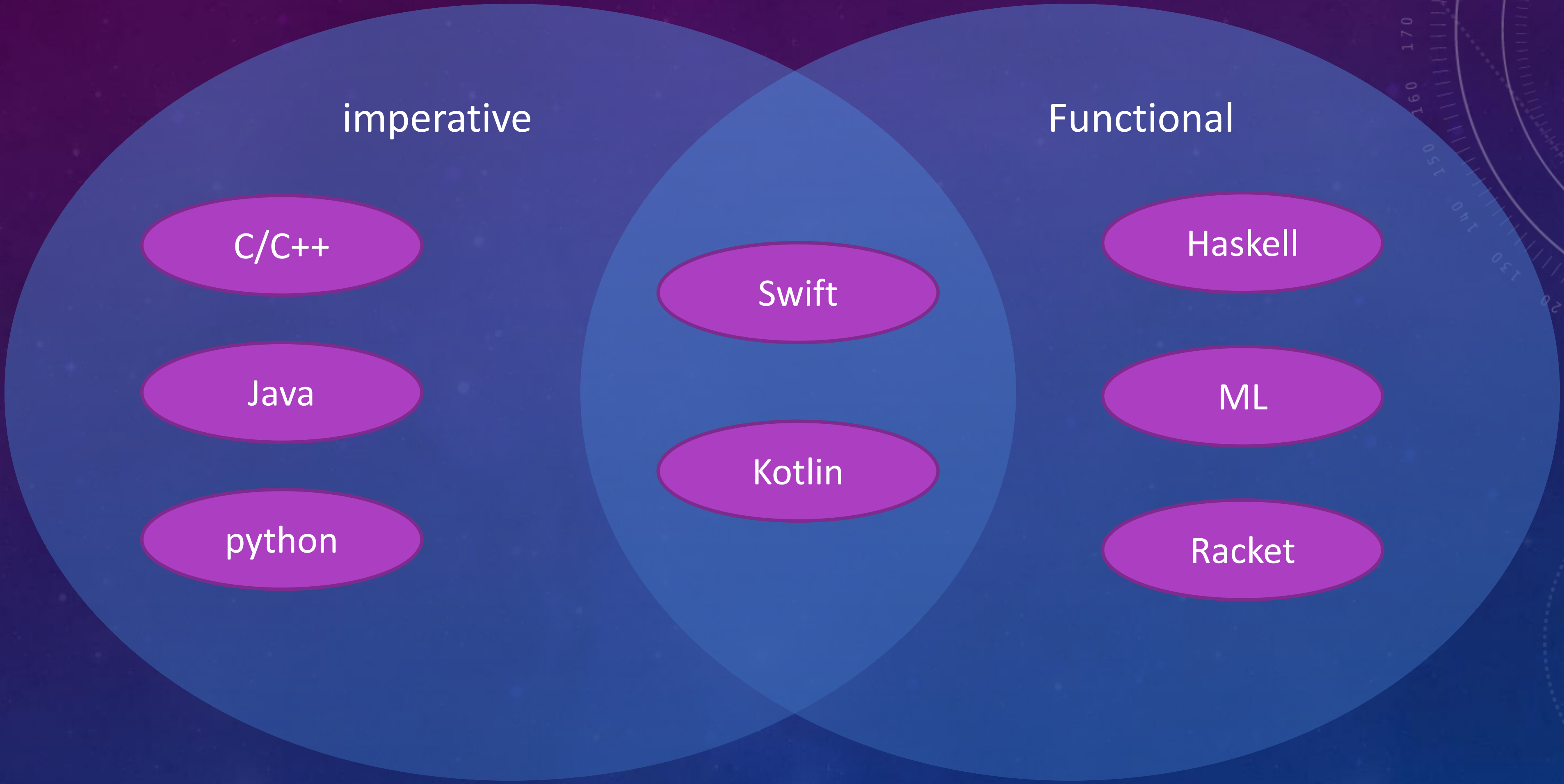
Church–Turing thesis

```
0  := λf.λx.x
1  := λf.λx.f x
2  := λf.λx.f (f x)
3  := λf.λx.f (f (f x))
```





John Von Neumann


imperative
Functional
C/C++
Java
python
Swift
Kotlin
Haskell
ML
Racket

WHAT IS FUNCTIONAL PROGRAMMING

- What to do not how to do
- Immutable, no state
- Recursive, no loop
- Function is first class object
- Type system
- Category theory

函数式编程思维：

函数式编程关心类型（代数结构）之间的关系，以及如何将这些关系组合起来，用数学的构造主义来构造程序

THE OLD CONVENTIONAL IDEAS

Imperative Programming

- Fast
- Easy to learn
- No GC
- A lots of industry application

Functional Programming

- Slow
- Difficult to learn – Immutable, recursive, cats
- Have GC
- Less industry application

THE WORLD IS CHANGED

- Multi-core processor
- GPU and DSP
- Much more complex task
- Cloud and distribution

- How to programming with parallel and concurrent
- How to write safety and stability program
- How to transfer the program to other remote computer

WHY HASKELL

- The one ring lord of rings
- Type system – System F
- Lazy evaluation
- Category theory
- A lots of packages
- Enough performance for normal application – 2~5 times less than C/C++
- A industry level compiler -- ghc

Haskell Brooks Curry

READY TO START HASKELL

- Install the Haskell-platform from https://www.haskell.org/downloads
- Launch ghci
- Or use ghc to compile the Haskell program

  ghc –make –O2 example.hs

VARIABLE AND FUNCTION, OPERATOR

- 2 :: Int, ‘a’ :: Char, “Hellow, world!” :: String
- a :: Int

  a = 2
- add :: Int -> Int -> Int

  add a b = a + b
- mult2 a = a * 2

  catString s1 s2 = s1 ++ s2

FUNCTION APPLY

> add 2 3

5

> mult2 3

6

> mult2 $ 3

6

> catString “Hello “ “world!”

“Hello world!”

TYPES AND TYPE VARIABLE

- Primary types: Int, Integer, Float, Double, Char, BOOL, Word
- Container types: [], Map, IntMap, Tree, Vector, Array
- type String = [Char]
- A type variable is a in type [a], Maybe a

  head :: [a] -> a, length :: [a] -> Int

  fst :: (a, b) -> a, snd :: (a, b) -> b

TYPE CONSTUCTION

A rich type is constructed by other types

```
data Maybe a = Nothing
             | Just a
             deriving(Eq, Show)

data [a] = [] | a : [a]

data Gendar = Male | Female  deriving(Eq, Ord, Show)
data Person = Person {
    name :: String
  , age  :: Int
  , gendar :: Gendar
  }
```

TYPE CLASS

A type constraint

```
class Eq a where
  (==) :: a -> a -> Bool
  a == b = not (a /= b)

  (/=) :: a -> a -> Bool
  a /= b = not (a == b)

class Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

TYPE AND NEWTYPE

Type alias and newtype for wrapper type that real same as original type

```
type Name = String
type Age  = Int

newtype Fd = Fd CInt
newtype State s a = State { runState :: s -> (a, s) }

newtype Any = Any { getAny :: Bool } deriving(Show)
instance Monoid Any where
  mempty  = Any False
  mappend = (||)

newtype All = All { getAll :: Bool } deriving(Show)
Instance Monoid All where
  mempty  = All True
  mappend = (&&)
```

FUNCTION USAGE – PATTERN MATCHING & GUARDS

Pattern matching

```
sayMe :: (Integral a) => a -> String
sayMe 1 = “One!”
sayMe 2 = “Two!”
sayMe 3 = “Three!”
sayMe x = “Not between 1 and 3”

addVectors (Num a) => (a, a)
                   -> (a, a)
                   -> (a, a)
addVectors (x1, y1) (x2, y2) =
  (x1 + x2, y1 + y2)
```

Guards

```
myCompare :: (Ord a) => a -> a -> Ordering
myCompare a b
   | a > b     = GT
   | a == b    = EQ
   | otherwise = LT

isOdd (Integral a) => a -> Bool
isOdd a
   | a `mod` 2 /= 0 = True
   | otherwise      = False
```

FUNCTION USAGE – LET, WHERE AND CASE

let and where

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea  = pi * r ^ 2
  in  sideArea + 2 * topArea


qsort []     = []
qsort (x:xs) = qsort left
             ++ (x : qsort right)
  where
    (left, right) = partition (< x) xs
```

case expression

```
describeList :: [a] -> String
describeList xs = “The list is ”
                ++ case xs of
                     []  -> “empty.”
                     [x] -> “a singleton list.”
                     xs  -> “a longer list.”

describeList :: [a] -> String
describeList xs = “The list is ” ++ what xs
  where what []  = “empty.”
        what [x] = “a singleton list.”
        what xs  = “a longer list.”
```

HIGH ORDER FUNCTION

A function that takes other functions as arguments or returns a function as result.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

CURRY AND UNCURRY

All multi-parameters function is can be treated as single parameter function

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f a b = f (a, b)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a, b) = f a b
```

FUNCTION COMPOSITION

The large function can be constructed by small functions

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

id :: a -> a
id x = x

oddSquareSum :: Integer
oddSquareSum =
    sum . takeWhile (<10000)
  . filter odd . map (^2) $ [1..]
```

Laws

- id . f = f
- f . id = f
- (f . g) . h = f . (g . h)

RECURSIVE

Who are you like?

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

sum :: (Num a) => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

maximum :: (Ord a) => [a] -> a
maximum [] = error “maximum of empty list”
maximum [x] = x
maximum (x:xs) = x `max` (maximum xs)
```

FUNCTION ABSTRACTION

A general function that can accept a function parameter

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = x `f` foldr f z xs

length :: [a] -> Int
-- length = foldr (\a len -> 1 + len) 0
length = foldr ((+) . const 1) 0

sum :: (Num a) => [a] -> a
sum = foldr (+) 0

product :: (Num a) => [a] -> a
product = foldr (*) 1
```

FUNCTOR AND MAP

A normal function can lift to a container transformation function

```
allAddBy2 :: (Num a) => [a] -> [a]
allAddBy2 [] = []
allAddBy2 (x:xs) = (x + 2) : allAddBy2 xs

allCheckOdd :: (Integral a) => [a] -> [Bool]
allCheckOdd [] = []
allCheckOdd (x:xs) = isOdd x : allCheckOdd xs

map :: (a -> b) -> [a] -> [b]
map f = foldr ((:) . f) []
-- map f [] = []
-- map f (x:xs) = f x : map f xs
```

FUNCTOR AND MAP

A normal function can lift to a container transformation function

```
allAddBy2 :: (Num a) => [a] -> [a]
allAddBy2 [] = []
allAddBy2 (x:xs) = (x + 2) : allAddBy2 xs

allCheckOdd :: (Integral a) => [a] -> [Bool]
allCheckOdd [] = []
allCheckOdd (x:xs) = isOdd x : allCheckOdd xs

map :: (a -> b) -> [a] -> [b]
map f = foldr ((:) . f) []
-- map f [] = []
-- map f (x:xs) = f x : map f xs

class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
allAddBy2 = map (+2)
```

```
allCheckOdd = map isOdd
```

Laws

- fmap id = id
- fmap (f . g) = (fmap f) . (fmap g)

APPLICATIVE

A context computation function can applied by context value

```
class Functor f => Applicative f where
  pure  :: a -> f a

  (<*>) :: f (a -> b) -> f a -> f b

(<$>) :: (a -> b) -> f a -> f b
(<$>) = fmap
```

Laws

- pure id <*> v = v
- pure f <*> pure x = pure (f x)
- f <*> pure y = pure ($ y) <*> f
- pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

APPLICATIVE EXAMPLES

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something

Instance Applicative [] where
  pure a = [a]
  fs <*> xs = [f x | f <- fs, x <- xs]

sequenceA :: Applicative f => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

MONAD

A context value can apply to a monadic function

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

  (>>=)  :: m a -> (a -> m b) -> m b
  m >>= f = join (fmap f m)

  join   :: m (m a) -> m a
  join m = m >>= id
```

Laws

- join . return   = id
- join . fmap return = id
- join . join = join . fmap join

MONAD EXAMPLES

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  (Just a) >>= f = f a

Instance Monad [] where
  return a = [a]
  as >>= f = concatMap f as
```

MONAD EXAMPLES

```
type Birds = Int
type Pole = (Birds, Birds)

leftLand :: Birds -> Pole -> Maybe Pole
leftLand n (l, r)
  | abs ((l + n) - r) < 4 = Just ((l + n), r)
  | otherwise             = Nothing

rightLand :: Birds -> Pole -> Maybe Pole
rightLand n (l, r)
  | abs (l - (r + n)) < 4 = Just (l, (r + n))
  | otherwise             = Nothing
```

MONAD EXAMPLES

```
routine :: Maybe Pole
routine = case landLeft 1 (0,0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```

```
routine :: Maybe Pole
routine = landLeft 1 (0, 0)
        >>= (\p1 -> (landRight 4 p1
        >>= (\p2 -> (landLeft 2 p2
        >>= (\p3 -> landLeft 1 p3)))))
```

MONAD DO NOTATION

```
routine :: Maybe Pole
routine = landLeft 1 (0, 0)
        >>= (\p1 -> (landRight 4 p1
        >>= (\p2 -> (landLeft 2 p2
        >>= (\p3 -> landLeft 1 p3)))))
```

```
routine :: Maybe Pole
routine = do
        p1 <- landLeft 1 (0, 0)
        p2 <- landRight 4 p1
        p3 <- landLeft 2 p2
        landLeft 1 p3
```

IO MONAD

We can interactive with real world through IO Monad

```
main :: IO ()
main = do
  putStrLn “Hello, World!”
  putStrLn “Please input youre name:”
  name <- getLine
  putStrLn “Good morning, ” ++ name
```

COMONAD

The comonadic functions can be applied context value

```
class Functor w => Comonad w where
  extract   :: w a -> a

  duplicate :: w a -> w (w a)
  duplicate = extend id

  extend    :: (w a -> b) -> w a -> w b
  extend f = fmap f . duplicate
```

Laws

- extract . duplicate = id
- fmap extract . duplicate = id
- duplicate . duplicate = fmap duplicate . duplicate

COMONAD EXAMPLES

```
data Stream a = a :> Stream a

tail :: Stream a -> Stream a
tail (_ :> as) = as

tails :: Stream a -> Stream (Stream a)
tails w = w :> tails (tail w)

instance Comonad Stream where
  extract = head
  duplicate = tails
  extend f w = f w :> extend f (tail w)
```

无用之用 -- IDENTITY

```
newtype Identity a = Identity { getIdentity :: a }

Instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance Applicative Identity where
  pure = Identity
  (Identity f) <*> (Identity a) = Identity (f a)

Instance Monad Identity where
  return = Identity
  (Identity a) >>= f = f a

Instance Comonad Identity where
  extract   = getIdentity
  duplicate = Identity
  extend f  = fmap f . Identity
```

May the fore be with you

Lightness

Darkness

$$Lan_k F\ a = \int^b D(k\ b,\ a) \times F\ b$$

$$Ran_k F\ a = \int_b Set(D(a,\ k\ b),\ F\ b)$$

```
data Lan k f a = forall b.
                 Lan (k b -> a) (f b)
```

```
newtype Ran k f a =
  Ran { getRan :: forall b. (a -> k b) -> f b }
```

REFERENCE

- Haskell趣学指南 -- 作者：Miran Lipovaca 译者：Fleurer，李亚舟，宋方睿
- Haskell函数式编程入门（第二版） -- 作者：张淞，刘长生
- Category Theory for Programmers – 作者： Bartosz Milewski

  链接： https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/
- Generic Programming with Adjunctions – 作者：Ralf Hinze