Maybe monad to handle failure

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

    (>>=) :: m a -> (a -> m b) -> m b

    return = pure
```

```
instance Monad Maybe where
    return x = Just x

    Nothing >>= f = Nothing
    Just x >>= f  =  f x
```

situation to model: birds can land on the pole , the balance is safe until the difference between left and right birds is less than 4

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

landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

it doesn’t model failure :

```
ghci> landLeft 10 (0,3)
(10,3)
gchi>(landLeft 5) . ( landRight 4) (0,0)
(5,4)
```

and also

```
ghci> (landRight -2) . (landLeft -1) . (landRight 4)
.(landLeft  1) (0,0)
(0,2)
```

seems ok, but is not

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
    | abs ((left + n)- right) < 4 = Just (left + n, right)
    | otherwise                   = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                   = Nothing
```

it works!

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
Nothing
```

using (>>=) of Maybe as a monad we can eaily compose functions

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

```
return (0,0) = Just (0,0) :: Maybe Pole
landLeft 2 :: Pole -> Maybe Pole

ghci> return (0,0) >>= landRight 2 >>= landLeft 2
 >>= landRight 2
Just (2,4)

ghci> return (0,0) >>= landLeft 1 >>= landRight 4
 >>= landLeft (-1) >>= landRight (-2)
Nothing
```

```
banana :: Pole -> Maybe Pole
banana _ = Nothing

ghci> return (0,0) >>= landLeft 1 >>= banana >>=
 landRight 1
Nothing
```

“by hand”:

```
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

pure (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft 2 >> landLeft 1
Just (4,4)
```

do-notation

```
routine :: Maybe Pole
routine = do
    start <- pure (0,0)
    first <- landLeft 1 start
    second <- landRight 4 first
    third <- landLeft 2 second
    landLeft 1 third
```

Maybe (4,4)

relation between >>= and do-notation

do {r} => r
do{x<-p; C; r} => p >>= \x-> q
where q <= do{C;r}

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    banana first
    second <- landRight 2 first
    landLeft 1 second
```

Nothing

it’s ugly, we do better with ST technique

```
newtype CP a = P (Pole -> (a,Pole))
app (P s)  p = s p

instance Monad CP where
  return x = P( \p -> (x,p))
  p >>= f = P(\s -> let (a,b) = app p s in app (f a) b)
```

```
landLeft :: Birds -> CP (Maybe Int)

landLeft n = P(\p -> aux n p)
aux n (x,y) = let z  = abs ((x+n)-y) in if z < 4 then (Just z, (x+n,y))
                                                 else (Nothing, (x,y))

landRight n = P(\p -> aux1 n p)
aux1 n (x,y) = let z  = abs ((y+n)-x) in if z < 4 then (Just z, (x,y+n))
                                                  else (Nothing, (x,y))
```

```
routine = do   landLeft 1
               landLeft 2
               landLeft (-3)
               landRight 2
```

app routine (0,0)
(Just 2, (0,2))

EX: how can we add the banana operator, now?