ABSTRACT COMONADS

FOR WHEN COMONADS AREN'T ABSTRACT ENOUGH

REVIEW

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

  duplicate :: w a -> w (w a)

  extend :: (w a -> b) -> w a -> w b
```

SELECTION QUERIES

SELECTION QUERIES

```
forall a. w a -> a
```

SELECTION QUERIES SELECT A SLOT NO MATTER WHAT'S INSIDE

E.G. `ix 5`

RUNNING A

SELECTION QUERY

RETRIEVES THE VALUE FROM THAT

SLOT

EXTENDING AN SELECTION QUERY REARRANGES SLOT POSITIONS

`extend (ix 2)`

SELECTS THE SLOT TWO TO THE RIGHT IN ALL VIEWS

SHIFTING EACH SLOT INDIVIDUALLY

SHIFTING THE WHOLE STREAM

forall a. Stream a -> a

```
λ> duplicate countStream
    (0 :> 1 :> 2 :> ...)
:> (1 :> 2 :> 3 :> ...)
:> (2 :> 3 :> 4 :> ...)
:> ...

λ> ix 2  $  duplicate countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...

λ> ix 2 <$> duplicate countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...
```

NOT A SELECTION QUERY

```
windowedAvg :: Int -> Stream Int -> Double
```

```
λ> windowedAvg 3 $ duplicate countStream
☢ Type Error: Stream (Stream Int) != Stream Int

λ> windowedAvg 3 <$> duplicate countStream
1.0 :> 2.0 :> 3.0 :> 4.0 :> 5.0 :> ...
```

```
λ> countStream
0 :> 1 :> 2 :> 3 :> 4 :> ...

-- Running a selection
λ> ix 5 countStream
5

-- Extending an selection
λ> extend (ix 5) countStream
5 :> 6 :> 7 :> 8 :> 9 :> ...
```

</foreshadowing>

NOTE ON NOTATION:

EXTEND

CONSTRUCTS & RUNS A CHAIN OF QUERIES

IT TAKES A STRUCTURE ON THE LEFT

```
(=>>) :: w a -> (w a -> b) -> w b
```

AND A QUERY ON THE RIGHT

NOTE ON NOTATION: INFIX EXTEND

```
λ> extend (ix 2) countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...

λ> countStream =>> ix 2
2 :> 3 :> 4 :> 5 :> 6 :> ...
```

NOTATION (EXTEND)

```
(=>>) :: w a -> (w a -> b) -> w b
```

```
λ> countStream
0 :> 1 :> 2 :> 3 :> 4 :> ...
```

```
-- =>> is left associative!
λ> countStream =>> ix 2 =>> ix 2
λ> (countStream =>> ix 2) =>> ix 2
4 :> 5 :> 6 :> 7 :> 8 :> ...
```

```
λ> countStream =>> ix 2 =>> takeS 3
[3,4,5] :> [4,5,6] :> [5,6,7] :> [6,7,8] :> [7,8,9] :> ...

λ> countStream =>> ix 2 =>> takeS 3 & extract
[3,4,5]
```

NOTE ON NOTATION:
COMPOSITION

COMPOSES QUERIES

CO-KLEISLI COMPOSITION

```
(=>>) :: w a
      -> (w a -> b)
      -> w b
```

VS.

```
(=>=) :: (w a -> b) -- Query 1
      -> (w b -> c) -- Query 2
      -> (w a -> c) -- Composed Queries
```

```
λ> ix 2 =>= ix 2 =>= ix 2 $ countStream
6

λ> ix 2 =>= takeS 3 $ countStream
[2,3,4]
```

QUESTIONS?

STORE

STUFF STORED BY KEY

SO WHAT'S A STORE?

WAREHOUSE

LABELED SHELVES WITH ITEMS

WELL STOCKED (EVERY SHELF IS OCCUPIED)

FORKLIFT IS ALWAYS AT A SHELF

WAREHOUSE

HAS AN

INVENTORY

AND A

FORKLIFT

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

WAREHOUSE QUERIES

POS

ASKS WHICH SHELF THE FORKLIFT IS AT

POS W

=

1

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

EXTRACT

GETS THE ITEM AT THE CURRENT SHELF

EXTRACT W

=

BOOKS

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

PEEK GETS THE ITEM FROM A SPECIFIC SHELF

PEEK 3 W

=

LAPTOPS

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

PEEKS

GETS THE ITEM

FROM A SHELF

RELATIVE TO CURRENT POSITION

PEEKS (+1) W

=

GUITARS

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

AS CODE:

```
data Store s a = Store (s -> a) s
```

WAREHOUSE EXAMPLE

```
inventory :: M.Map Int String
inventory =
    M.fromList [ (0, "Fidget spinners")
               , (1, "Books")
               , (2, "Guitars")
               , (3, "Laptops")
               ]
```

```
λ> lookup 0 inventory
Just "Fidget spinners"

λ> lookup 42 inventory
Nothing
```

EVERY KEY HAS A VALUE

#-2	Nothing
#-1	Nothing
#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops
#4	Nothing
#5	Nothing

```haskell
warehouse :: Store Int (Maybe String)
warehouse =
  store (\shelf -> M.lookup shelf inventory) 1
```

```
λ> pos warehouse
1

λ> peek 0 warehouse
Just "Fidget Spinners"

λ> peeks (+1) warehouse
Just "Guitars"
```

#0	Fidget Spinners
	Books
#2	Guitars
#3	Laptops

SEEK

MOVES THE FORKLIFT TO A SPECIFIC SHELF

SEEK 3 W

#0	Fidget Spinners
#1	Books
#2	Guitars
	Laptops

SEEKS

MOVES THE FORKLIFT

RELATIVE TO CURRENT POSITION

SEEKS (+1) W

#0	Fidget Spinners
	Books
	Guitars
#3	Laptops

STORE

A.K.A. CO-STATE

A SLOT FOR EVERY KEY

A VIEW FROM EVERY KEY

EXTRACT THE VALUE AT YOUR KEY

STORE HAS A GETTER AND A KEY

STORE QUERIES

POS

```
pos :: Store s a -> s
```

GETS THE CURRENT KEY

EXTRACT

```
extract :: Store s a -> a
```

GETS THE SLOT AT THE CURRENT KEY

PEEK

```
peek :: s -> Store s a -> a
```

EXTRACTS THE SLOT AT ANOTHER KEY

* SELECTION QUERY!!

```haskell
squared :: Store Int Int
squared =
  Store (\x -> x^2) -- The getter
        10          -- The current key
```

```haskell
squared :: Store Int Int
squared = Store (\x -> x^2) 10

λ> pos squared
10

λ> extract squared
100 -- 10^2

λ> peek 2 squared
4 -- 2^2
```

PEEKS

```
peeks :: (s -> s) -> Store s a -> a
```

GET A SLOT AT A RELATIVE POSITION TO THE CURRENT KEY

```
λ> pos squared
10

λ> peeks (+2) squared
144 -- (10 + 2)^2
```

RELATIVE QUERIES

LIKE PEEKS

HAVE DIFFERENT RESULTS

DEPENDING ON THE CURRENT VIEW

ABSOLUTE QUERIES

IGNORE

THE CURRENT VIEW

peek 2

IS AN ABSOLUTE QUERY

STORE MUTATIONS

SEEK

```
seek :: s -> Store s a -> Store s a
```

FOCUS THE SLOT AT THE GIVEN KEY

SEEK IS AN ABSOLUTE MUTATION

STORE

```
λ> pos $ seek 5 squared
5

λ> pos $ seek 3 squared
3

λ> extract $ seek 3 squared
9
```

SEEKS

```
seeks :: (s -> s) -> Store s a -> Store s a
```

THE RELATIVE VERSION OF SEEK


EXPERIMENT

IT'S REALLY ANNOYING
TO PEEK AT LOTS OF NEARBY VALUES

WHAT IF WE WANT TO KEEP TRACK OF THE RESULTS?

OR EVEN

FAIL

BASED ON OUR POSITION?

EXPERIMENT GETS A BUNCH OF VALUES RELATIVE TO YOUR POSITION

EXPERIMENT

```
experiment :: Functor f
           => (s -> f s)
           -> Store s a
           -> f a
```

STORE EXAMPLE: SQUARED

```
λ> experiment (\n -> [n - 10, n + 10]) squared
[ 0     -- (10-10)^2
, 400   -- (10+10)^2
]
```

```
aboveZero n | n > 0      = Just n
            | otherwise = Nothing
λ> experiment aboveZero (seeks 10 squared)
Just 100

λ> experiment aboveZero (seeks (-10) squared)
Nothing
```

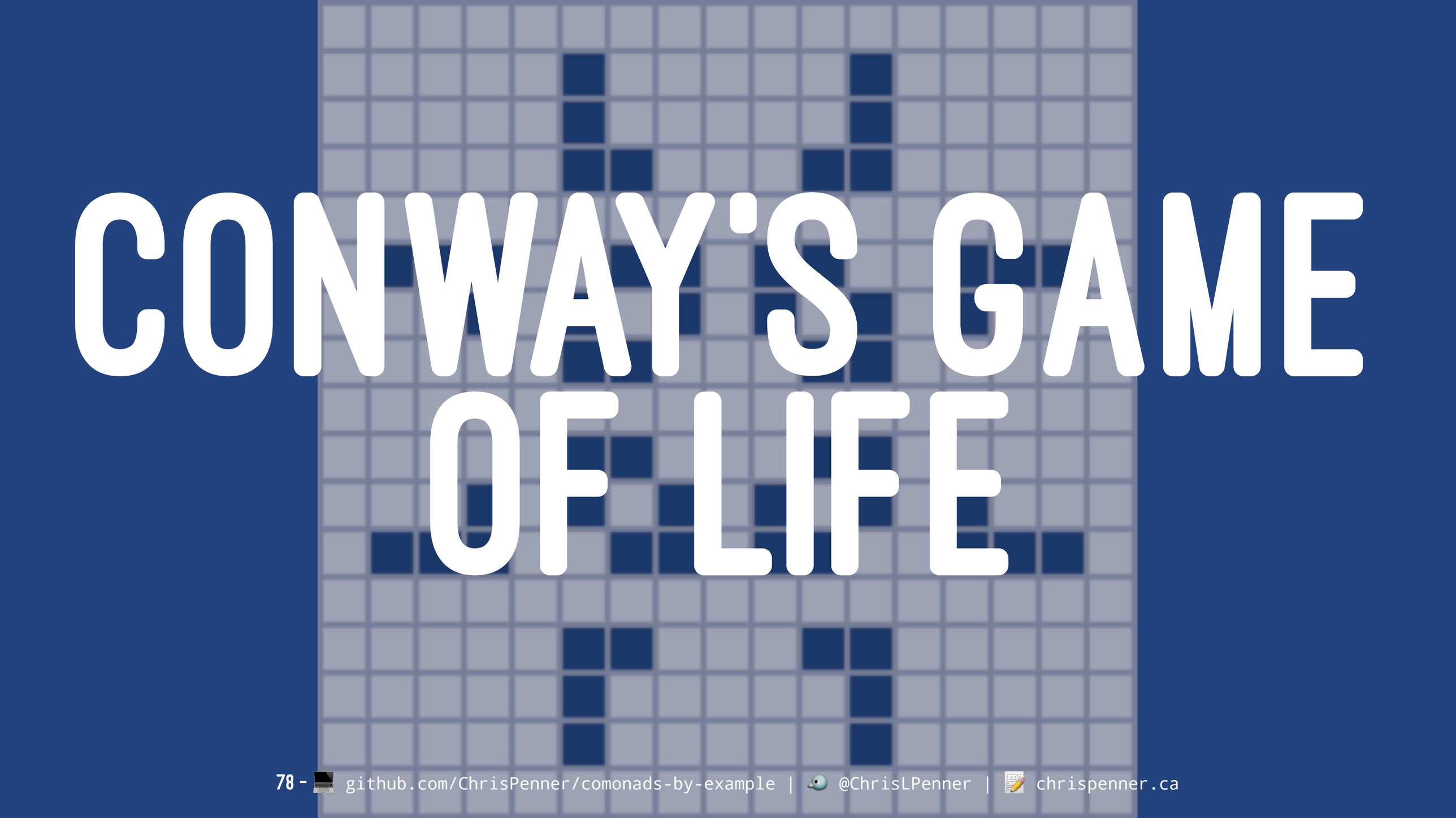
```haskell
withN :: Store Int (String, Int)
withN = squared =>> experiment (\n -> (show n, n))
```

```
λ> extract withN
("10",100)

λ> peek 5 withN
("5",25)
```

STORE EXAMPLE: SQUARED

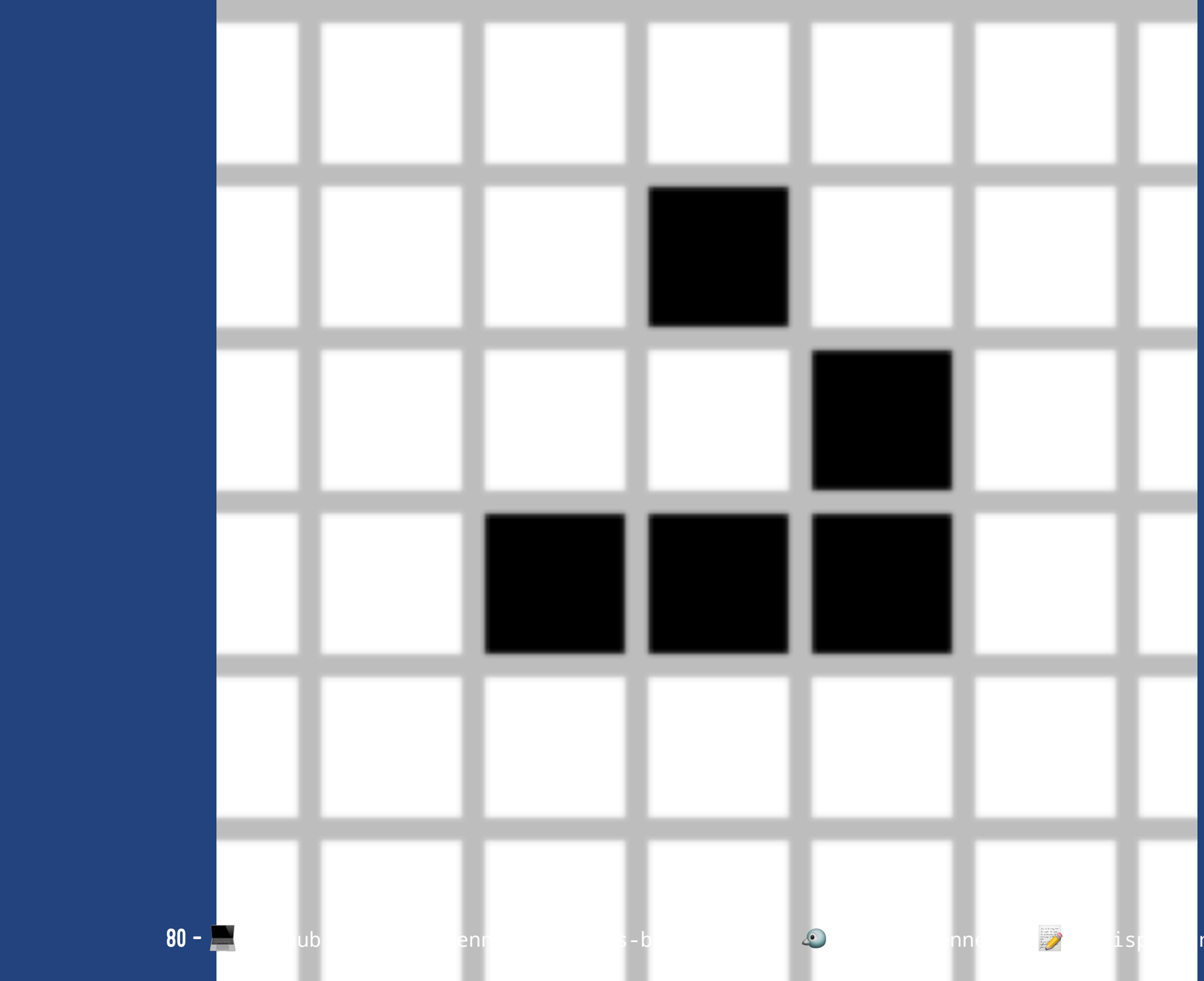
```
shifted :: Store Int (String, Int)
shifted = withN =>> peeks (+10)

λ> extract shifted
("20",400)

λ> peek 5 shifted
("15",225)
```

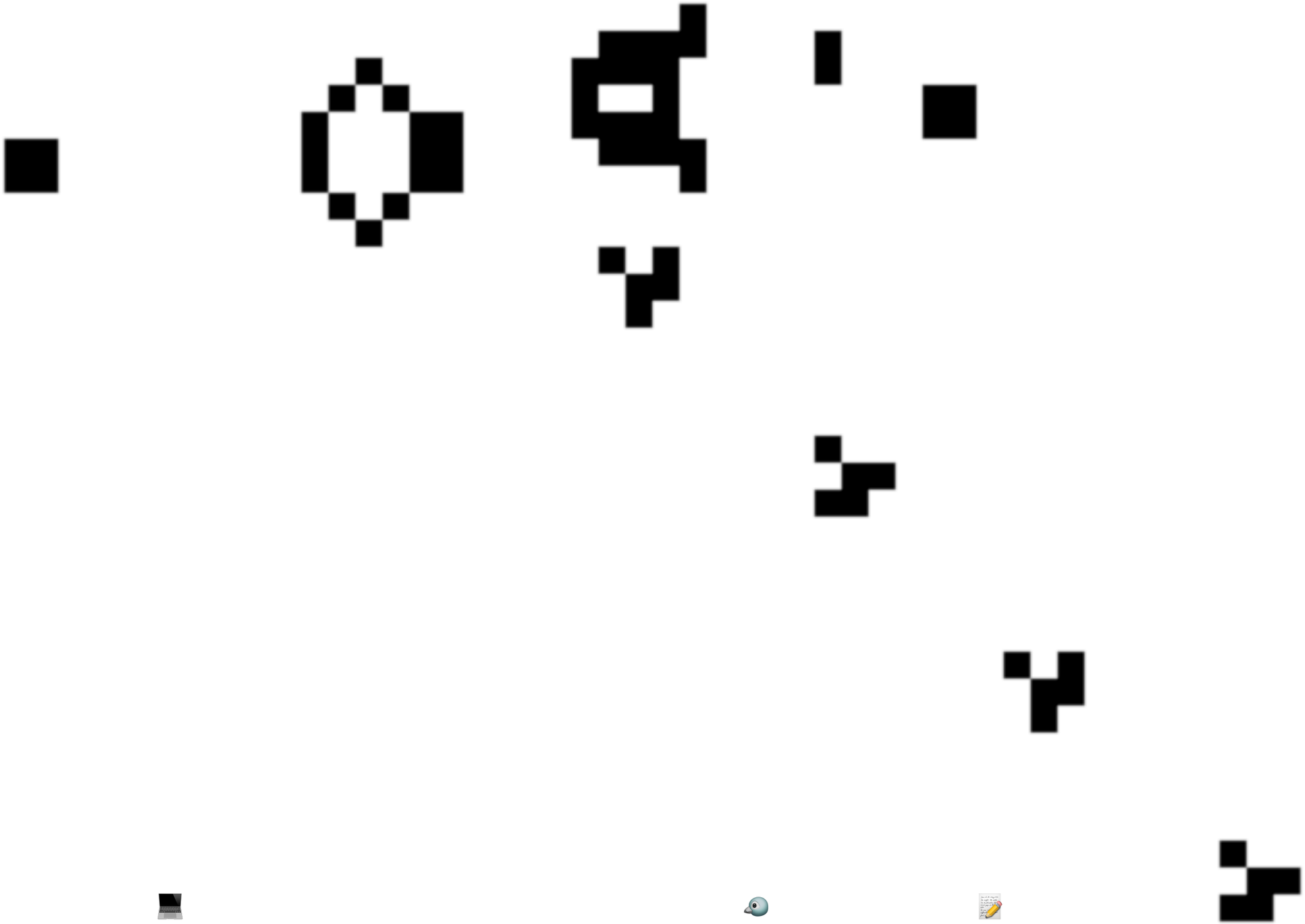
QUESTIONS?


CONWAY'S GAME
OF LIFE

RULES

- LIVING CELLS WITH 2 OR 3 LIVING NEIGHBOURS STAYS ALIVE
  - DEAD CELLS WITH 3 LIVING NEIGHBOURS COME TO LIFE
    - ALL OTHER SCENARIOS CAUSE A CELL TO DIE





WE GONNA BUILD THIS:

```
λ> animateGrid startingGrid
..#....|.#.....|..#....|.......|.......
#.#....|..##...|...#...|.#.#...|...#...
.##....|.##....|.###...|..##...|.#.#...
.......|.......|.......|..#....|..##...
.......|.......|.......|.......|.......
```

LET'S PLAN

WE NEED TO REPRESENT A 2D GRID OF CELLS

EACH CELL

IS EITHER

ALIVE

OR

DEAD

SO I'M THINKING

```
Store (Sum Int, Sum Int) Bool
```

```
(row, column) -> alive?
```

QUESTIONS?

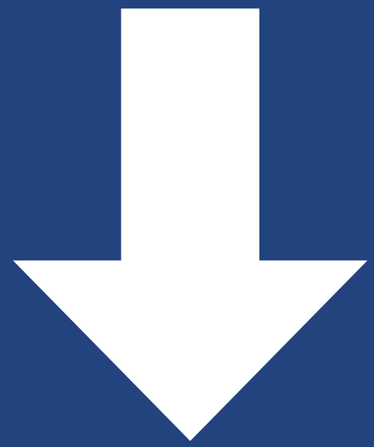
```haskell
startingGrid :: Store (Sum Int, Sum Int) Bool
startingGrid = store checkAlive (0, 0)
  where
    checkAlive :: (Sum Int, Sum Int) -> Bool
    checkAlive coord = S.member coord livingCells

    livingCells :: S.Set (Sum Int, Sum Int)
    livingCells = S.fromList [(1, 0), (2, 1), (0, 2), (1, 2), (2, 2)]
```

```
λ> peek (Sum 0, Sum 0) startingGrid
False

λ> peek (Sum 1, Sum 0) startingGrid
True

λ> putStrLn . drawGrid 3 $ startingGrid
..#
#.#
.##
```

WE WANT A MUTATION WHICH SIMULATES A STEP

AS ALWAYS

LET'S BREAK IT DOWN



INTO QUERIES

QUERY:

```
checkCellAlive :: Store (Sum Int, Sum Int) Bool
               -> Bool
```

IF WE CAN COMPUTE WHETHER

ONE CELL

IS ALIVE:

WE CAN EXTEND IT TO

ALL CELLS

😮 WOW 😮

OUR RECIPE NEEDS THE FOLLOWING INGREDIENTS:

1. WHETHER THE CURRENT CELL IS ALIVE OR DEAD
2. NUMBER OF LIVING NEIGHBOURS

QUERY TO CHECK THE CURRENT CELL

ANY IDEAS?

EXTRACT

...NEXT

NUMBER OF LIVING NEIGHBOURS

```
numLivingNeighbours :: Store (Sum Int, Sum Int) Bool
                    -> Int
```

WE NEED TO CHECK THE CELLS NEXT DOOR

REMEMBER

EXPERIMENT?

```
experiment :: Functor f
           => (s -> f s)
           -> Store s a
           -> f a
```

WE CAN CHECK OUR NEIGHBOURS ALL AT ONCE!

FORMULATING OUR EXPERIMENT

```
neighbourLocations :: (Sum Int, Sum Int) -> [(Sum Int, Sum Int)]
neighbourLocations location = mappend location <$> [
      (-1, -1), (-1, 0), (-1, -1)
    , (0,  -1),          (0,  1)
    , (1,  -1), (1, 0),  (1,  1)
]
```

NOW TO TEST OUR HYPOTHESIS:

```haskell
numLivingNeighbours :: Store (Sum Int, Sum Int) Bool -> Int
numLivingNeighbours w =
    getSum . foldMap toCount . experiment neighbourLocations $ w
  where
    toCount :: Bool -> Sum Int
    toCount False = Sum 0
    toCount True = Sum 1
```

ALL SET!

```haskell
checkCellAlive :: Store (Sum Int, Sum Int) Bool -> Bool
checkCellAlive grid =
    case (extract grid, numLivingNeighbours grid) of
        (True, 2) -> True
        (_,    3) -> True
        _         -> False
```

QUESTIONS?

NOW WE EXTEND IT TO ALL THE CELLS!

```haskell
step :: Store (Sum Int, Sum Int) Bool
     -> Store (Sum Int, Sum Int) Bool
step = extend checkCellAlive
```

A NOTE ON PERFORMANCE

QUESTIONS?

CHALLENGE A: IMPLEMENT COMONAD FOR STORE

```
instance Comonad (Store s) where
  extract :: Store s a -> a

  duplicate :: Store s a -> Store s (Store s a)

  extend :: (Store s a -> b) -> Store s a -> Store s b
```

CHALLENGE B: ADD A NEW RULESET FOR CONWAY'S GAME OF LIFE

END

```haskell
instance Comonad (Store s) where
  extract :: Store s a -> a
  extract (Store f s) = f s

  duplicate :: Store s a -> Store s (Store s a)
  duplicate (Store f s) =
      Store (\s' -> Store f s') s

  extend :: (Store s a -> b) -> Store s a -> Store s b
  extend g st = g <$> duplicate st
```

chrispenner.ca

github.com/ChrisPenner

 @ChrisLPenner

BONUS

EXTEND
LET'S US WRITE
FUNCTION MIDDLEWARE

```haskell
-- Precompose
normalizePath :: Store T.Text T.Text -> T.Text
normalizePath = peeks cleaned
  where
    cleaned url = T.strip . T.toLower $ url

-- Postcompose
collapseSlashes :: Store T.Text T.Text -> T.Text
collapseSlashes = T.replace "//" "/" . extract

-- Wrapping transform
showTransformation :: Store T.Text T.Text -> T.Text
showTransformation w = pos w <> ": " <> extract w

badUrl :: T.Text
badUrl = " gOOgLe.com//a/B//c  "

urlTransformer :: Store T.Text T.Text
urlTransformer = store id "" =>> collapseSlashes =>> normalizePath =>> showTransformation
```