COMONADS

COMONOIDS IN THE CATEGORY OF ENDOFUNCTORS

WHAT'S THE PROBLEM?

OUTLINE

1. Session 1
    1. Overview of Comonads
    2. Streams
    3. *BREAK*

1. Session 2
    1. Store

GET THE SRC

GITHUB.COM/CHRISPENNER/COMONADS-BY-EXAMPLE

```
git clone git@github.com:ChrisPenner/comonads-by-example
cd comonads-by-example
stack build
```


MONADS


CO-MONADS

MONADS

INTRODUCE EFFECTS

```
putStrLn :: String -> IO ( )
```

```
readFile :: FilePath -> IO String
```

```
words :: String -> [String]
```

EFFECT CONSTRUCTION

```
find :: (a -> Bool) -> [a] -> Maybe String
```

EFFECTS *

ADD A **CONTEXT** TO AN **ELEMENT**

a -> m b

*A.K.A CO-ALGEBRAS

WHAT ABOUT COEFFECTS?

DUALS

FLIP ALL THE THINGS

COEFFECTS

```
a -> m b
a <- w b
```

COEFFECTS

```
a -> m b
w a -> b
```

COEFFECTS *
ARE QUERIES
OVER A STRUCTURE

*A.K.A ALGEBRAS

```
length :: [a] -> Int
```

```
head :: [a] -> a
```

QUERY CONSTRUCTION

```
elem :: a -> ([a] -> Bool)
```

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

```
sum :: Tree Int ->
```

MONADS LET US COMPOSE EFFECTS

MONADS ARE A CONTEXT WHERE WE CAN INTRODUCE EFFECTS

COMONADS ARE A CONTEXT WHERE WE CAN RUN QUERIES

CONFUSING?

LET'S ADD SOME **CONTEXT**

LET'S LOOK AT A SPECIFIC CONTEXT

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

```haskell
data Stream a = a :> Stream a
        deriving (Functor, Foldable)
```

:> IS AN INFIX DATA CONSTRUCTOR

HELPERS

```
fromList :: [a] -> Stream a
fromList xs = go (cycle xs)
  where
    go (a:rest) = a :> go rest

countStream :: Stream Int
countStream = fromList [0..]
λ> 0 :> 1 :> 2 :> 3 :> 4 :> ...
```

LET'S WRITE SOME

QUERIES

A.K.A. COEFFECTS**

** A.K.A. ALGEBRAS

```
ix :: Int -> Stream a -> a
```

```haskell
ix :: Int -> Stream a -> a
ix n _ | n < 0 = error "whoops"
ix 0 (a :> _) = a
ix n (_ :> rest) = ix (n - 1) rest
```

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

λ> ix 0 countStream
0

λ> ix 2 countStream
2

λ> ix 1337 countStream
1337
```


You can ask me anything you want,
ask me anything.

SO WHAT??

NOW WE WANT TO WRITE DROP!

```
dropS :: Int -> Stream a -> Stream a
```

DESIRED BEHAVIOUR

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

λ> dropS 1 countStream
1 :> 2 :> 3 :> 4 :> 5 :> ...

λ> dropS 2 countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...
```

```
Stream a -> Stream b
```

IS A

MUTATION

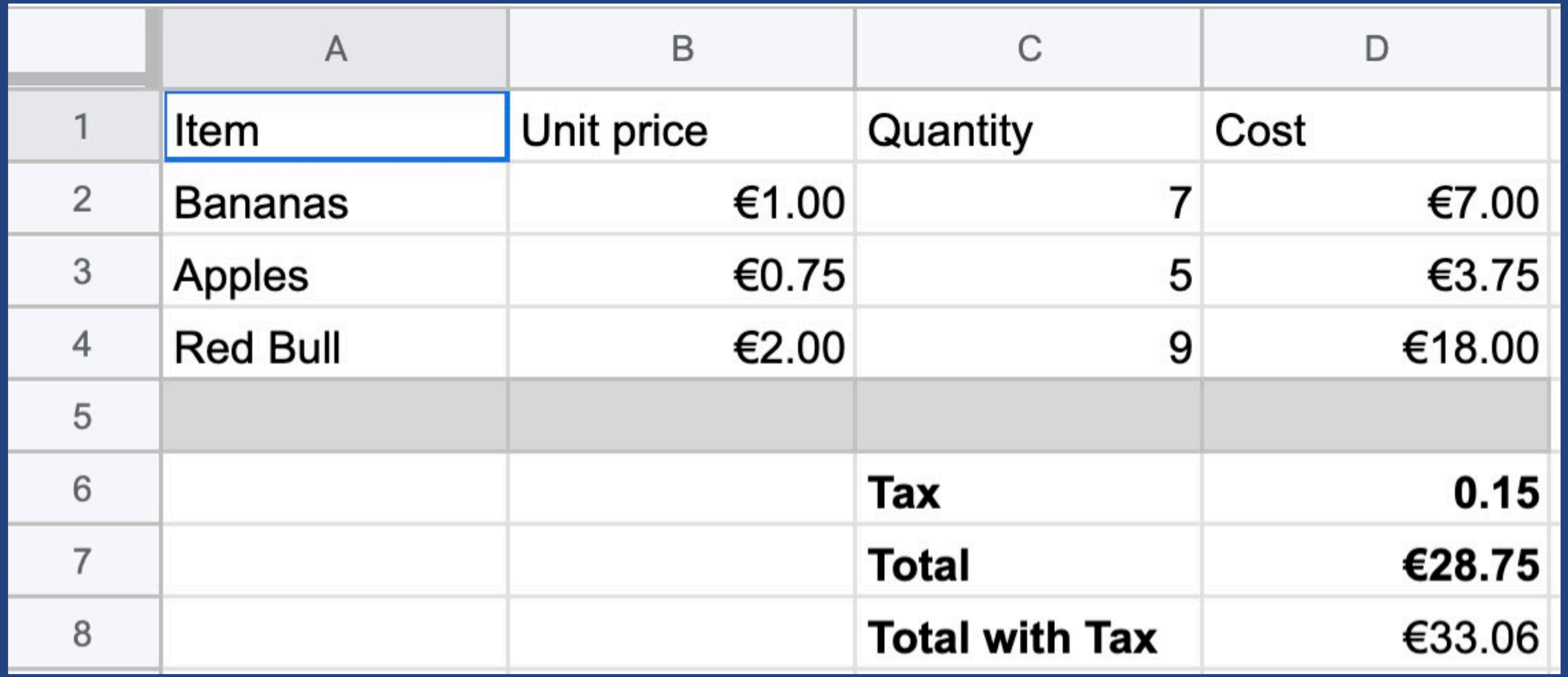
OTHER MUTATIONS

```
gaussianBlur :: Image Pixel -> Image Pixel
```

```
followPath :: [e] -> Graph e v -> Graph e v
```

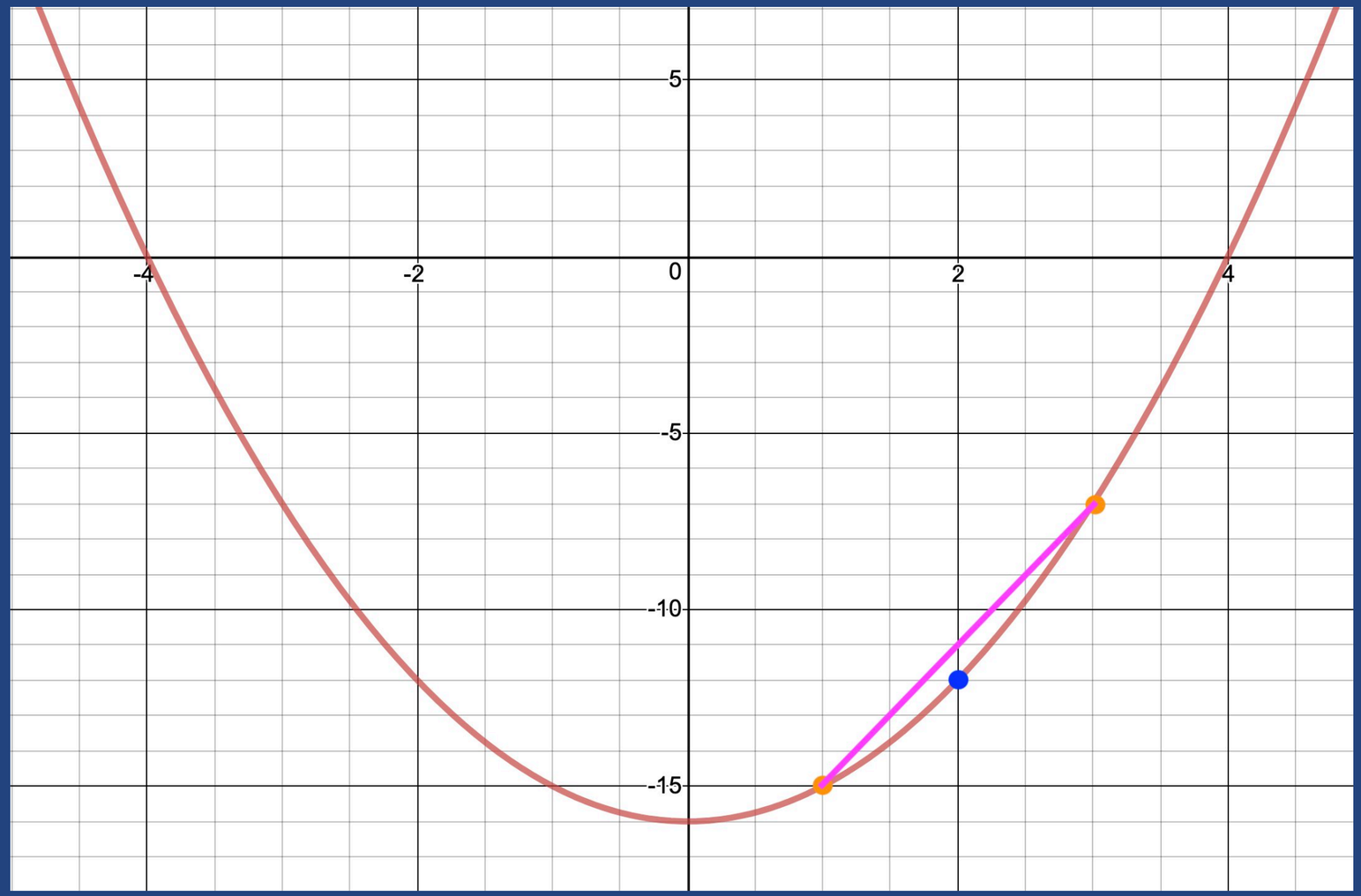
```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

```
popHeap_ :: Heap Int -> Heap Int
```



	A	B	C	D
1	Item	Unit price	Quantity	Cost
2	Bananas	€1.00	7	€7.00
3	Apples	€0.75	5	€3.75
4	Red Bull	€2.00	9	€18.00
5				
6			**Tax**	**0.15**
7			**Total**	**€28.75**
8			**Total with Tax**	€33.06

```
deleteColumn :: SpreadSheet a -> SpreadSheet a
```




```
derivative :: (Double -> Double) -> (Double -> Double)
```

ALSO

- SOLVE HILL CLIMBING PROBLEMS
- COMPUTE ROOTS USING NEWTON'S METHOD
- COMPUTE DEPENDENCY TREES
- CRUSH THE CODING INTERVIEW (RAINWATER PROBLEM)

WHERE WERE WE?

have:

```
λ> ix 2 countStream
2
```

want:

```
λ> dropS 2 countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...
```

SIMILAR??

```
ix    :: Int -> Stream a ->          a

dropS :: Int -> Stream a -> Stream a
```

CAN WE TURN A QUERY INTO A MUTATION?

BACK TO THE MONAD

JOIN

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

```
join :: m (m a) -> m
cojoin?? :: w a -> w (w a)
```

WTH IS COJOIN?

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

WUT?

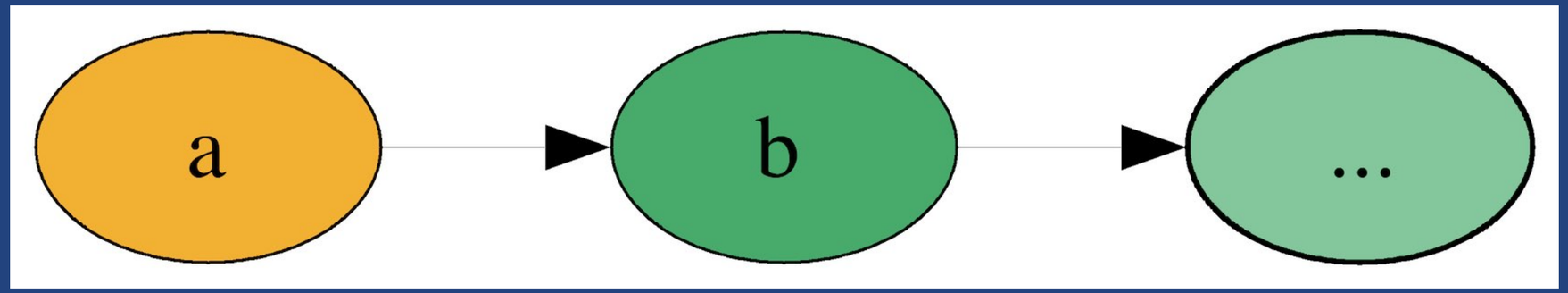
DUPLICATE
NESTS YOUR STRUCTURE
WHILE MAINTAINING
CONTEXT

*WE'LL TALK ABOUT LAWS LATER

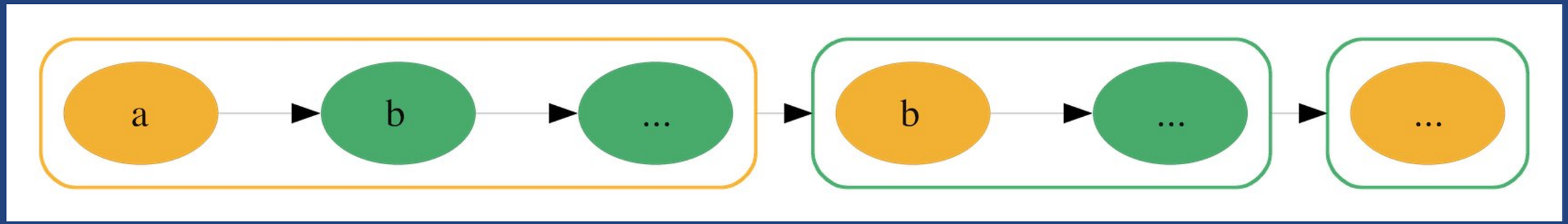
THE CONTEXT OF A STREAM IS ITS POSITION

```
duplicate :: Stream a -> Stream (Stream a)
```

stream



duplicate stream



```
λ> countStream
1 :> 2 :> 3 :> 4 :> 5 :> ...

λ> duplicate countStream
   (1 :> 2 :> 3 :> ...) -- The original stream
:> (2 :> 3 :> 4 :> ...) -- The stream viewed from the second element
:> (3 :> 4 :> 5 :> ...) -- The stream viewed from the third element
:> ...                  -- Continue ad nauseam...
```

DUPLICATE

SHOWS US ALL POSSIBLE

VIEWS

OF A STRUCTURE

EACH VIEW APPEARS IN A SLOT ACCORDING TO SOME INTUITION

* AND SOME LAWS WHICH I PROMISE WE'LL TALK ABOUT REAL SOON

THINK ABOUT EACH SLOT

```
λ> countStream
1 :> 2 :> 3 :> 4 :> 5 :> ...

λ> duplicate countStream
   (1 :> 2 :> 3 :> ...) -- The original stream
:> (2 :> 3 :> 4 :> ...) -- The stream viewed from the second element
:> (3 :> 4 :> 5 :> ...) -- The stream viewed from the third element
:> ...                  -- Continue ad nauseam...
```


SERIOUSLY NOW,
IF YOU HAVE ANY QUESTIONS,

IX FOCUSES A SLOT WITHIN THE STRUCTURE

DUPLICATE

FILLS EACH

SLOT

WITH A COPY OF THE STREAM

VIEWED FROM THAT POSITION

SO IF DUPLICATING GETS US ALL THE VIEWS

AND

IX CAN SELECT ONE OF THOSE VIEWS

THEN...

ix

+

duplicate

=

dropS

```
λ> duplicate countStream
   (1 :> 2 :> 3 :> ...) -- The original stream
:> (2 :> 3 :> 4 :> ...) -- The stream viewed from the second element
:> (3 :> 4 :> 5 :> ...) -- The stream viewed from the third element
:> ...                  -- Continue ad nauseam...

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

λ> ix 42 (duplicate countStream)
42 :> 43 :> 44 :> 45 :> 46 :> ...
```

DUPLICATE

HELPS US LIFT A

QUERY

INTO A

MUTATION

dropS

```
dropS :: Int -> Stream a -> Stream a
dropS n s = ix n (duplicate s)
```

ERROR HANDLING IS FOR NERDS 🤓

COOL COOL COOL QUESTIONS?


OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN


CAN WE GO
BACK?

CAN WE TURN A MUTATION INTO A QUERY?

CAN WE USE THIS:

```
dropS :: Int -> Stream a -> Stream a
```

TO IMPLEMENT THIS:

```
ix    :: Int -> Stream a ->        a
```

??

```
dropS :: Int -> Stream a -> Stream a
ix    :: Int -> Stream a ->          a
```

```
λ> dropS 1 countStream
1 :> 2 :> 3 :> 4 :> 5 :> ...
^
```

```
λ> ix 1 countStream
1
```

```
λ> dropS 2 countStream
2 :> 3 :> 4 :> 5 :> 6 :> ...
^
```

```
λ> ix 2 countStream
2
```

```
ix n s = ix 0 (dropS n s)
```

* BUT INFINITE LOOPS MAKE ME SAD 😭

IX IO

IS SPECIAL

IT

EXTRACTS

THE

FOCUSED SLOT

```
extract :: Stream a -> a
```

#EZ-PZ 🍋 SQUEEZY

```
extract :: Stream a -> a
extract (a :> _) = a
```

extract == ix 0

EXTRACT

IS THE

NULL/NO-OP

QUERY

*LAWS ARE SERIOUSLY COMING UP I SWEAR

```
ix n s = extract (dropS n s)
```

LAWS

THE ORIGINAL VIEW OF THE STRUCTURE MUST BE STORED IN FOCUSED SLOT WHEN DUPLICATING

```
extract (duplicate w) == w
```

THE FOCUSED SLOT OF EACH VIEW MUST MATCH THE SLOT IT'S STORED IN AFTER DUPLICATING

```
extract <$> duplicate w == w
```

```
duplicate (duplicate w)
          ==
duplicate <$> duplicate w
```

COMONADS

```haskell
class Functor w => Comonad w where
  extract   :: w a -> a
  duplicate :: w a -> w (w a)
  extend    :: (w a -> b) -> w a -> w b
{-# MINIMAL extract, (duplicate | extend) #-}
```

* EXTEND LIFTS FROM A QUERY TO A MUTATION

COMONADS ARE STRUCTURES OR SPACES

THAT WE CAN

VIEW

FROM DIFFERENT

FOCUSES

THE VIEW CAN BE INCOMPLETE

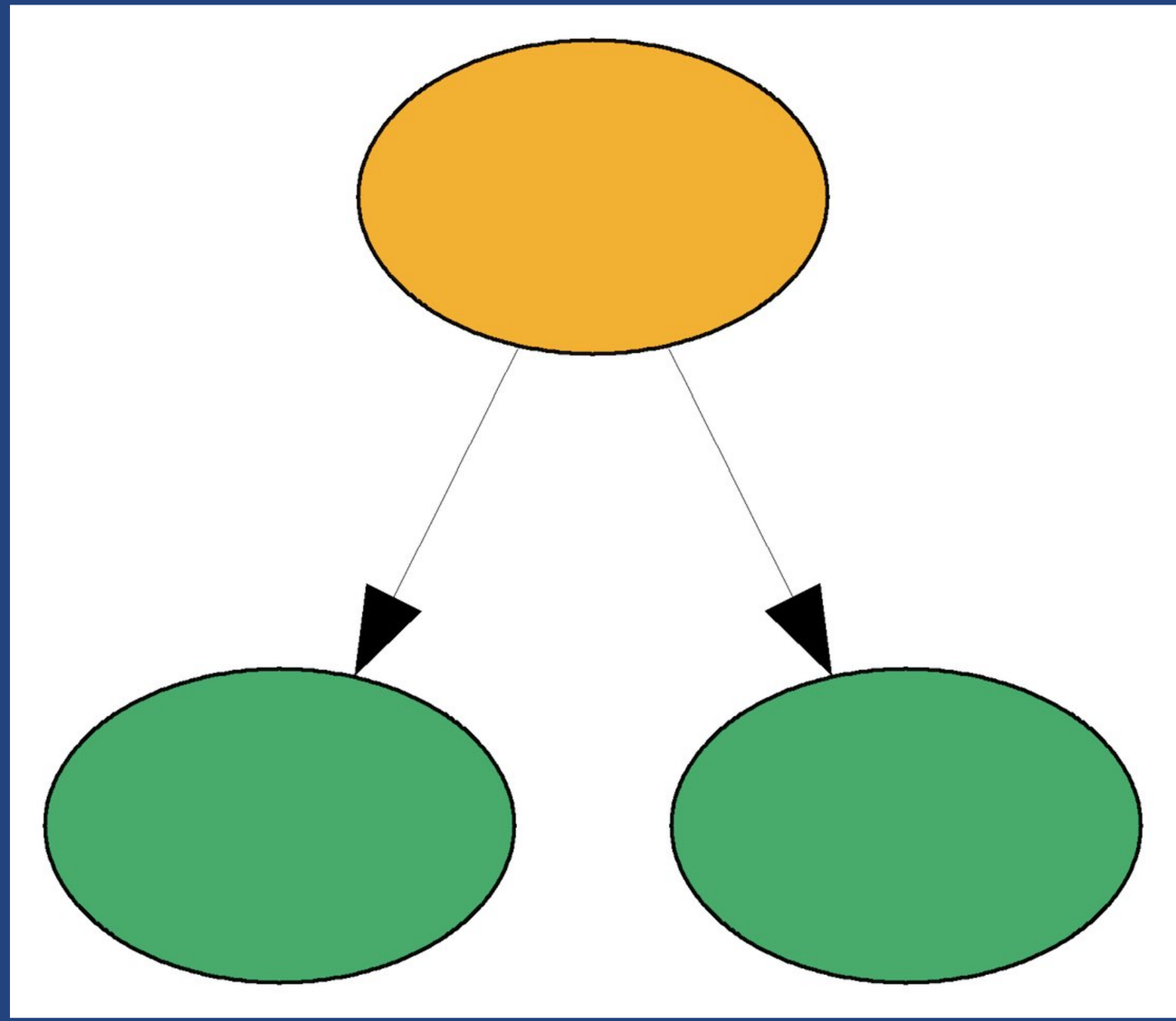
*SOME ELEMENTS MAY BE MISSING FROM SOME VIEWS

TREES

NODES ARE THE SLOTS

VIEWS ARE SUBTREES

EXTRACT THE SUBTREE'S ROOT

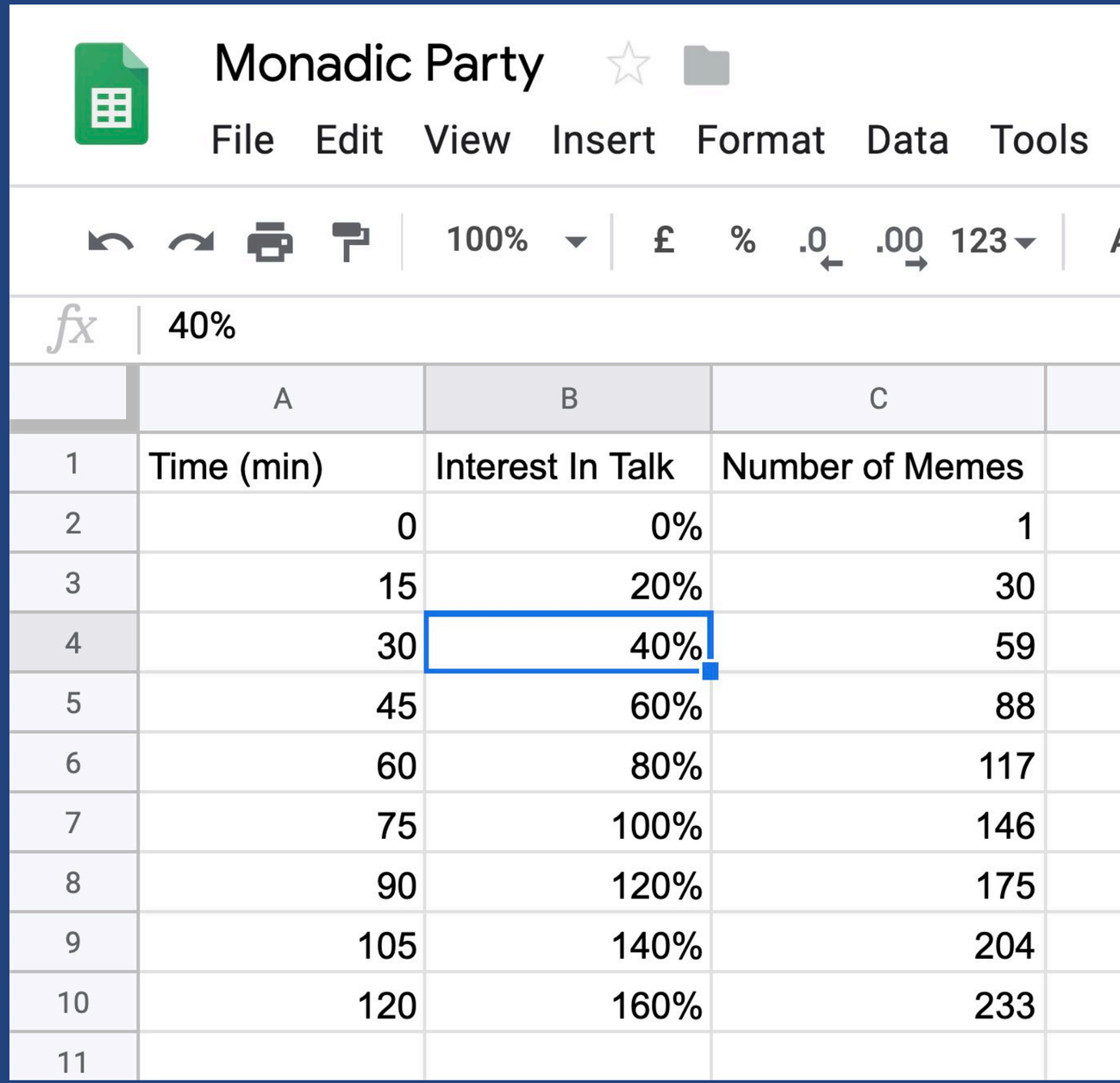


SPREADSHEETS

CELLS ARE THE SLOTS

VIEWS ARE RELATIVE

EXTRACT THE FOCUSED CELL



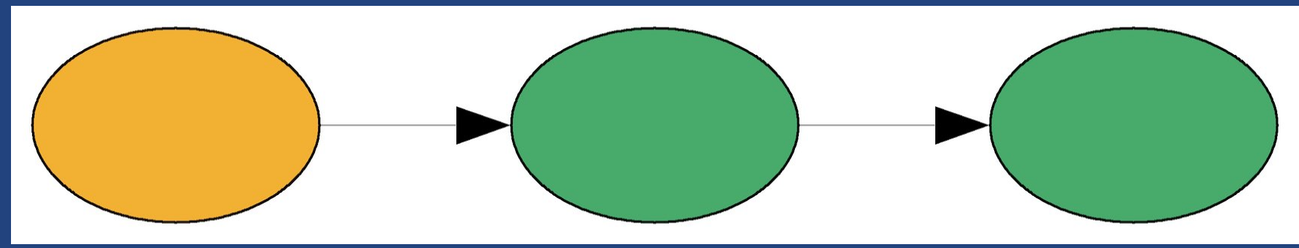
	A	B	C
1	Time (min)	Interest In Talk	Number of Memes
2	0	0%	1
3	15	20%	30
4	30	40%	59
5	45	60%	88
6	60	80%	117
7	75	100%	146
8	90	120%	175
9	105	140%	204
10	120	160%	233
11			

NON-EMPTY LISTS

ELEMENTS ARE THE SLOTS

VIEWS ARE LIST TAILS

EXTRACT THE HEAD OF THE LIST

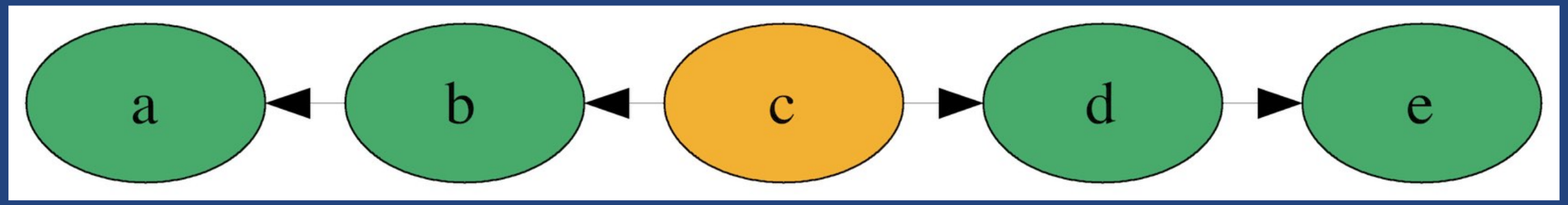


ZIPPERS

ELEMENTS ARE THE SLOTS

VIEWS ARE DOUBLY LINKED LISTS

EXTRACT THE FOCUS OF THE DOUBLY LINKED LIST



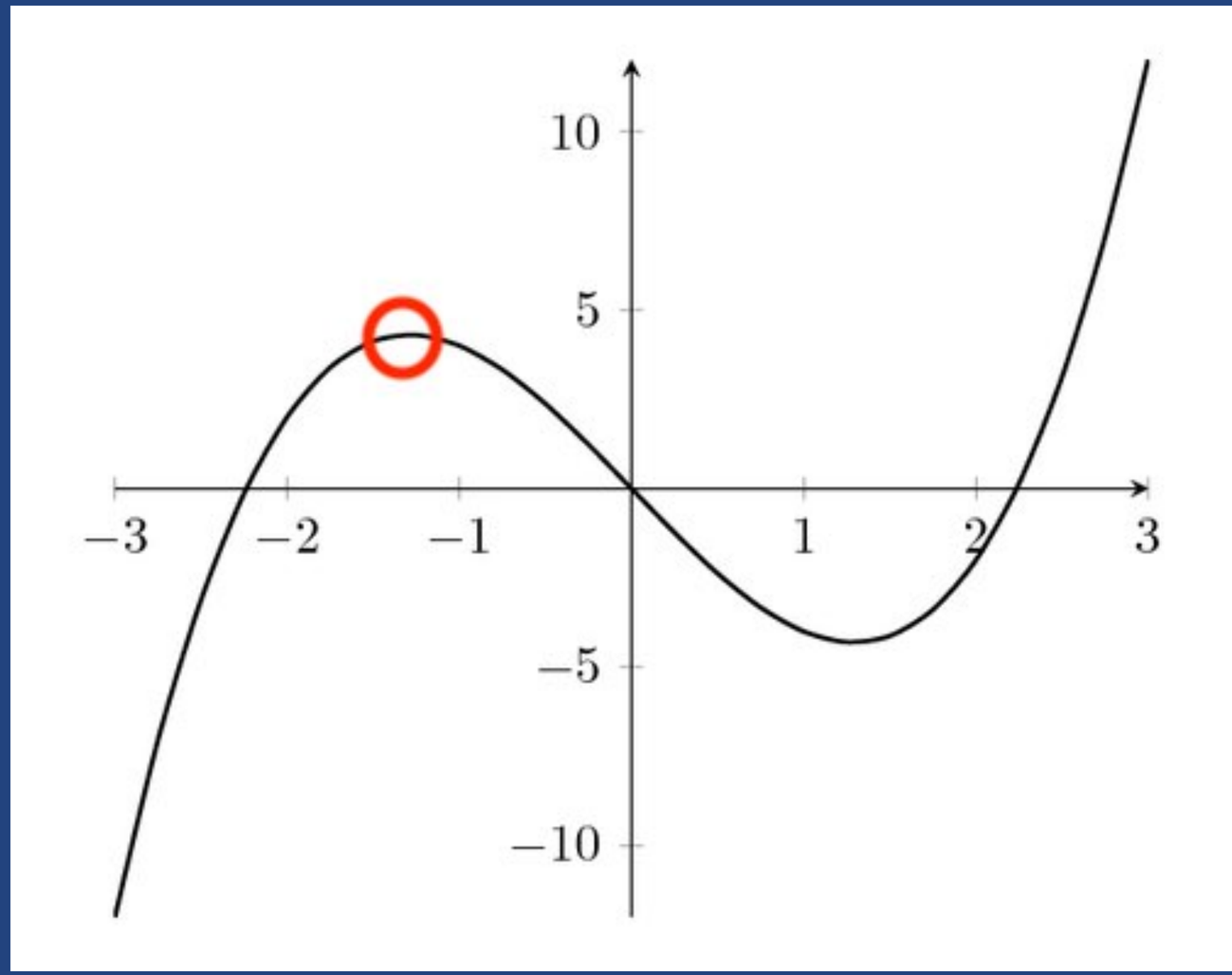
FUNCTIONS

***WHEN PAIRED WITH AN X-VALUE**

X-AXIS POSITIONS ARE THE **SLOTS**

VIEWS ARE RELATIVE TO YOUR **X**

EXTRACT THE **Y** AT YOUR **X**



NOT COMONADS

LISTS

CAN'T ALWAYS EXTRACT

FUNCTIONS

*WHEN **NOT** PAIRED WITH AN **X-VALUE**

WHICH **Y** DO WE **EXTRACT**?

IO

CAN'T GET VALUES OUT

MAYBE / EITHER

CAN'T ALWAYS EXTRACT

QUIZ ME
IS IT A COMONAD??

LET'S LIFT ANOTHER QUERY

```
takeS :: Int -> Stream a -> [a]
```

A RELATIVE **QUERY**

WHICH TAKES THE NEXT N SLOTS FROM THE CURRENT VIEW

```
takeS :: Int -> Stream a -> [a]
```

```haskell
takeS :: Int -> Stream a -> [a]
takeS n input = take n (toList input)
```

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

```
λ> takeS 3 countStream
[0,1,2]
```

EXTEND

LIFTS

QUERIES

INTO

MUTATIONS

EXTEND

RUNS A QUERY OVER EACH SLOT'S VIEW

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

λ> takeS 3 countStream
[0,1,2]

λ> extend (takeS 3) countStream
    [0,1,2]
:> [1,2,3]
:> [2,3,4]
:> [3,4,5]
:> [4,5,6]
:> ...
```

CHALLENGE

COMPUTE A ROLLING AVERAGE OVER A STREAM OF INTEGERS

```
rollingAvg :: Int                -- Window Size
           -> Stream Int         -- Input Stream
           -> Stream Double      -- Stream of averages
```

E.G. rollingAvg 2

```
λ> evens
0 :> 2 :> 4 :> 6 :> ...

λ> rollingAvg 2 evens
(0 + 2) / 2 :> (2 + 4) / 2 :> (4 + 6) / 2 :>  ...
-- reduces to
1 :> 3 :> 5 :> 7 :> ...
```

THE TYPE TELLS US

```
rollingAvg :: Int                -- Window Size
           -> Stream Int         -- Input Stream
           -> Stream Double      -- Stream of averages
```

IT'S A MUTATION

BUT

MUTATIONS ARE HARD

WHY AVERAGE EVERY SLOT WHEN WE CAN JUST LIFT A QUERY?

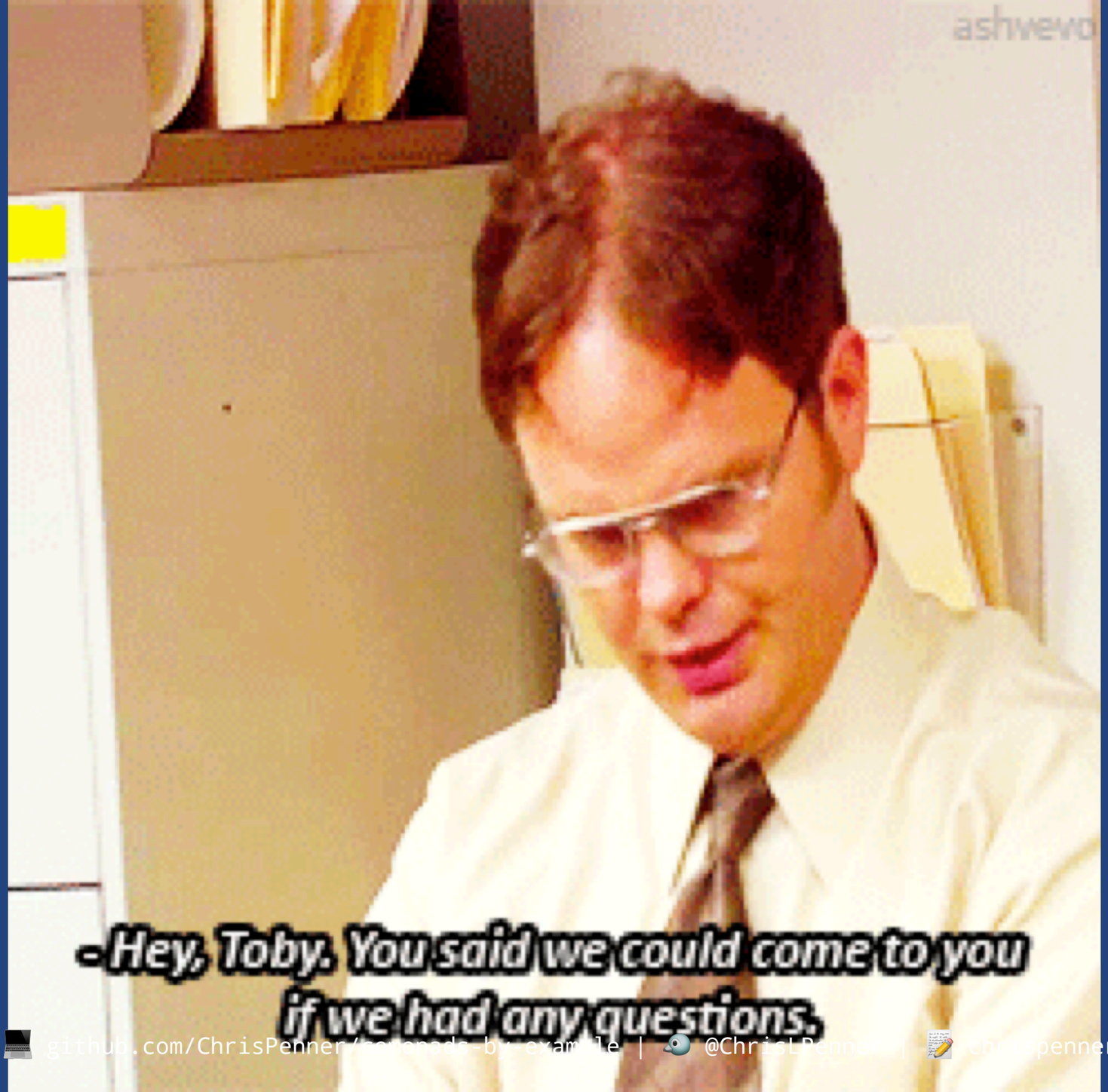
QUERY VERSION

```haskell
windowedAvg :: Int -> Stream Int -> Double
windowedAvg windowSize = avg . takeS windowSize
  where
    avg :: [Int] -> Double
    avg xs =
        fromIntegral (sum xs)
        / fromIntegral (length xs)
```

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

rollingAvg  :: Int -> Stream Int -> Stream Double
```

A ROLLING AVERAGE IS JUST A WINDOWED AVERAGE AT EVERY VIEW

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

λ> extend (windowedAvg 2) countStream
0.5 :> 1.5 :> 2.5 :> 3.5 :> 4.5 :> ...

λ> extend (windowedAvg 3) countStream
1.0 :> 2.0 :> 3.0 :> 4.0 :> 5.0 :> ...
```


ashwewo
- Hey, Toby. You said we could come to you
if we had any questions.

CHALLENGE: IMPLEMENT COMONAD FOR STREAM

```
data Stream a = a :> Stream a
        deriving (Functor, Foldable)

instance Comonad Stream where
  extract :: Stream a -> a

  duplicate :: Stream a -> Stream (Stream a)

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

```haskell
instance Comonad Stream where
  extract :: Stream a -> a
  extract (a :> _) = a

  duplicate :: Stream a -> Stream (Stream a)
  duplicate s@(_ :> rest) = s :> duplicate rest

  extend :: (Stream a -> b) -> Stream a -> Stream b
  extend f s@(_ :> rest) = f s :> extend f rest
  -- OR
  extend f s = f <$> duplicate s
```