TRACED

A.K.A. CO-WRITER

a.k.a. Monoid m => m -> a

```
newtype Traced m a = Traced (m -> a)
```

A FUNCTION WHERE THE ARGUMENT IS A MONOID

TRACED IS A QUERY BY ITSELF

```
sum :: [ Int ] -> Int
```

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

```
maxWordCount :: Map String Int -> Int
```

```
weightOf :: Set Item -> Int
```

TRACED IS LIKE STORE BUT ALL POSITIONS ARE RELATIVE

I.E.

WE DON'T HAVE ACCESS TO THE CURRENT POSITION

m -> a

IS ONLY A COMONAD WHEN M IS A MONOID

TRACED QUERIES

```haskell
extract :: Traced m a -> a
extract (Traced f) = f mempty
```

```
λ> :t sum
sum :: [Int] -> Int

λ> let adder = traced sum
adder :: Traced [Int] Int

λ> extract adder -- a.k.a. sum []
0
```

TRACE

APPLIES THE FUNCTION TO AN ARGUMENT

TRACE

```
trace :: m -> Traced m a -> a
```

```
λ> trace [1,2,3] adder -- a.k.a. sum [1,2,3]
6
```

EXTENDING TRACE BAKES IN ARGUMENTS

```
λ> let adder' = adder =>> trace [1,2,3]
adder' :: Traced [Int] Int

λ> extract adder' -- a.k.a. sum (mempty <> [1,2,3])
6

λ> trace [10] adder' -- a.k.a. sum ([10] <> [1,2,3])
16
```

EXTENDING

CREATES A NEW TRACED FUNCTION

WITH ARGUMENTS INSIDE

```
duplicate :: Traced m a -> Traced m (Traced m a)
duplicate (Traced f) =
    Traced $ \m -> Traced (f . mappend m)
```

EVEN IF WE APPLY THE INNER FUNCTION
IT REMEMBERS ITS ARGUMENT.

EACH TIME WE EXTEND TRACE WE PREPEND TO THE ARGUMENT

```haskell
newBuilder :: Traced [String] String
newBuilder = traced concat

logMsg :: Traced [String] String -> String
logMsg = trace ["hello "] =>= trace ["world"]

-- Uh Oh!
-- λ> logMsg newBuilder
-- "worldhello "
```

TRACE ALWAYS PREPENDS NEW ELEMENTS

QUESTIONS?

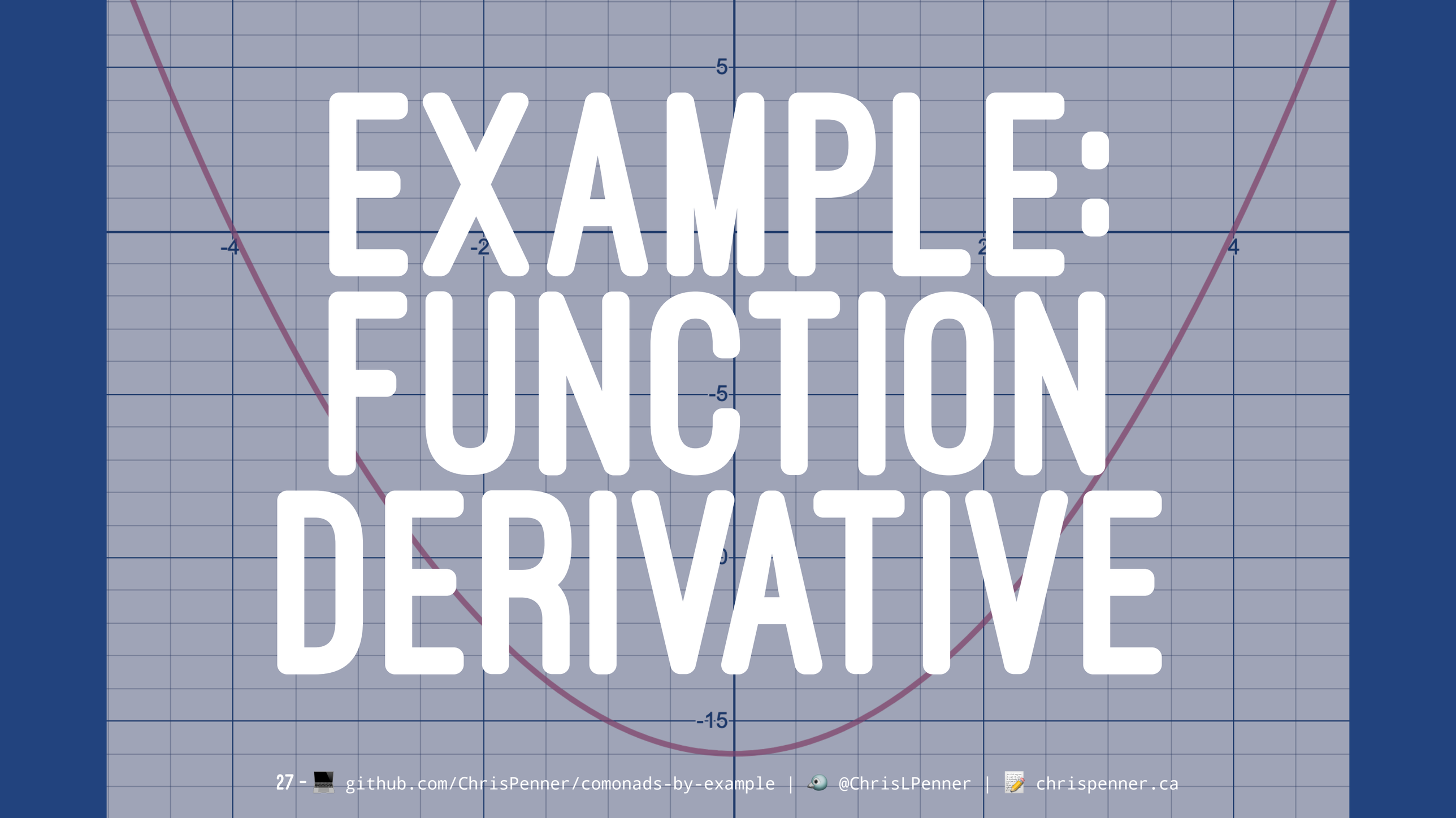
TRACED

YOUR POSITION IS IMPLICIT

TRACE MOVES RELATIVELY

YOU CAN LOOK NEARBY

EXTENDING TRACE MOVES YOU



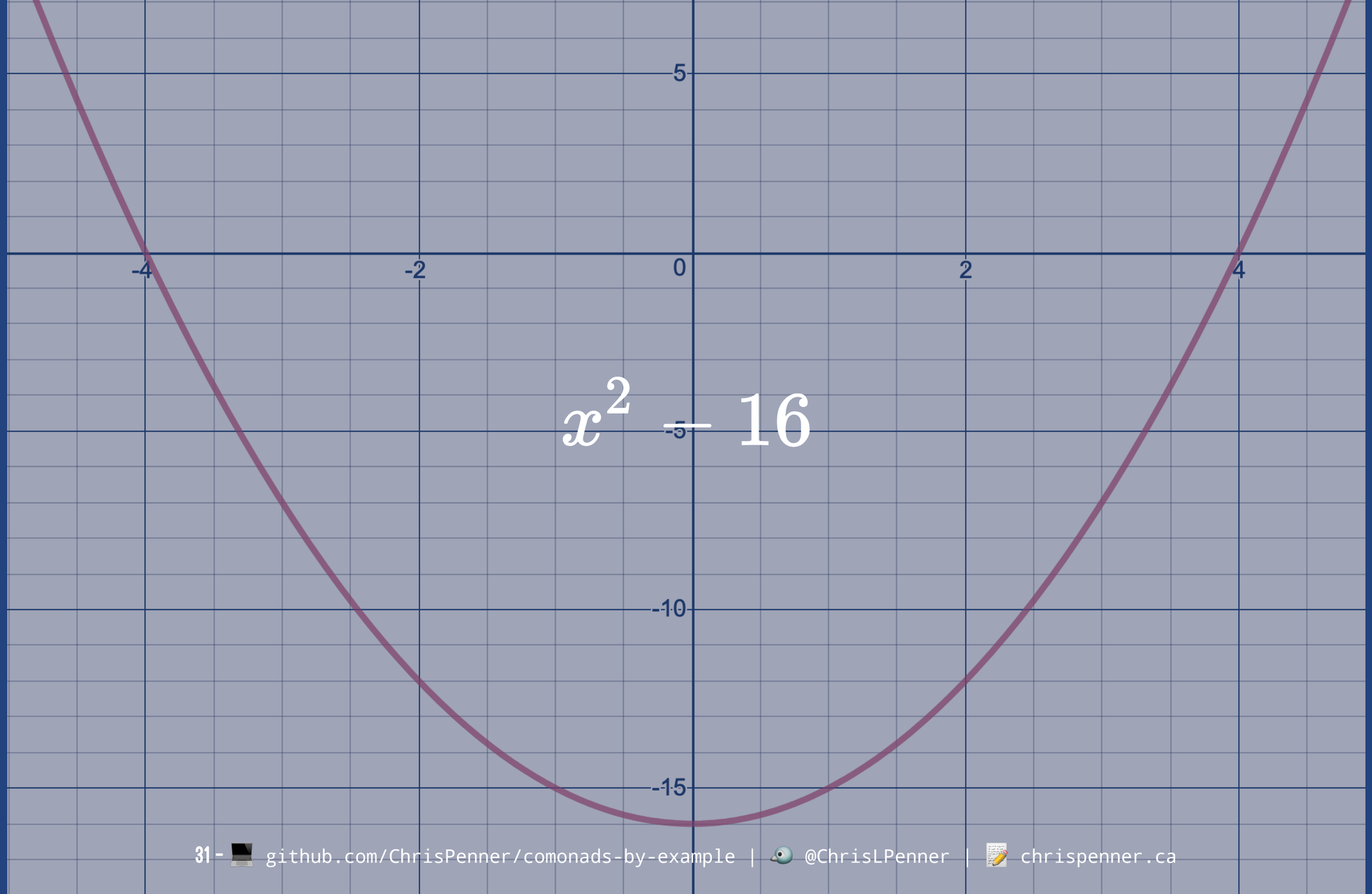
EXAMPLE: FUNCTION DERIVATIVE

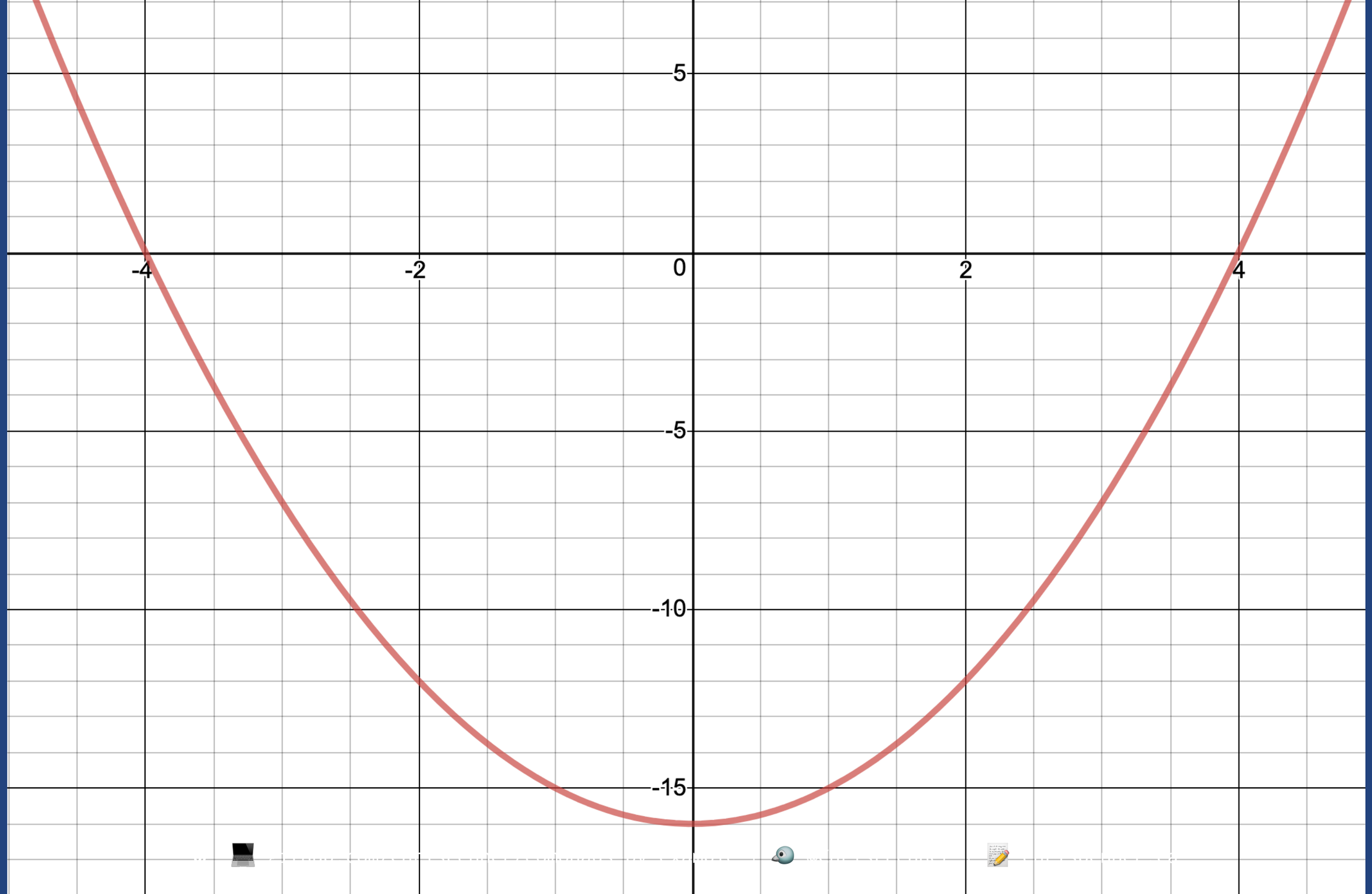
A DERIVATIVE IS A CONTEXTUAL CALCULATION

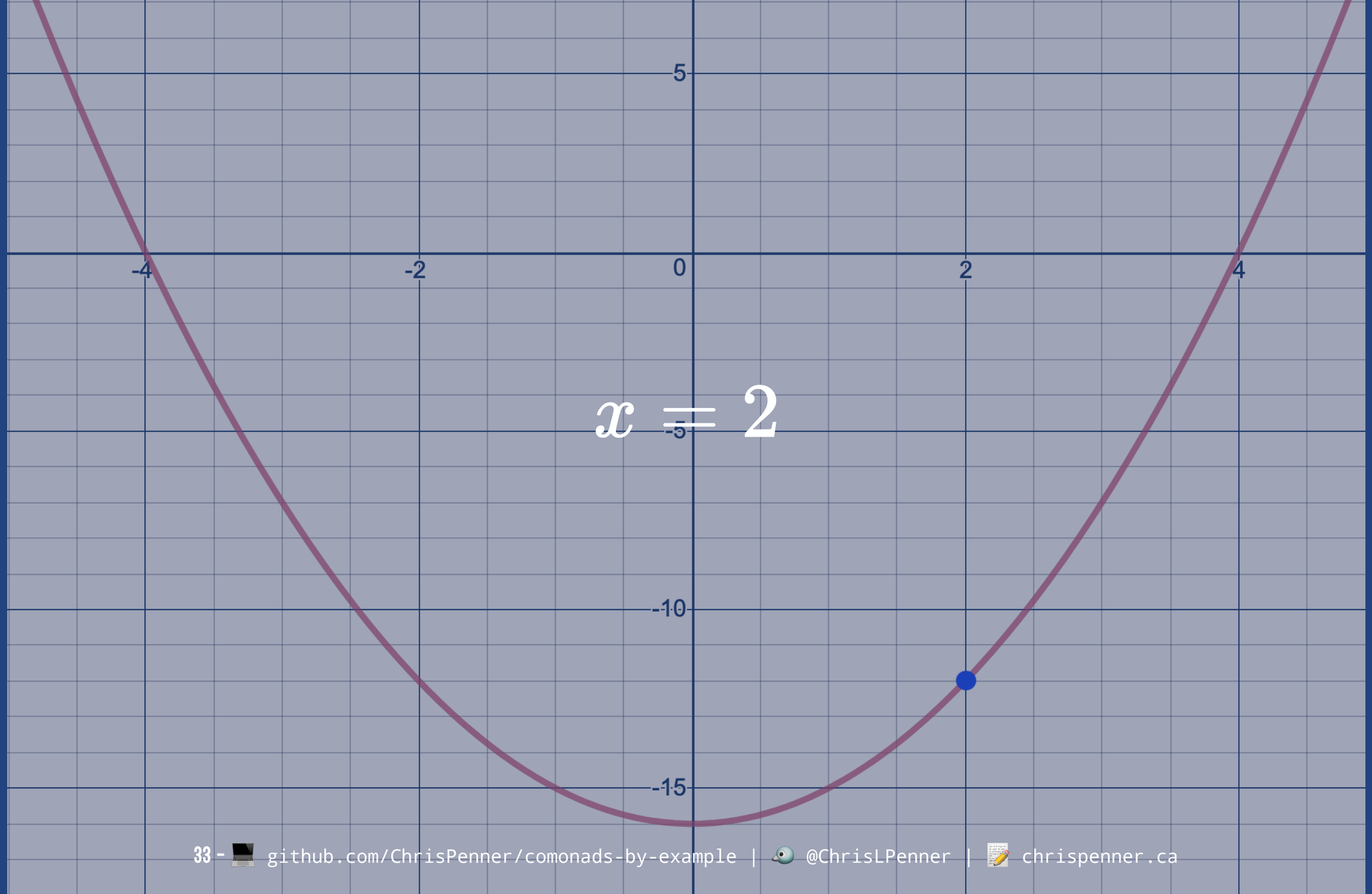
THE DERIVATIVE AT A GIVEN POINT IS DEPENDANT ON THE NEARBY VALUES

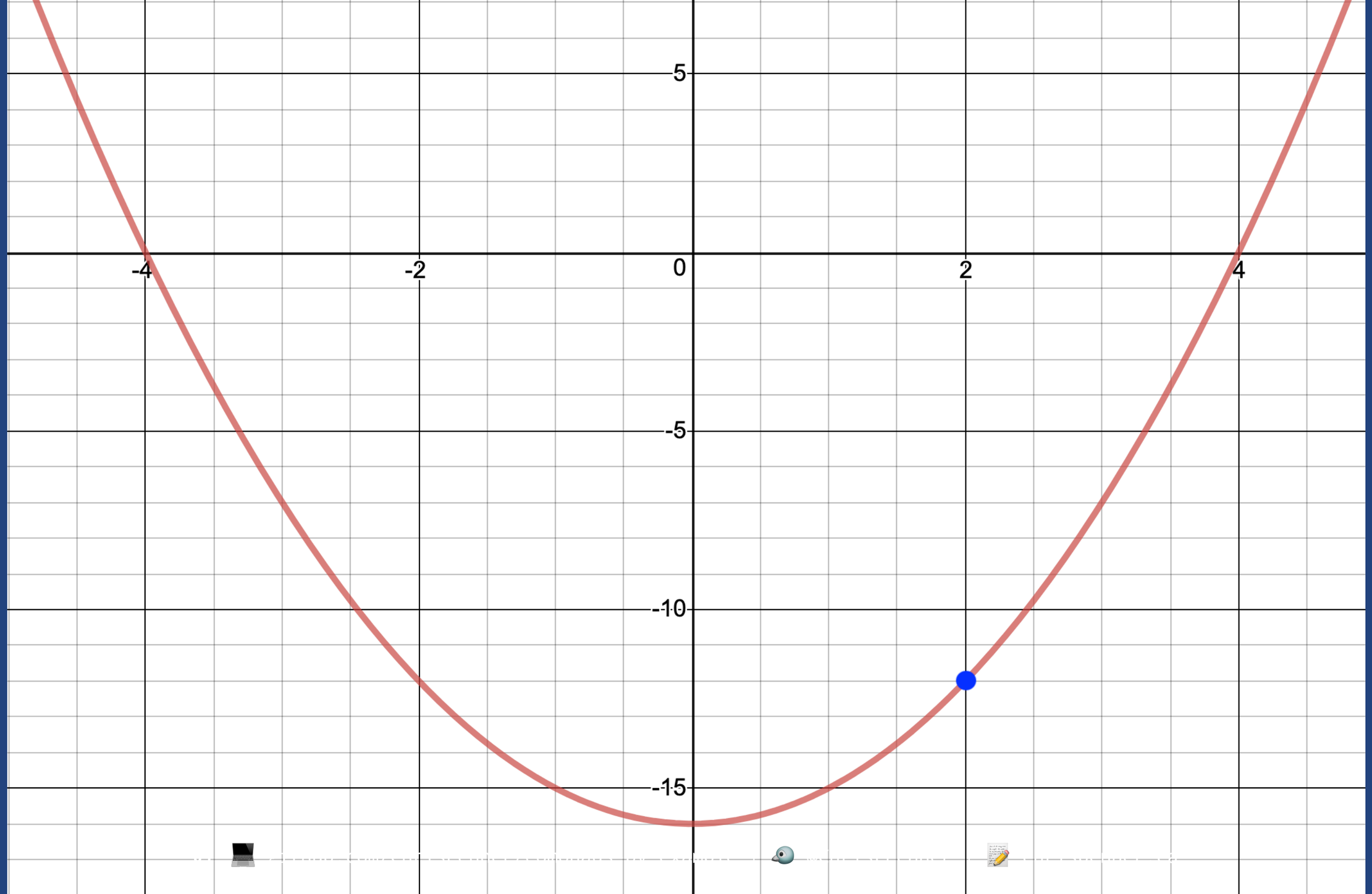
WE CAN ROUGHLY ESTIMATE

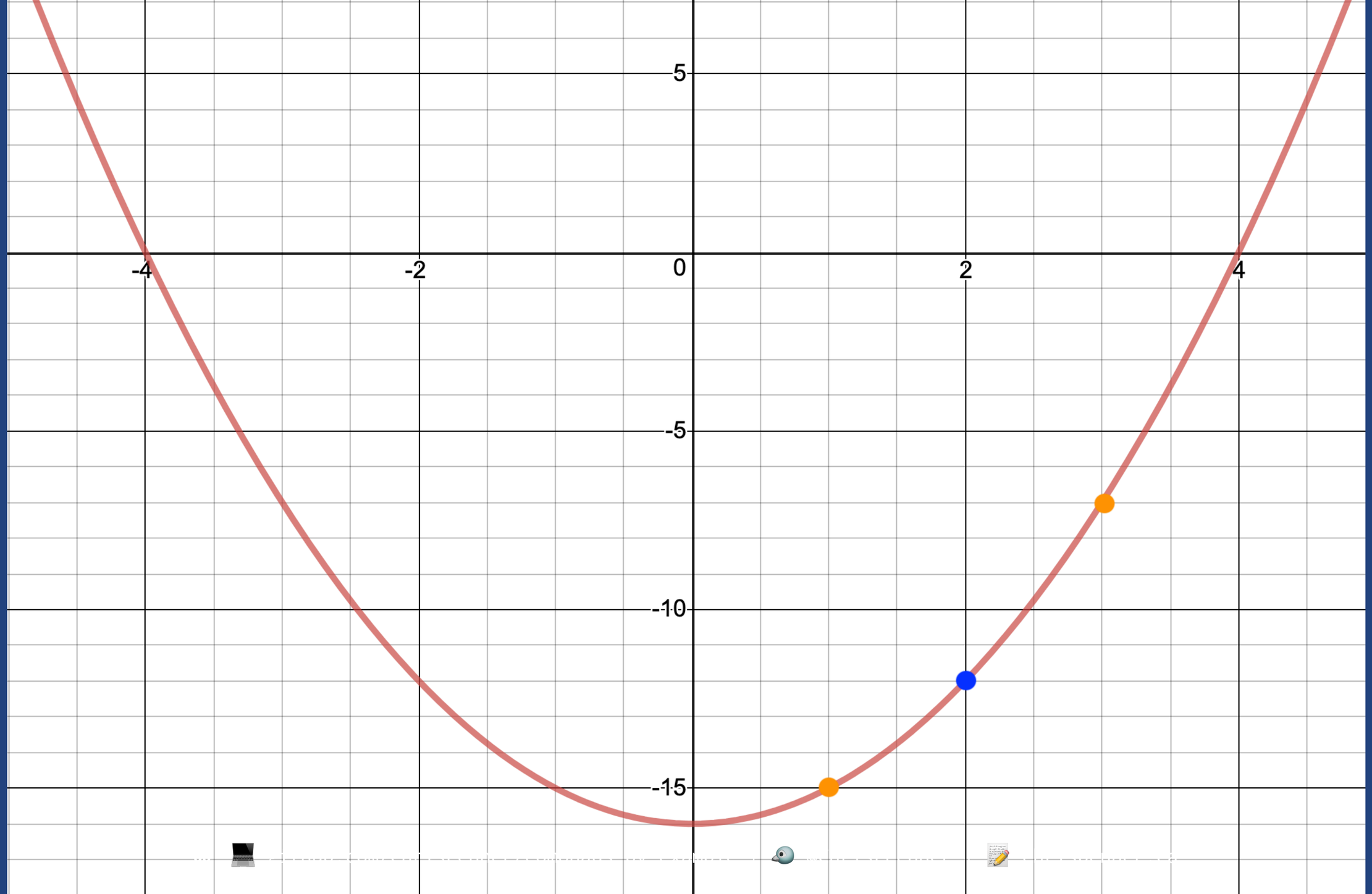
$$f'(x) = \frac{f(x+1) - f(x-1)}{2}$$

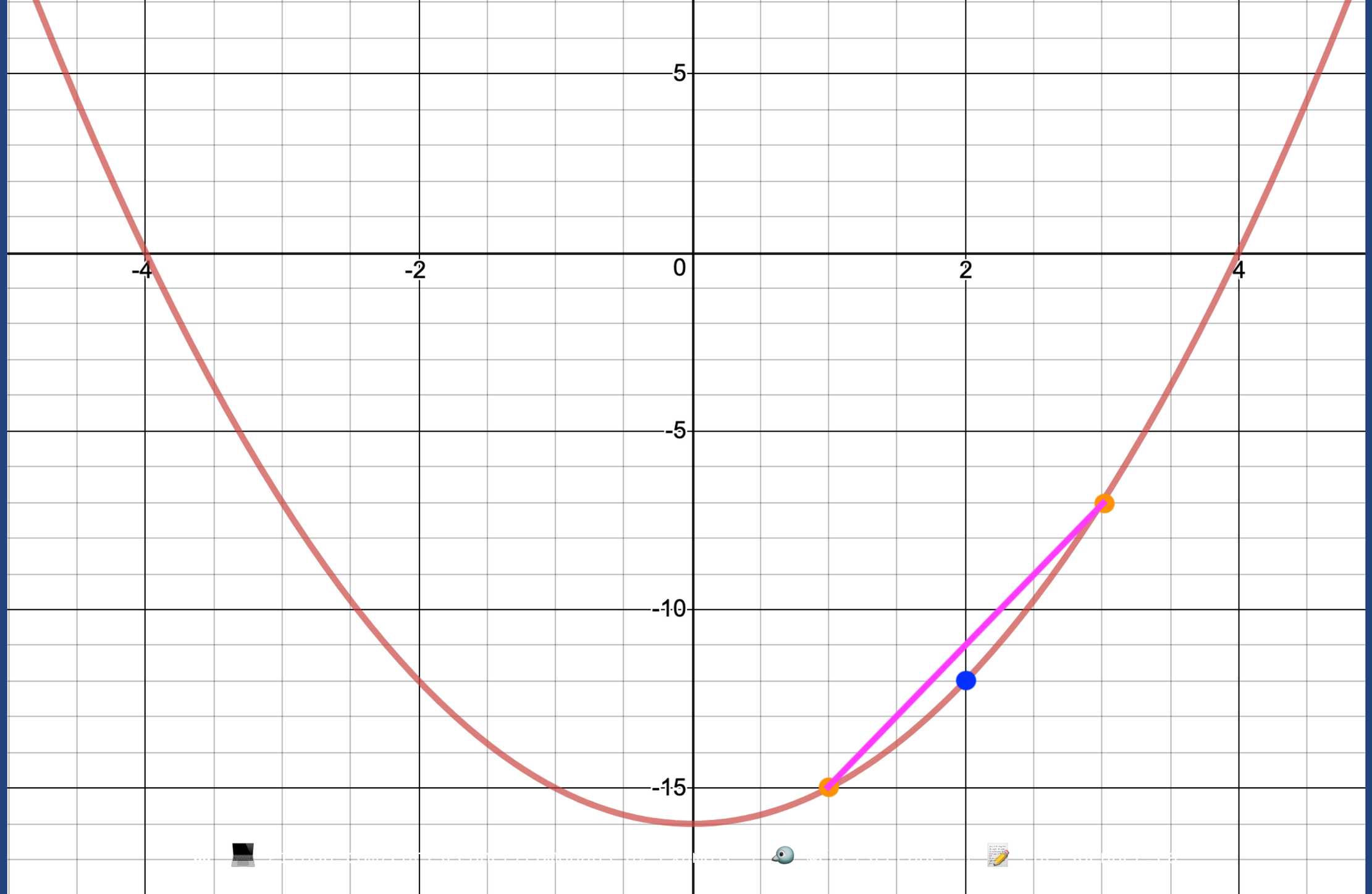

5
-4
-2
0
2
4
-5
-10
-15
$x^2 - 16$

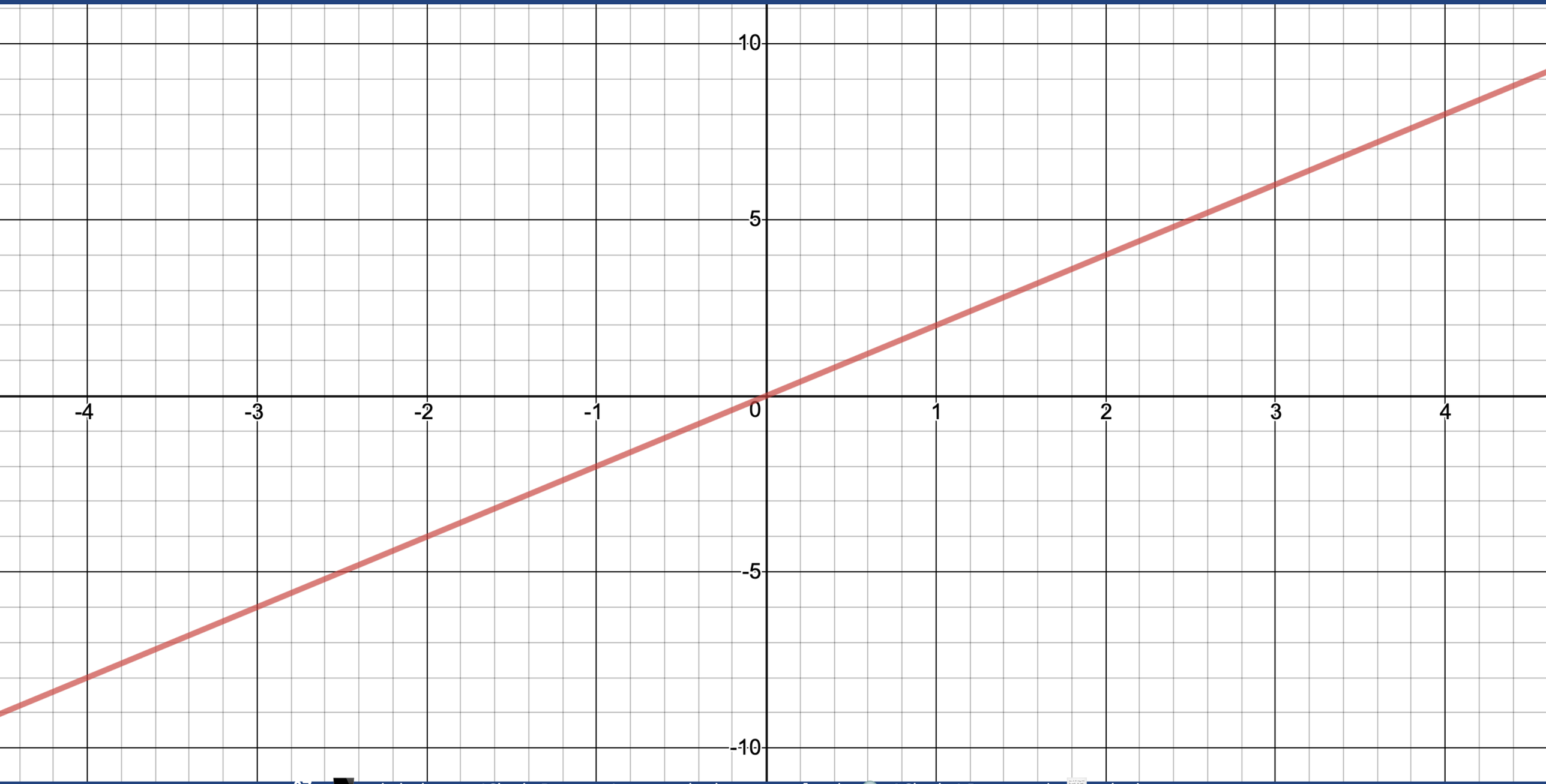

5
-4
-2
0
2
4
-5
-10
-15


5
0
-4
-2
2
4
-5
-10
-15
$x = 2$


5
-4
-2
0
2
4
-5
-10
-15


5
-4
-2
0
2
4
-5
-10
-15


5
-4
-2
0
2
4
-5
-10
-15


10
5
-5
-10
-4
-3
-2
-1
0
1
2
3
4

```haskell
-- Solution for the sqrt of 16
func :: Sum Double -> Double
func (Sum x) = (x ^ (2 :: Integer)) - 16

f :: Traced (Sum Double) Double
f  = traced func
```

```
λ> extract f -- f (Sum 0)
-16.0
λ> trace (Sum 2) f
-12.0
λ> trace (Sum 6) f
20.0
λ> trace (Sum 4) f
0.0
```

WE CAN WRITE A QUERY TO ESTIMATE f' AT A SINGLE POSITION

```
estimateDerivative
:: Traced (Sum Double) Double -> Double
```

```haskell
estimateDerivative :: Traced (Sum Double) Double
                   -> Double
estimateDerivative w =
    let leftY = trace (Sum (-1)) w
        rightY = trace (Sum 1) w
     in (rightY - leftY) / 2
```

MORE DO-NOTATION

ABUSE

DO-NOTATION

```haskell
estimateDerivative w =
    let leftY = trace (Sum (-1)) w
        rightY = trace (Sum 1) w
     in (rightY - leftY) / 2

estimateDerivativeReader :: Traced (Sum Double) Double -> Double
estimateDerivativeReader = do
    leftY <- trace (Sum (-1))
    rightY <- trace (Sum 1)
    return $ (rightY - leftY) / 2
```

IF WE ESTIMATE f'
AT ALL POSITIONS
WE ESTIMATE THE WHOLE FUNCTION!

```haskell
derive :: Traced (Sum Double) Double
                    -> Traced (Sum Double) Double
derive = extend estimateDerivative
```

COMONADS HAVE `liftW2`

LIKE `liftA2`

BUT ALWAYS ZIPPY

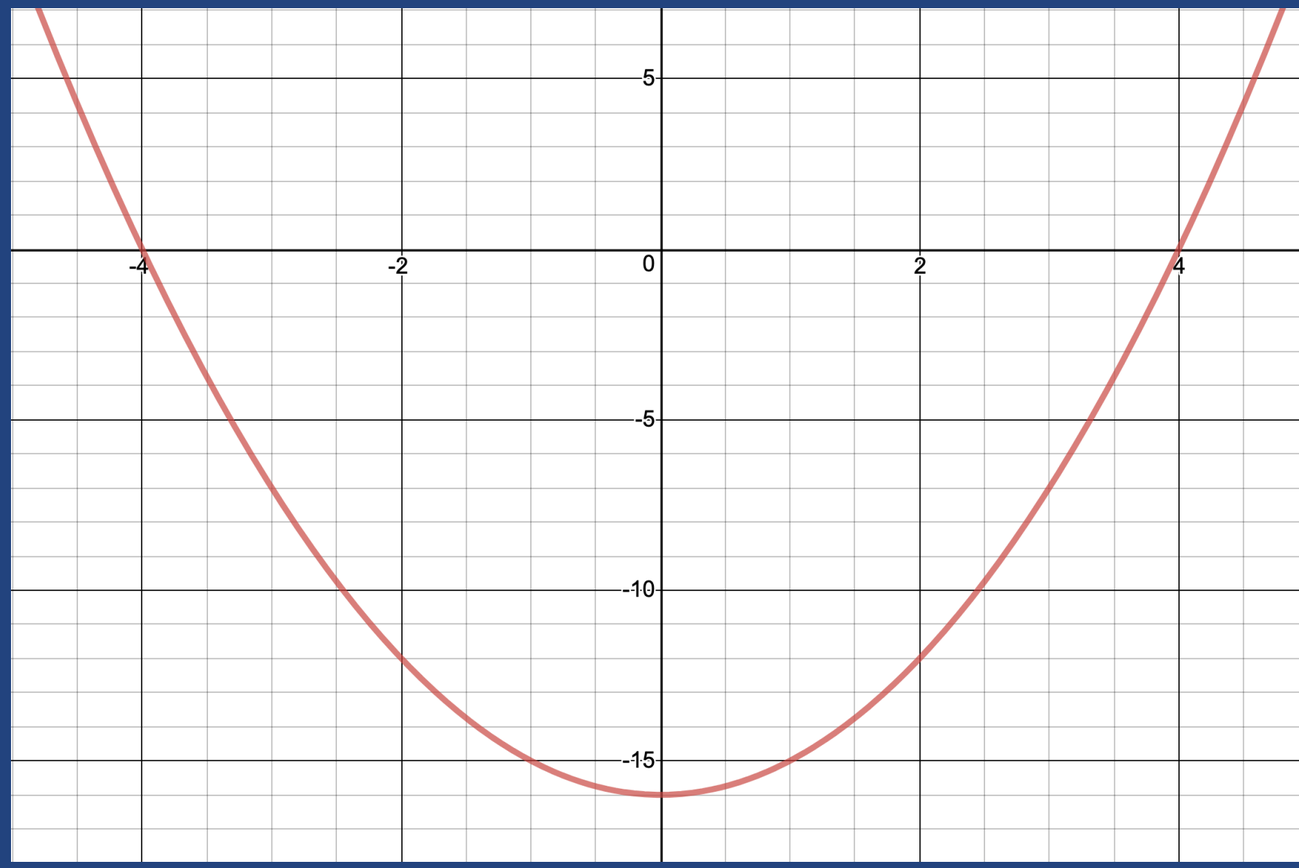
```
liftW2 :: (a -> b -> c) -> w a -> w b -> w c
```

IF A COMONAD IS ALSO APPLICATIVE

```
liftW2 == liftA2
```

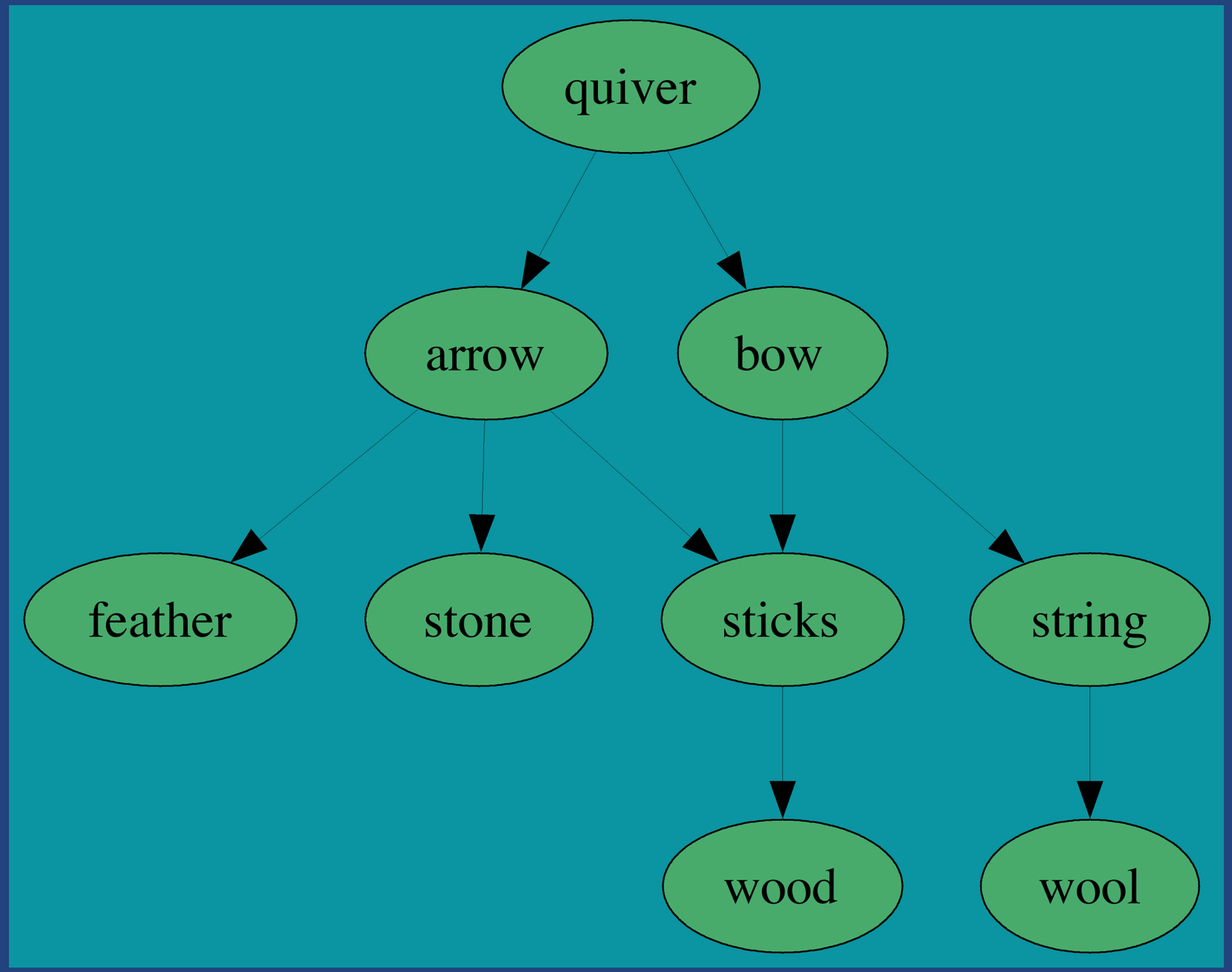
```haskell
withDerivative :: Traced (Sum Double) (Double, Double)
withDerivative = liftW2 (,) f (derive f)
```

```
λ> extract withDerivative
(-16.0, 0.0)

λ> trace (Sum 0) withDerivative
(-16.0, 0.0)

λ> trace (Sum 1) withDerivative
(-15.0, 2.0)

λ> trace (Sum 4) withDerivative
(0.0, 8.0)
```



QUESTIONS?

DEPENDENCY TRACKING

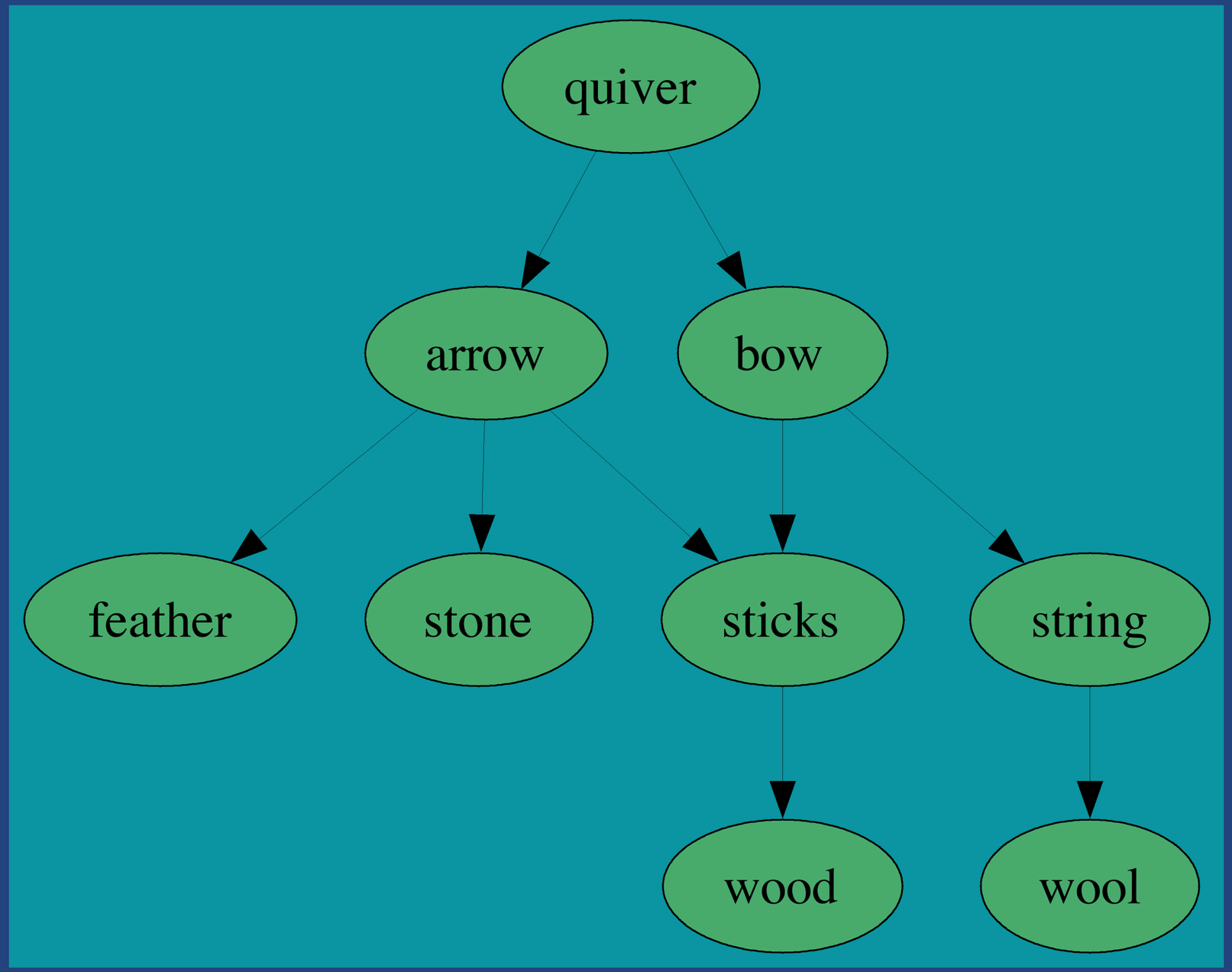

quiver
arrow
bow
feather
stone
sticks
string
wood
wool

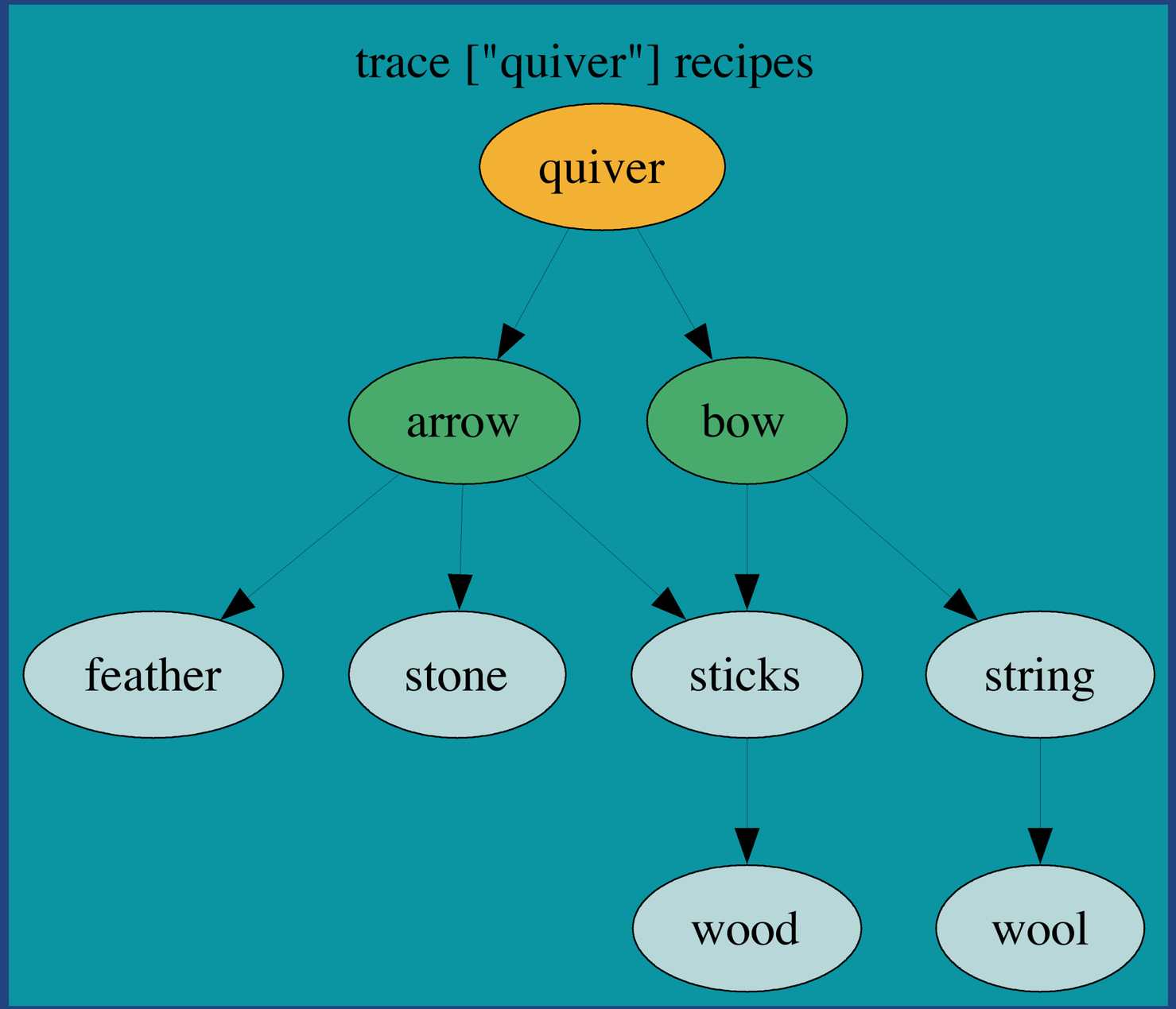
```haskell
ingredientsOf :: String -> S.Set String
ingredientsOf "string"  = S.fromList ["wool"]
ingredientsOf "sticks"  = S.fromList ["wood"]
ingredientsOf "bow"     = S.fromList ["sticks", "string"]
ingredientsOf "arrow"   = S.fromList ["sticks", "feather", "stone"]
ingredientsOf "quiver"  = S.fromList ["arrow", "bow"]
ingredientsOf "torches" = S.fromList ["coal", "sticks"]
-- Everything else has no dependencies
ingredientsOf _         = mempty

recipes :: Traced (S.Set String) (S.Set String)
recipes = traced (foldMap ingredientsOf)
```

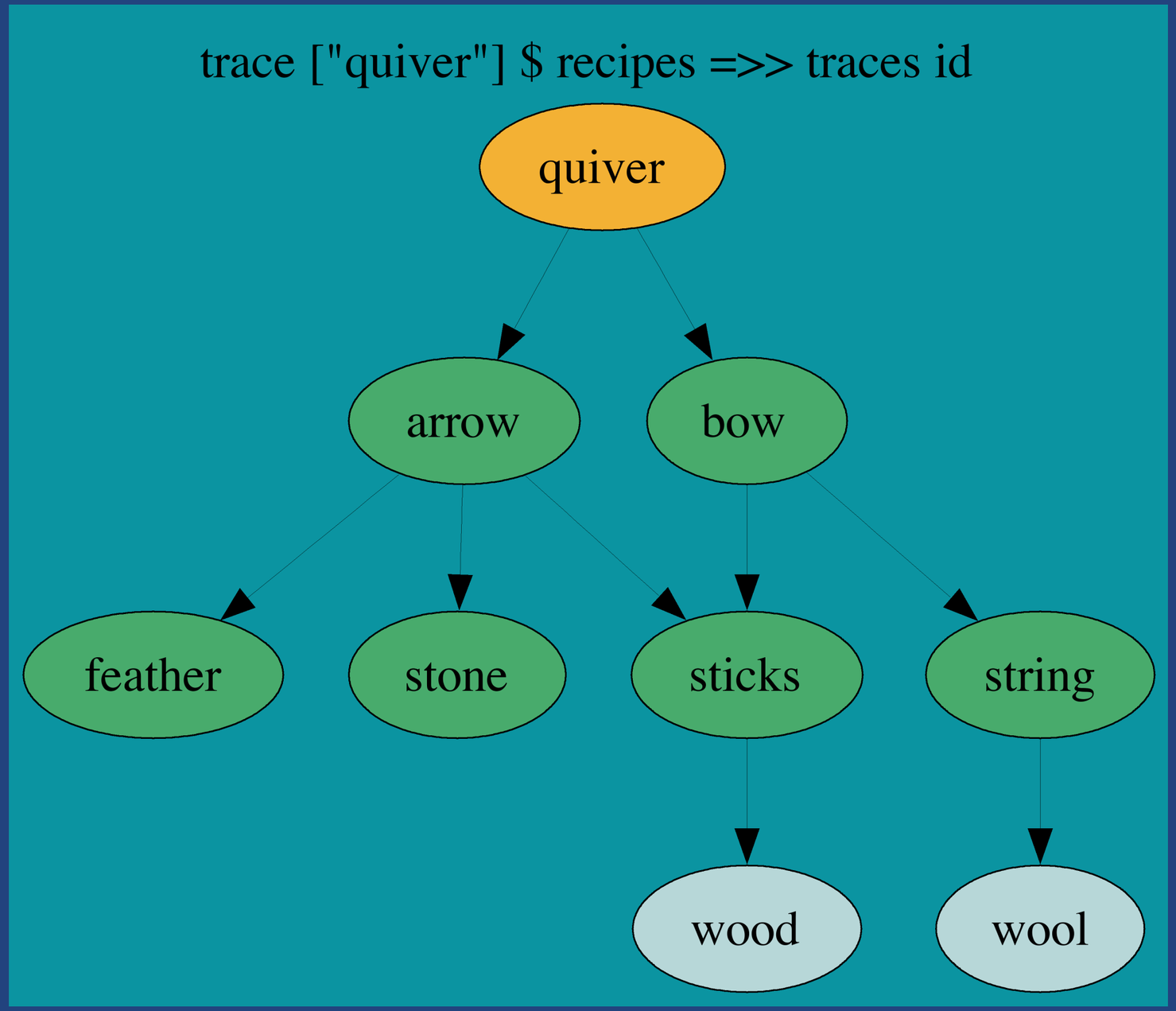
```
quiver   -> arrow, bow
arrow    -> sticks, feather, stone
bow      -> sticks, string
torches -> coal, sticks
sticks   -> wood
string   -> wool

λ> trace ["string"] recipes
fromList ["wool"]

λ> trace ["string", "torches"] recipes
fromList ["coal","sticks","wool"]

λ> recipes & trace ["torches"]
fromList ["coal","sticks"]

λ> recipes & (trace ["string", "torches"] =>= trace ["sticks"])
fromList ["coal","sticks","wood","wool"]
```

```
quiver  -> arrow, bow
arrow   -> sticks, feather, stone
bow     -> sticks, string
torches -> coal, sticks
sticks  -> wood
string  -> wool
```

```
λ> trace ["quiver"] $ recipes
fromList ["arrows","bow"]

λ> trace ["quiver"] $ recipes =>> traces id
fromList ["arrows","bow","feathers","sticks","stone","string"]

λ> trace ["quiver"] $ recipes =>> traces id =>> traces id
fromList ["arrows","bow","feathers","sticks","stone","string","wood","wool"]
```


quiver
arrow
bow
feather
stone
sticks
string
wood
wool


trace ["quiver"] recipes
quiver
arrow
bow
feather
stone
sticks
string
wood
wool


trace ["quiver"] $ recipes =>> traces id
quiver
arrow
bow
feather
stone
sticks
string
wood
wool

QUESTIONS?

chrispenner.ca

github.com/ChrisPenner

 @ChrisLPenner