COMONAD FIX!

FOR WHEN YOUR COMONADS ARE BROKEN

GIVEN THE ANSWER
COMPUTE THE ANSWER

SERIOUSLY HASKELL WTF?

LIKE SPREADSHEETS

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

GIVEN A COMONAD FILLED WITH QUERIES OVER THE FINAL RESULT

RESOLVE THEM ALL USING THE RESULT TO GET THE RESULT

FACTORIAL

```
> factorial 3
3 * 2 * 1
6
```

FACTORIAL

```
-- The FUNCTION is the context
-- which we can use in the computation
let fact 0 = 1
    fact n = n * fact (n -1)
                  ^^^^

> fact 3
6
```

NOTE HOW `fact 0 = 1`
DOESN'T REFERENCE THE CONTEXT

A.K.A. IT DOESN'T RECURSE

EACH LAYER NEEDS ACCESS TO A NEARBY ANSWER

FACTORIAL

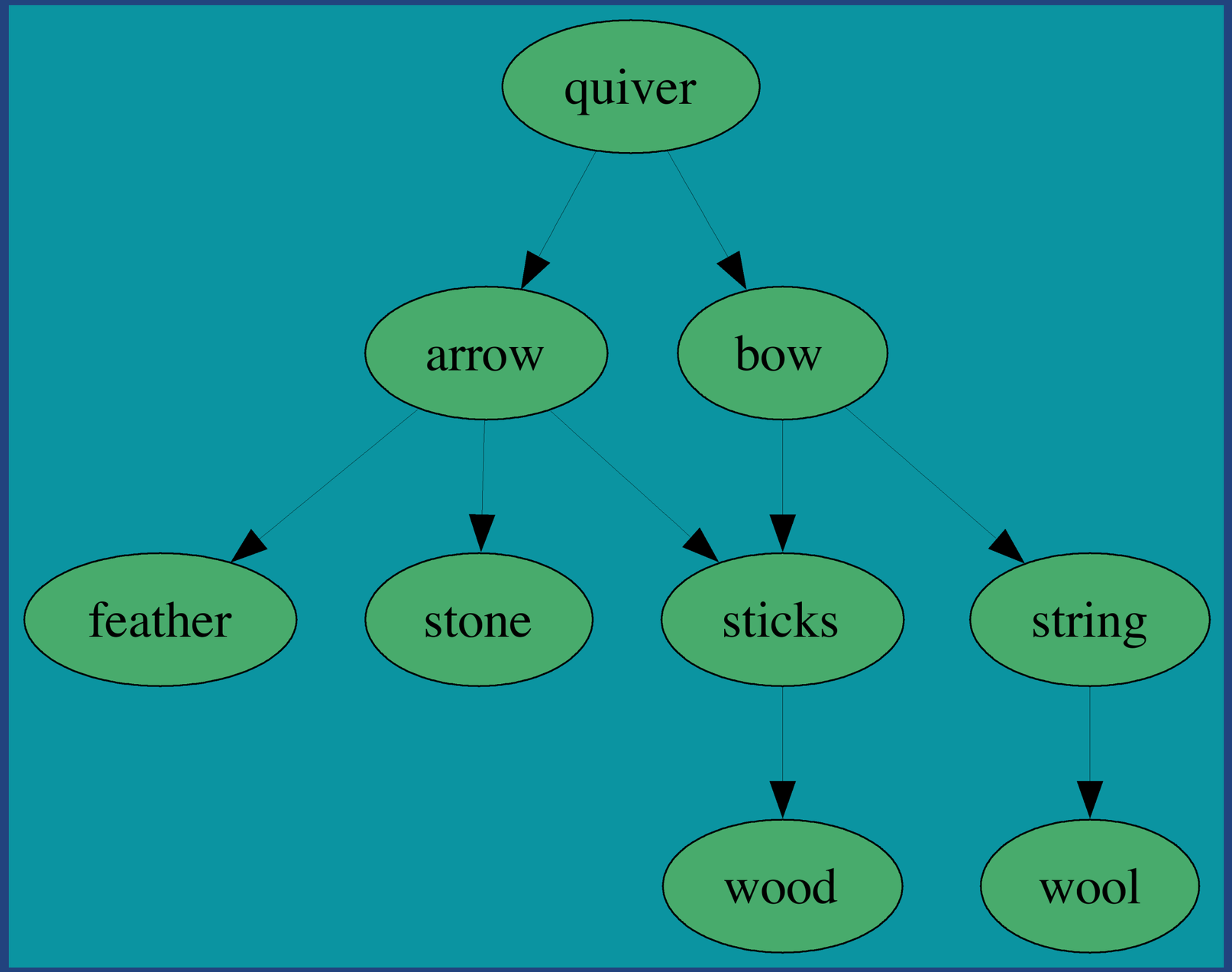
```haskell
factorialTraced :: Traced (Sum Int) Int
factorialTraced = extend wfix (traced go)
  where
    go :: Sum Int -> Traced (Sum Int) Int -> Int
    go (Sum 0) _ = 1
    go (Sum n) t = n * trace (-1) t
```

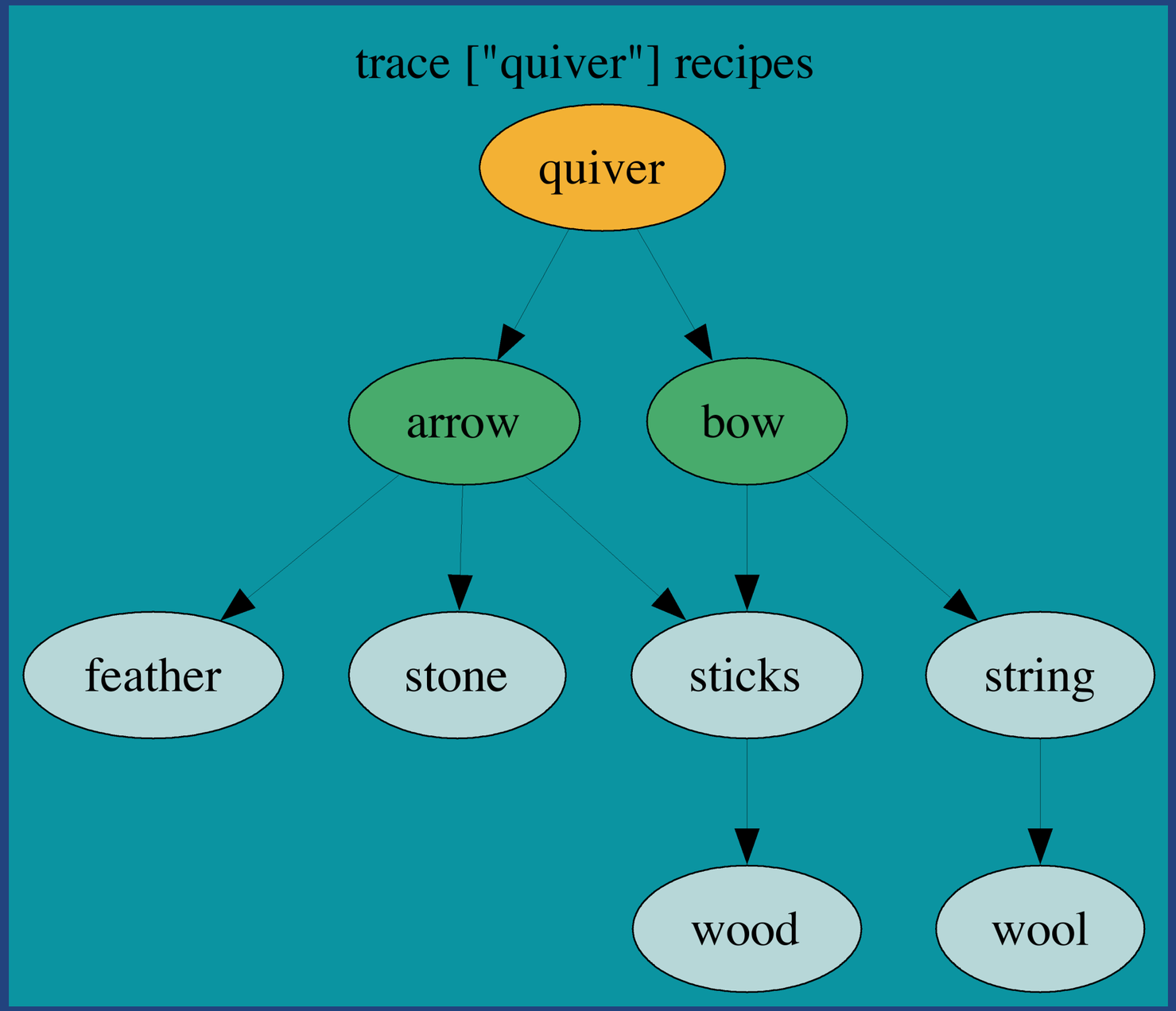
EXAMPLE: DEPENDENCY TRACKING

```
ingredientsOf :: String -> S.Set String
ingredientsOf "string" = S.fromList ["wool"]
ingredientsOf "sticks"  = S.fromList ["wood"]
ingredientsOf "bow"     = S.fromList ["sticks", "string"]
ingredientsOf "arrows"  = S.fromList ["sticks", "feathers", "stone"]
ingredientsOf "quiver" = S.fromList ["arrows", "bow"]
ingredientsOf "torches"  = S.fromList ["coal", "sticks"]
ingredientsOf _         = mempty

recipes :: Store (S.Set String) (S.Set String)
recipes = store (foldMap ingredientsOf) mempty
```

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

λ> peek ["string"] recipes
fromList ["wool"]
λ> peek ["string", "torches"] recipes
fromList ["coal","sticks","wool"]
λ> extract $ recipes =>> peek ["torches"]
fromList ["coal","sticks"]
```

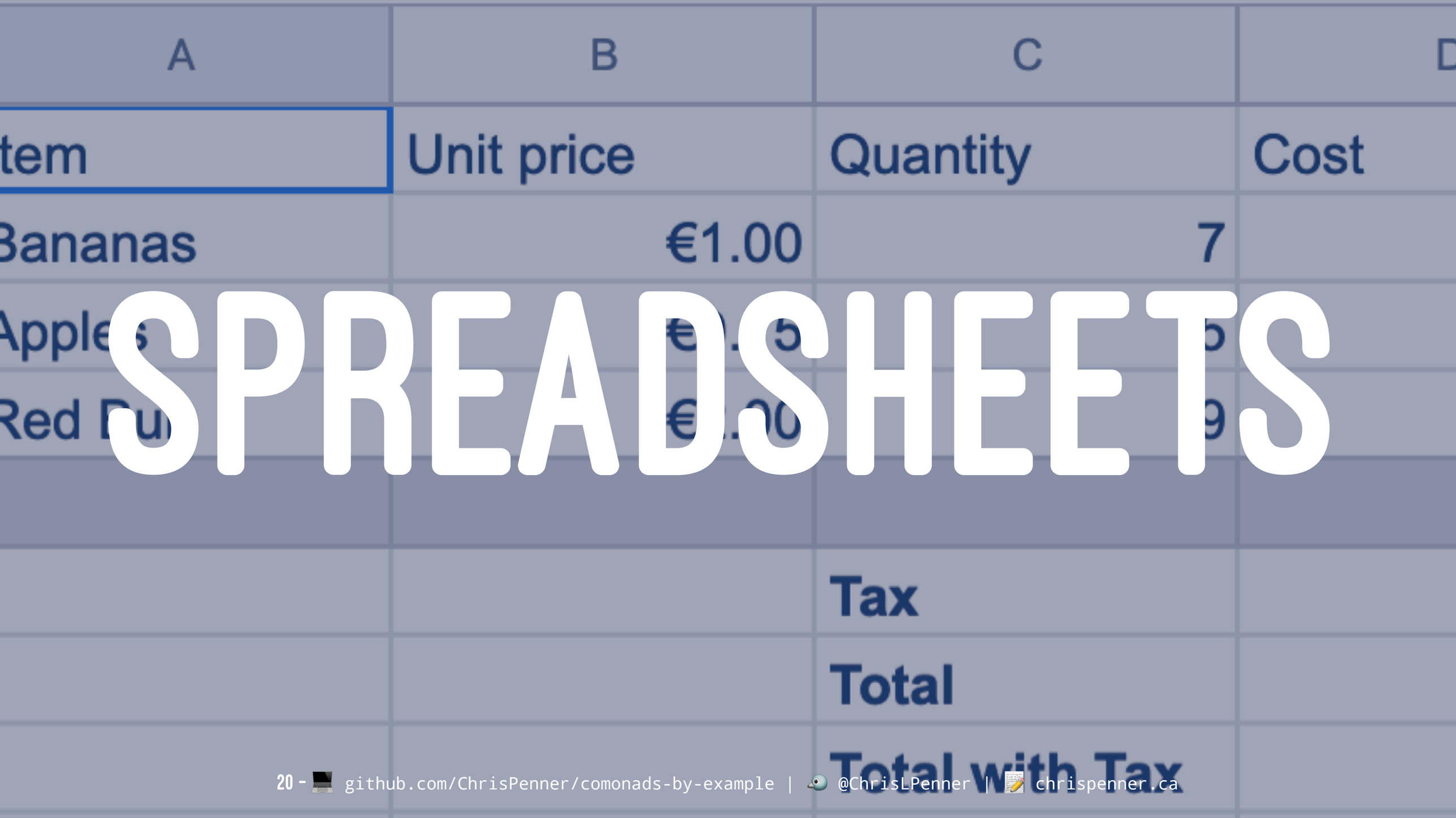
```
string  -> wool
sticks  -> wood
bow     -> sticks, string
arrow   -> sticks, feather, stone
quiver  -> arrow, bow
torches -> coal, sticks
```


quiver
arrow
bow
feather
stone
sticks
string
wood
wool


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

```haskell
recipes :: Store (S.Set String) (S.Set String)
recipes = store (foldMap ingredientsOf) mempty

allDeps :: Store (S.Set String) (S.Set String)
allDeps = extend wfix (go <$> recipes)
  where
    go :: S.Set String -> Store (S.Set String) (S.Set String) -> (S.Set String)
    go deps _    | S.null deps = mempty
    go deps result             = deps <> peek deps rec
```

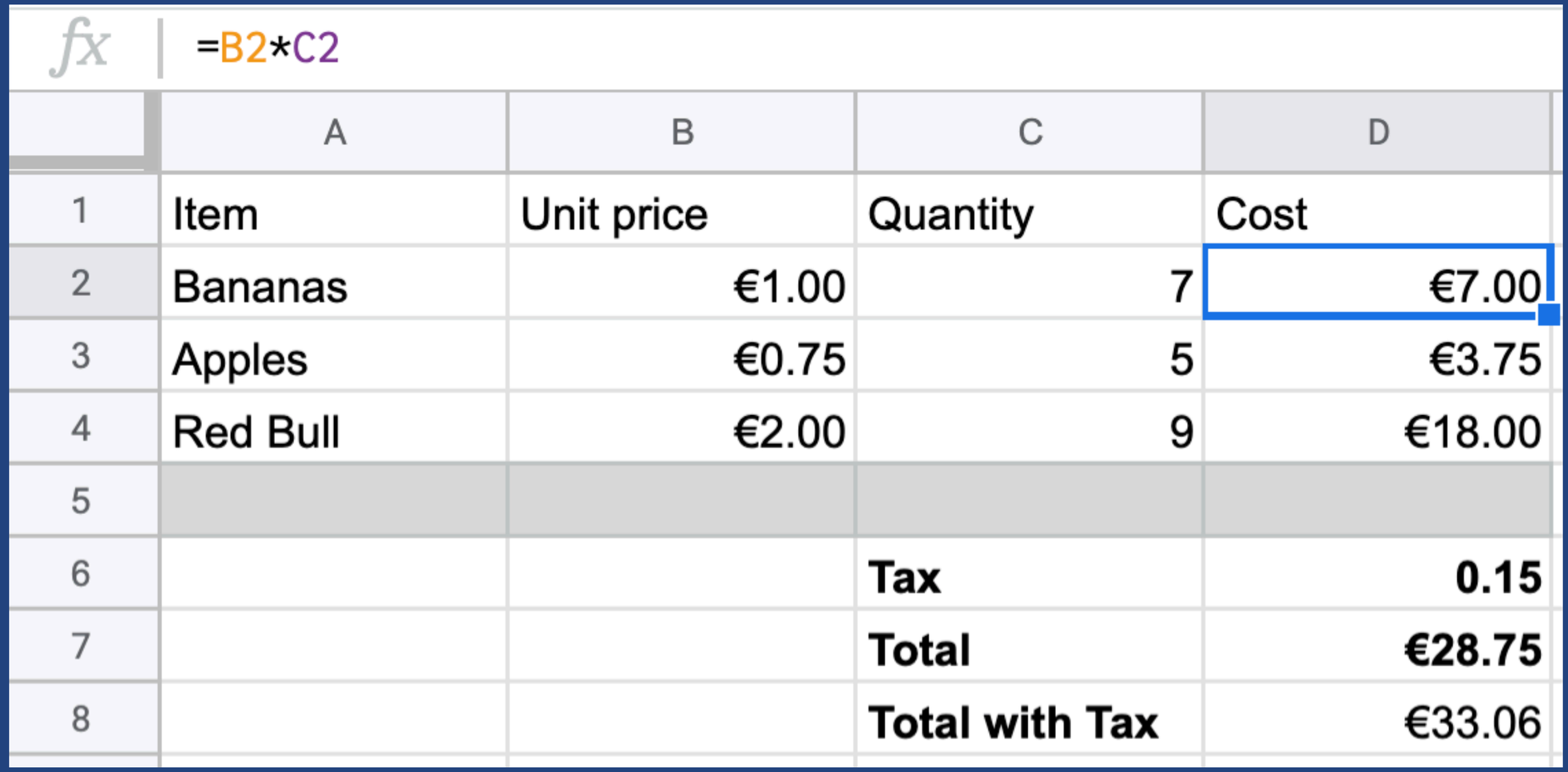
```
string  -> wool
sticks  -> wood
bow     -> sticks, string
arrow   -> sticks, feather, stone
quiver  -> arrow, bow
torches -> coal, sticks

λ> peek ["arrows"]  allDeps
fromList ["feathers","sticks","stone","wood"]

λ> peek ["arrows", "torches"]  allDeps
fromList ["coal","feathers","sticks","stone","wood"]

λ> peek ["quiver"]  allDeps
fromList ["arrows","bow","feathers","sticks","stone","string","wood","wool"]
```


A
B
C
tem
Unit price
Quantity
Cost
Bananas
€1.00
7
Tax
Total
Total with Tax
SPREADSHEETS

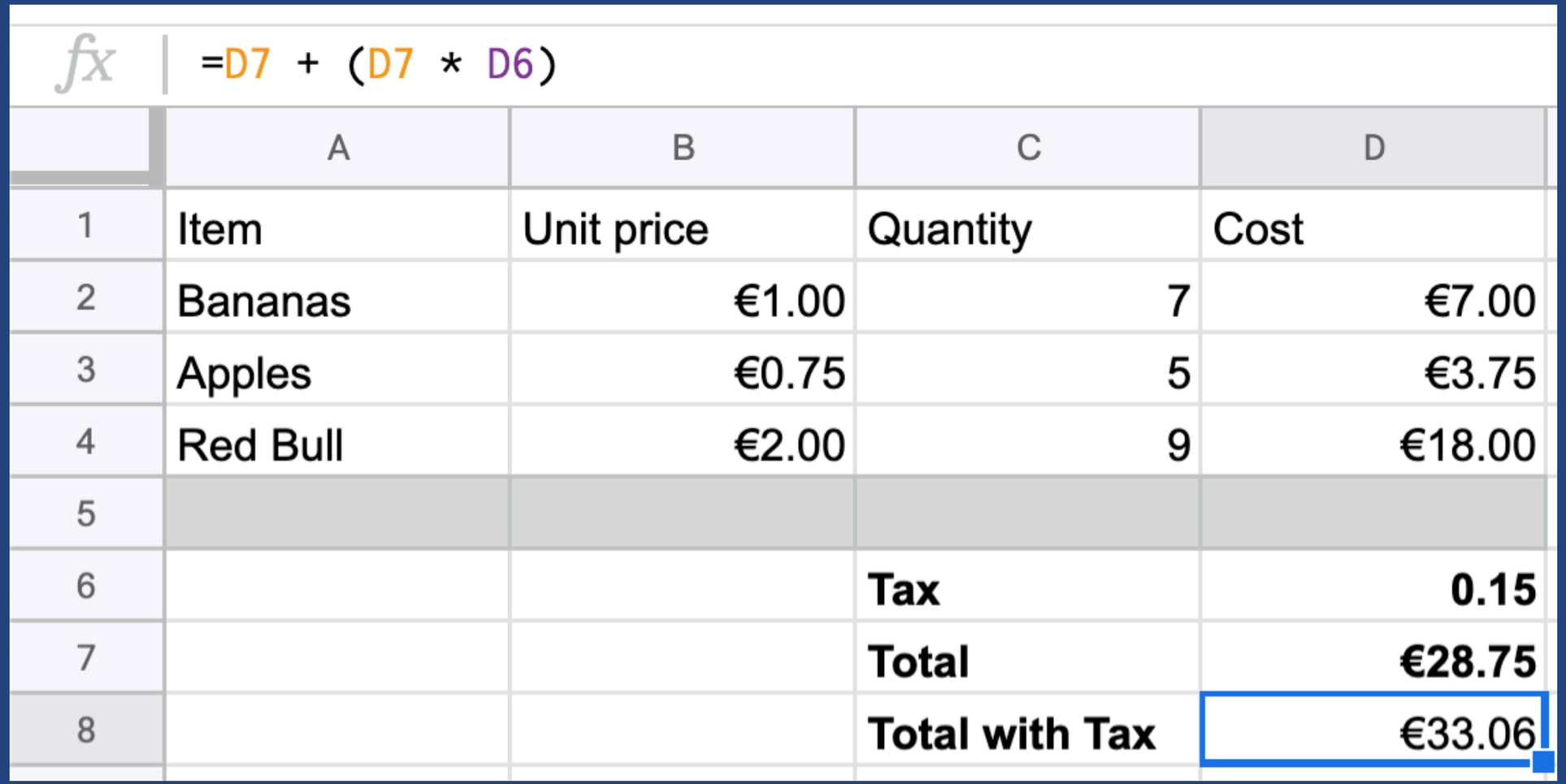
SPREADSHEETS

	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

FORMULAS

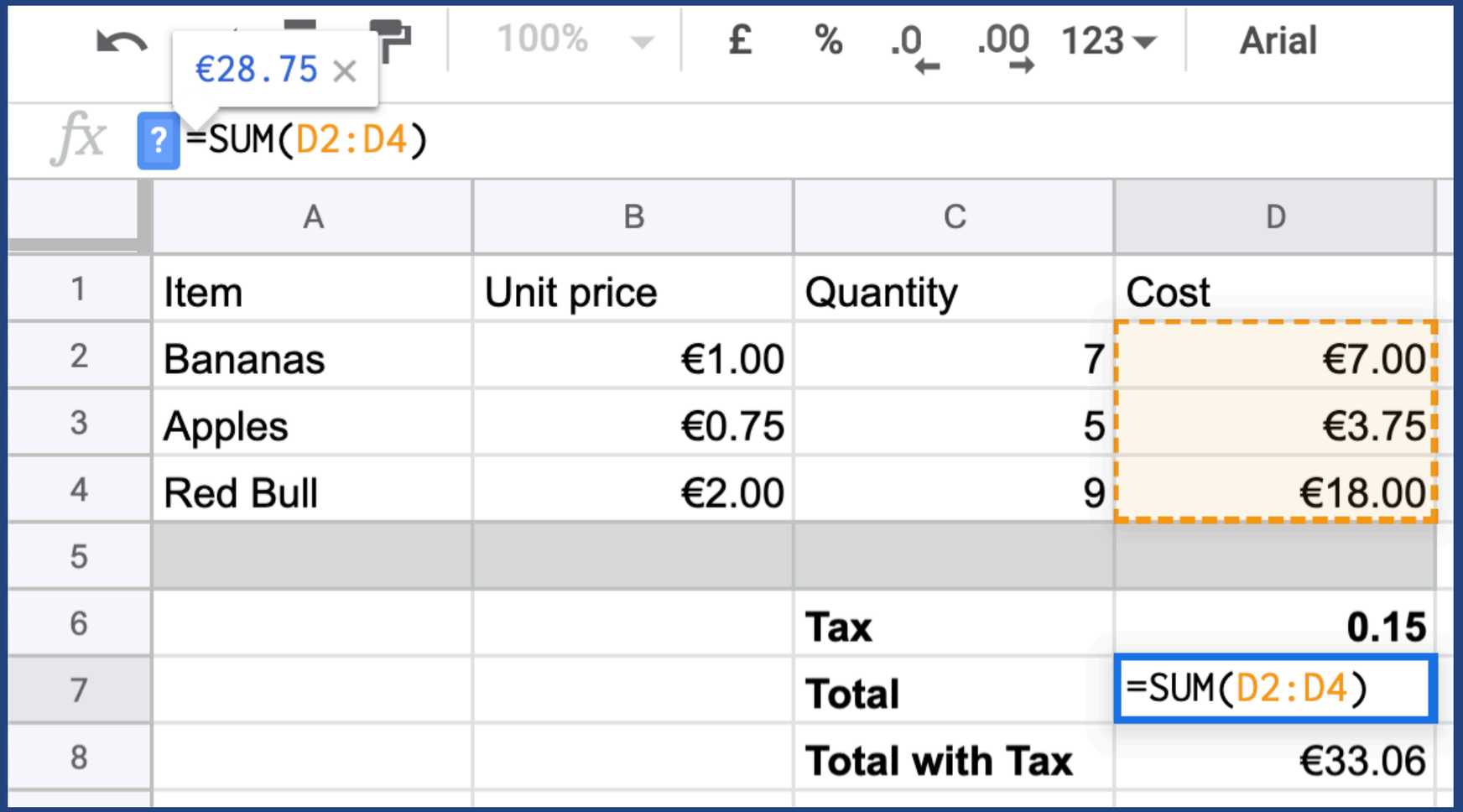

fx
=B2*C2
A
B
C
D
1
Item
Unit price
Quantity
Cost
2
Bananas
€1.00
7
€7.00
3
Apples
€0.75
5
€3.75
4
Red Bull
€2.00
9
€18.00
5
6
Tax
0.15
7
Total
€28.75
8
Total with Tax
€33.06

FORMULAS



	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

FORMULAS



	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**	=SUM(D2:D4)
8			**Total with Tax**	€33.06

```haskell
dataDef :: (Char, Int) -> Double
dataDef ('B', 2) = 1
dataDef ('B', 3) = 0.75
dataDef ('B', 4) = 2

dataDef ('C', 2) = 7
dataDef ('C', 3) = 5
dataDef ('C', 4) = 9

dataDef2 ('D',  row) = ???

dataDef _ = 0
```

	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

```
sheet :: Store (Char, Int) Double
sheet = (store (dataDef . first toUpper ) ('A', 1))

λ> peek ('B', 2) sheet
1.0

λ> peek ('B', 4) sheet
2.0

λ> peek ('C', 4) sheet
9.0
```

fx =B2*C2

	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

```haskell
dataDef2 :: (Char, Int)
         -> Store (Char, Int) Double
         ->  Double

-- B and C are the same; we ignore the new input...

dataDef2 ('D',  row) w | row < 6 =
  let price = peek ('B', row) w
      quant = peek ('C', row) w
   in price * quant
```

	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

```haskell
getCells :: Functor f => f s -> Store s a -> f a
getCells cells w = experiment (const cells) w

-- Tax
dataDef2 ('D', 6) _ = 0.15
-- Total
dataDef2 ('D', 7) w = sum . getCells (('D',) <$> [1..5]) $ w
-- Total With Tax
dataDef2 ('D', 8) w =
    let tax = peek ('D', 6) w
        total = peek ('D', 7) w
     in (tax * total) + total
```