```haskell
-- Why does this work:
g xs ys = map fst . filter (uncurry (==)) $ zip xs ys

-- But this does not:
g = map fst . filter (uncurry (==)) $ zip

-- i.e., why can't we just remove the arguments as we have done a
lot of times earlier? The problem is that zip will not consume both
arguments before being passed to the "filter" function. Thus, we do
not pass a list ("filter" expects a list) but the function "zip
xs". Why is this a function? Because it takes an argument (a list
(here ys)) and returns a list of tuples.
-- Thus we need to find a way to "force" zip to consume both
arguments first. This solves the problem:

g = (map (fst) . ) . (filter (uncurry (==)) . ) . zip

-- This is the best explanation I could come up with for this
solution (and why it does what we want), I hope its somewhat
understandable:

--Consider this fact:

(f . (g x)) = ((f . ) . g) x


-- What happens here on the right side and why is it equal to the
left side? Not trivial to see but during the evaluation, haskell
does this: The outermost function is the second (.). Thus, we could
theoretically rewrite the entire thing like this:

((.) (f .) g) x


-- Note that all I have done is rewriting the RHS from infix to
"normal" function notation. Now since we cannot evaluate further
without consuming the argument x, we proceed by doing so: The
function ((.) (f .) g) consumes an argument by passing it as an
argument to its "second argument", thus g.
-- [If you are advanced: What actually happens is that a new
function is created where first (.) consumes (f.) to create an
"intermediate" function which then consumes the function g to
create a function that consumes x. But this "final" function
consumes x in such a way that first x is applied to g and then the
result is passed to (f.)]

-- Hence, we get:

((.) (f .) g) x = (f .) (g x) = f . (g x)


-- Now, if we have two arguments, the following happens:

  ((f . ) . g) x y
= ((.) (f .) g) x y
= ((f .) (g x)) y
= (f . (g x)) y


-- Now, using the same logic (Def. of composition) as earlier, we
see`that the function (f . (g x))  consumes its argument (y) by
applying it to the second argument of (.), thus (g x), i.e. (here
the same note holds as earlier):

(f . (g x)) y = f ((g x) y) = f (g x y)


-- Thus if you have some function g that should consume two
arguments before being passed as an argument to the "next" function
f in a composition (in this exercise you'd use $ but in this
special case it is logically (not tech.!!) equivalent), you write
the composition as

(f .) . g


-- instead of just

f . g

--to make g consume the two arguments first.
```