U3SJEDR96 : and stuff like css animation would get messed up if you swapped the contents of two nodes with the same key
U2GPAEU1L : Ah I see. Ya if every key is the same then it'll make every key different, and if every key is different it's doing the overhead of checking if anything moved without success because there are no same-keys
U2LAL86AY : i will explain what i was thinking `map` means . there was this nice picture once done on `maybe` - which maybe is a container - the mapper - is an action of opening up the container - grabbing the value - give it to a function - take the result - put it inside the container - and close the cap. :smile: - this is me a couple months ago understanding `Maybe.map`. Then you say - this exact pattern is applied to a function. Inside my mind this analogy is broken now - or at least is confusing for now. But i guess i'm learning something. Here is a similar picture - i can't find that exact one..
U3SJEDR96 : And that's the _effect_ it has, if you consider a function as data, I like that drawing :slightly_smiling_face:
U48AEBJQ3 : <@U2LAL86AY> That is not a bad way to think of things, but I like to think of it a bit differently. `map` is a function which transforms functions. If you have a function `a -> b` lying around, `Maybe.map` will transform it for you to a function of `Maybe a -> Maybe b`.
U0K92QFST : <@U3SJEDR96> I do consider functions to be data! This is how you can tell I know a bit of Haskell :stuck_out_tongue:
U0K92QFST : <@U2LAL86AY> I remember reading the same blog post; that's a very good way to think about it
U23SA861Y : If you consider mapping to be "apply this function to this boxed data and return to me a box of the same shape", then you can expand this to functions where is the "data" is the algorithm and the box is the abstraction of a function. So now you apply a function to a function shaped box to get another function which just so happens to be composition.
U0K92QFST : :point_up: great summary!
U23SA861Y : ```type Function a b = Function (a->b)

map: (b->c) -> Function a b -> Function a c
map f (Function g) =
  Function (\x -> f (g x))
```

U23SA861Y : In this case there are two types so there is an equally valid map of```
map_: (a-&gt;c) -&gt; Function a b -&gt; Function c b
map_ f (Function g) =
  Function (\x -&gt; g (f x))
```

U23SA861Y : maybe call the first one `andThen` and the second `map`
U3SJEDR96 : `andThen` would _probably_ be something like `andThen : (a -> Function c b) -> Function c a -> Function c b` (i.e. `bind` in other languages)
U3SJEDR96 : in Elm, I think your second version of `map` would be something like `mapInput`, and you first one would be just `map`
U37HUSJ4R : If my model looks like:```
type alias Application =
    { id : Int
    , term : Int
    , amount : Int
    }

type alias Model =
    { application : Application
    }
```

and I am trying to update the `term` value, I have `onInput UpdateTerm` on an input

U37HUSJ4R : inside my update case statement how do I update this value?
U23SA861Y : sure...
U23SA861Y : pretends to know what he's talking about
U2GPAEU1L : btw was still reading, forgot to say thanks, thanks <@U3SJEDR96> and <@U0EUHKVGB>
U0EUHKVGB : jonf: Can you clarify what you mean by this? Comes across a little hostile
U23SA861Y : as in I pretend to know what what I'm talking about
```

U0EUHKVGB : <@U37HUSJ4R> Try asking in <#C192T0Q1E|beginners> :slightly_smiling_face:
U3SJEDR96 : <@U23SA861Y> I don't mean anything by it, just pointing out how those names tend to be used in common Elm libraries :slightly_smiling_face:
U3SJEDR96 : good call that there are two valid ways to define a `map` for a `Function a b`, tho - in which case one usually guesses which one would be the most commonly used (like `Result.map` maps the `Ok`, not the `Err`), and defining an alternative for the other one (like `Result.mapError`)