Leveling Up Clojure Runtime Specs

Madison Clojure

Ambrose Bonnaire-Sergeant

Stream starting soon...

Leveling Up Clojure Runtime Specs

Ambrose Bonnaire-Sergeant

Programming <u>before</u> Specs

1. Write the program
2. Try to break it
3. Fix the program

```
f(x)=1
```

“Takes an argument x and returns x.”

```
f(1)=>1 ✓
f(“hello”)=>“hello” ✗
```

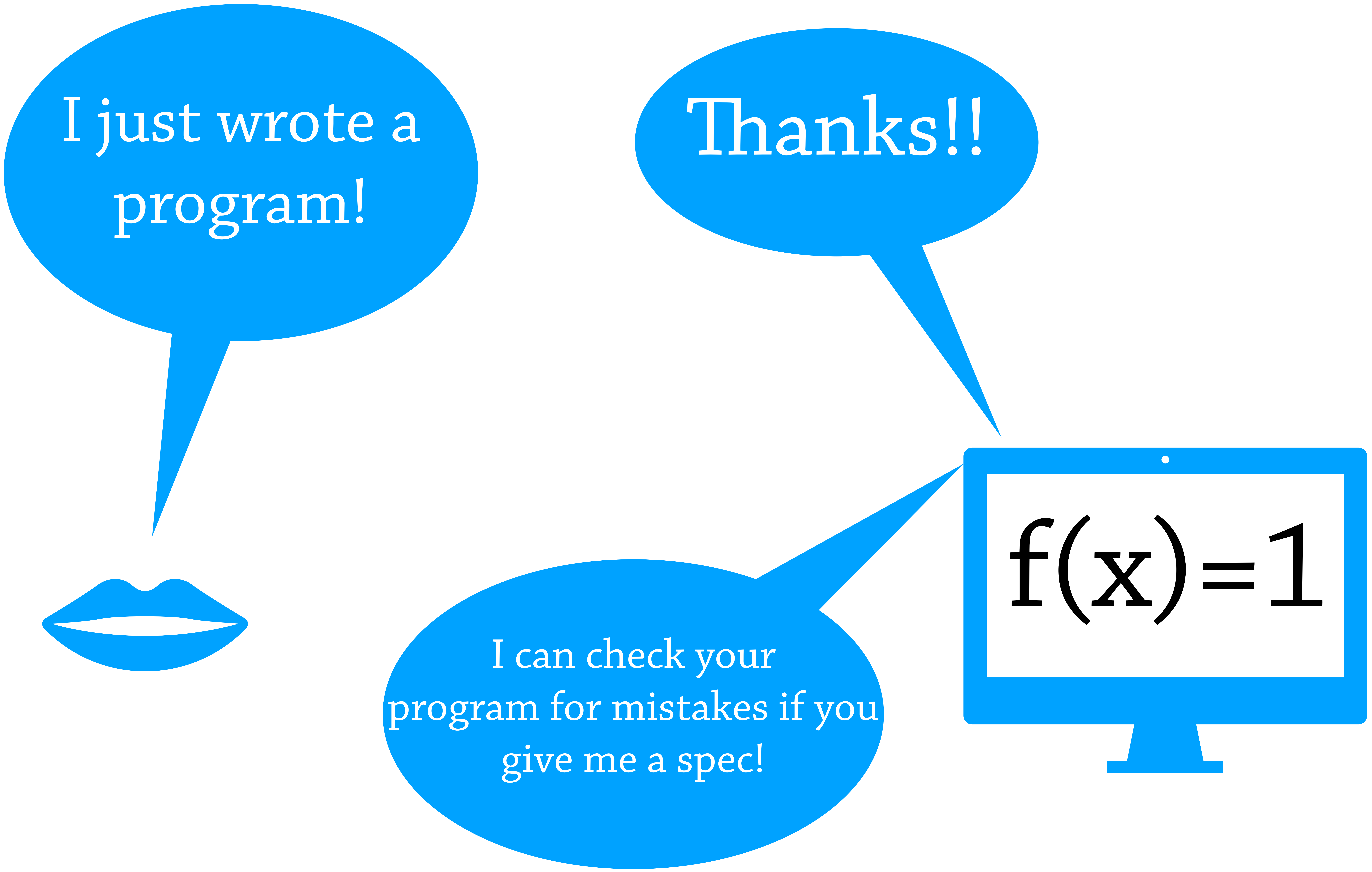
```
f(x)=x
```

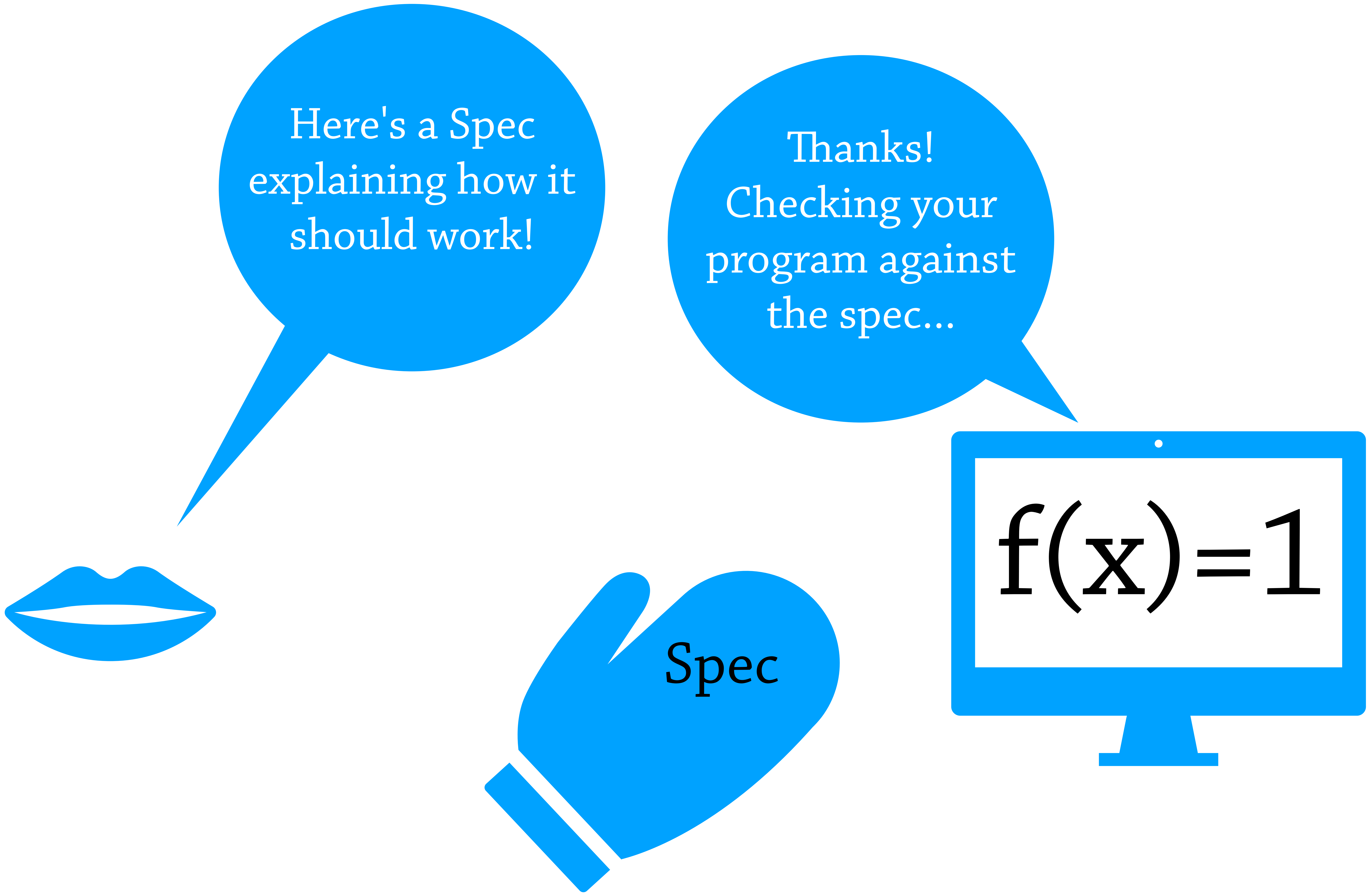
“Takes an argument x and returns x.”

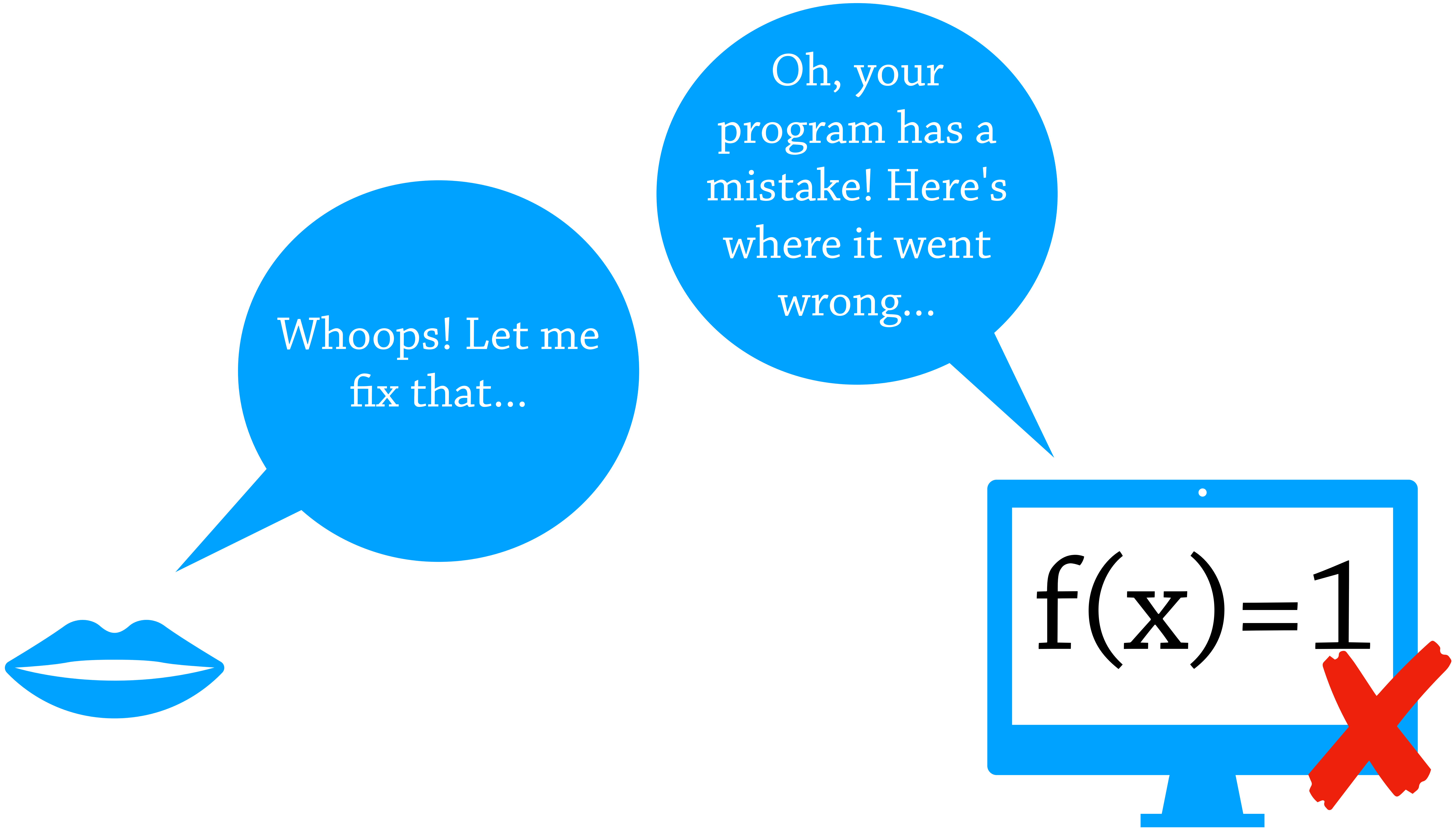
```
f(1)=>1 ✓
f(“hello”)=>“hello” ✓
```

Programming
after Specs

1. Write the program
2. Write a "spec"
3. ?????????????
4. Fix the program


I just wrote a program!
Thanks!!
I can check your program for mistakes if you give me a spec!
f(x)=1


Here's a Spec explaining how it should work!
Thanks! Checking your program against the spec...
Spec
f(x)=1


Whoops! Let me fix that...
Oh, your program has a mistake! Here's where it went wrong...
f(x)=1

schema

clojure.spec

Malli

Intro to specs (via Malli)

```
{:street "Washington Ave",
 :city "Madison"
 :zip 53701
 :lonlat [43.0812792448301, -89.37430643983365]}
```

Address

```
(def Address
   [:map
    [:street string?]
    [:city string?]
    [:zip int?]
    [:lonlat [:tuple double? double?]]])
```

Spec for Addresses

```
{:street "Washington Ave",
 :city "Madison"
 :zip 53701
 :lonlat [43.0812792448301, -89.37430643983365]}
```

```
(def Address
   [:map
    [:street string?]
    [:city string?]
    [:zip int?]
    [:lonlat [:tuple double? double?]]])
```

```
{:street "Washington Ave",
 :city "Madison"
 :zip 53701
 :lonlat [43.0812792448301, -89.37430643983365]}
```

```
(def Address
   [:map
    [:street string?]
    [:city string?]
    [:zip int?]
    [:lonlat [:tuple double? double?]]])
```

```
{:street "Washington Ave",
 :city "Madison"
 :zip 53701
 :lonlat [43.0812792448301, -89.37430643983365]}
```

```
(def Address
   [:map
    [:street string?]
    [:city string?]
    [:zip int?]
    [:lonlat [:tuple double? double?]]])
```

```
{:street "Washington Ave",
 :city "Madison"
 :zip 53701
 :lonlat [43.0812792448301, -89.37430643983365]}
```

```
(def Address
  [:map
   [:street string?]
   [:city string?]
   [:zip int?]
   [:lonlat [:tuple double? double?]]])
```

```
(def Address
  [:map
   [:street string?]
   [:city string?]
   [:zip int?]
   [:lonlat [:tuple double? double?]]])
```

Validate

"Does this value conform to this spec?"

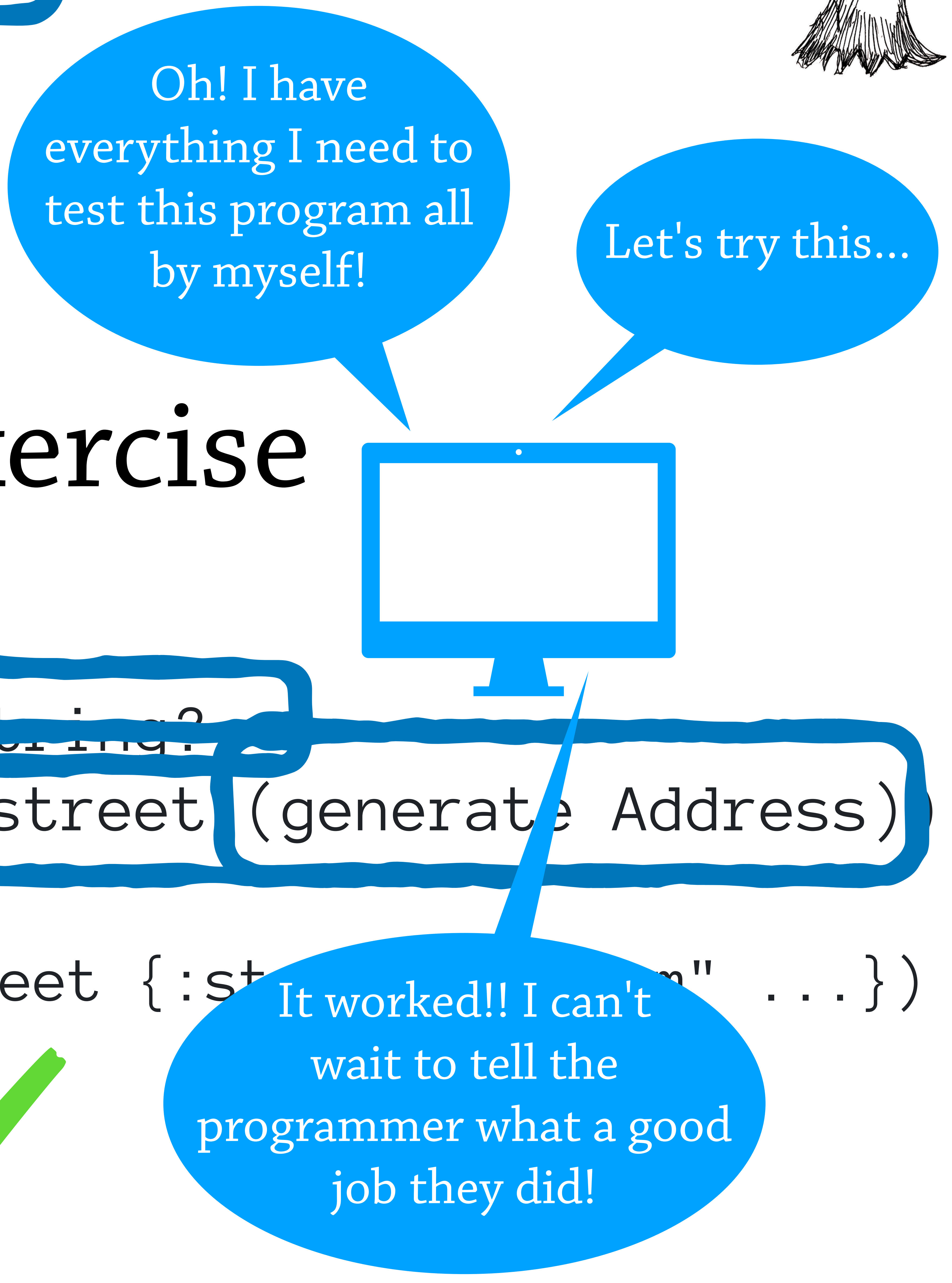
```
(explain
  Address
  {:street "Washington Ave",
   :city "Madison"}))
=>
{:zip ["missing required key"],
 :lonlat ["missing required key"]}
```

Generate

"Create an example value for this spec."

```
(generate Address)
=>
 :street "0D8916M7fZ3gGz48eNRZz86Q3100",
 :city "",
 :zip -1,
 :lonlat [96.5218505859375 -156.7041015625]
```

```
(=> address-street [:=> Address string?])
(defn address-street [address]
  (:street address))
```


Oh! I have everything I need to test this program all by myself!
Let's try this...
It worked!! I can't wait to tell the programmer what a good job they did!

Instrument

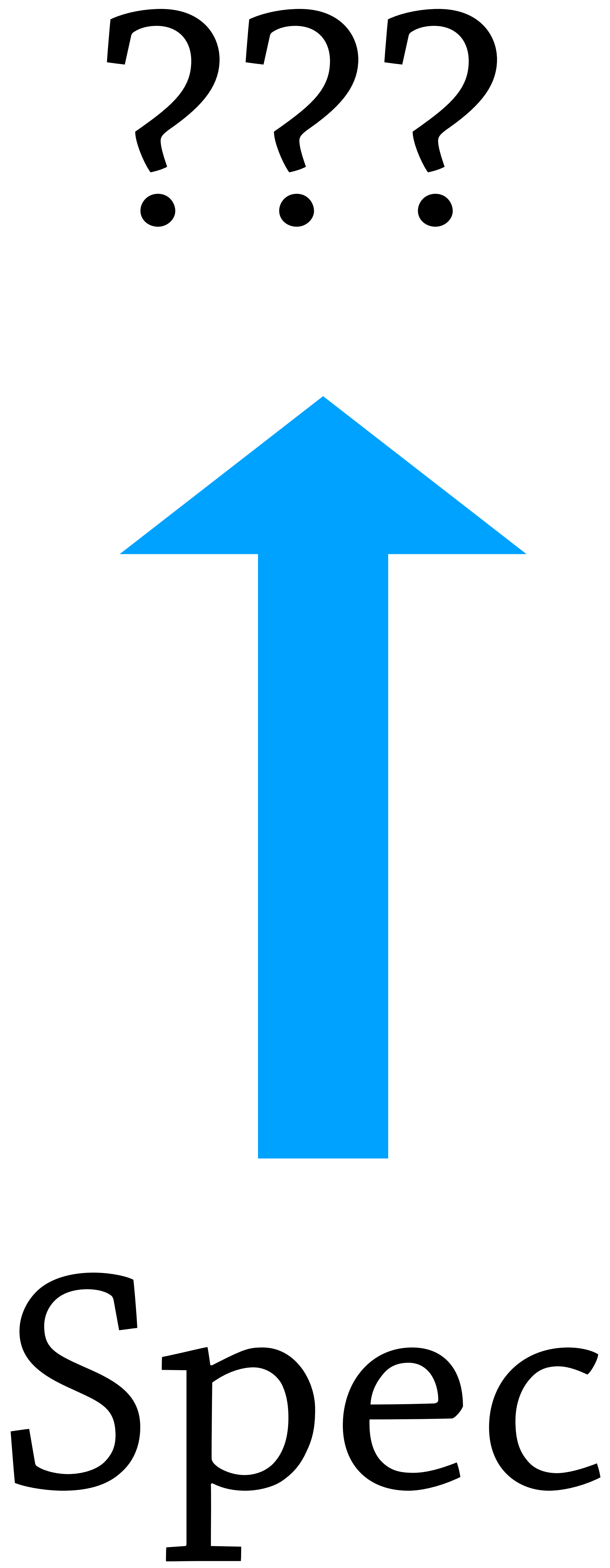
```
(defn address-street [address]
  (coerce Address address)
  (coerce string? (:street address)))

(address-street {:street 52 ...})
```

Exercise

```
(~~validate string?~~
  (address-street (generate Address)))

(address-street {:st        m" ...})
=> "random"
```

Oh! I want you to check something specific about this program, but I don't think you support it!
I can help you find even more mistakes if you make a really specific spec!

This talk



Leveling-Up Function Specs

Data flow

identity

"Returns its argument."

```
(identity "a") => "a"

(identity 1) => 1

(identity nil) => nil
```

identity

"Returns its argument."

schema	Any -> Any
spec	any? -> any?
malli	:any -> :any

Sequences

Collections

Create

Sequences > Create

`seq`	Returns a seq on the collection. If the collection is empty, returns nil. (se…
`sequence`	Coerces coll to a (possibly empty) sequence, if it is not already one. Will…
`eduction`	Returns a reducible/iterable application of the transducers to the items i…
`repeat`	Returns a lazy (infinite!, or length n if supplied) sequence of xs.
`replicate`	DEPRECATED: Use 'repeat' instead. Returns a lazy seq of n xs.
`range`	Returns a lazy seq of nums from start (inclusive) to end (exclusive), by s…
`repeatedly`	Takes a function of no args, presumably with side effects, and returns a…
`iterate`	Returns a lazy sequence of x, (f x), (f (f x)) etc. f must be free of side-effe…
`lazy-seq`	Takes a body of expressions that returns an ISeq or nil, and yields a Se…
`lazy-cat`	Expands to code which yields a lazy sequence of the concatenation of t…
`cycle`	Returns a lazy (infinite!) sequence of repetitions of the items in coll.
`interleave`	Returns a lazy seq of the first item in each coll, then the second etc.
`interpose`	Returns a lazy seq of the elements of coll separated by sep. Returns a s…
`tree-seq`	Returns a lazy sequence of the nodes in a tree, via a depth-first walk. br…
`xml-seq`	A tree seq on the xml elements as per xml/parse
`enumeration-seq`	Returns a seq on a java.util.Enumeration
`iterator-seq`	Returns a seq on a java.util.Iterator. Note that most collections providin…
`file-seq`	A tree seq on java.io.Files
`line-seq`	Returns the lines of text from rdr as a lazy sequence of strings. rdr must…

Use ('Modification')

Sequences > Use ('Modification')

`conj`	conj[oin]. Returns a new collection with the xs 'added'. (conj nil item) ret…
`concat`	Returns a lazy seq representing the concatenation of the elements in th…
`distinct`	Returns a lazy sequence of the elements of coll with duplicates remove…
`group-by`	Returns a map of the elements of coll keyed by the result of f on each el…
`partition`	Returns a lazy sequence of lists of n items each, at offsets step apart. If…
`partition-all`	Returns a lazy sequence of lists like partition, but may include partitions…
`partition-by`	Applies f to each value in coll, splitting it each time f returns a new valu…
`split-at`	Returns a vector of [(take n coll) (drop n coll)]
`split-with`	Returns a vector of [(take-while pred coll) (drop-while pred coll)]
`filter`	Returns a lazy sequence of the items in coll for which (pred item) return…
`filterv`	Returns a vector of the items in coll for which (pred item) returns logical…
`remove`	Returns a lazy sequence of the items in coll for which (pred item) return…
`replace`	Given a map of replacement pairs and a vector/collection, returns a vec…
`shuffle`	Return a random permutation of coll
`random-sample`	Returns items from coll with random probability of prob (0.0 - 1.0). Retu…
`flatten`	Takes any nested combination of sequential things (lists, vectors, etc.) a…
`sort`	Returns a sorted sequence of the items in coll. If no comparator is supp…
`sort-by`	Returns a sorted sequence of the items in coll, where the sort order is d…
`reverse`	Returns a seq of the items in coll in reverse order. Not lazy.
`dedupe`	Returns a lazy sequence removing consecutive duplicates in coll. Retur…

Use (General)

Sequences > Use (General)

`first`	Returns the first item in the collection. Calls seq on its argument. If coll i…
`second`	Same as (first (next x))
`last`	Return the last item in coll, in linear time
`rest`	Returns a possibly empty seq of the items after the first. Calls seq on it…
`next`	Returns a seq of the items after the first. Calls seq on its argument. If th…
`ffirst`	Same as (first (first x))
`nfirst`	Same as (next (first x))
`fnext`	Same as (first (next x))
`nnext`	Same as (next (next x))
`nth`	Returns the value at the index. get returns nil if index out of bounds, nth…
`nthnext`	Returns the nth next of coll, (seq coll) when n is 0.
`nthrest`	Returns the nth rest of coll, coll when n is 0.
`rand-nth`	Return a random element of the (sequential) collection. Will have the sa…
`butlast`	Return a seq of all but the last item in coll, in linear time
`take`	Returns a lazy sequence of the first n items in coll, or all items if there ar…
`take-last`	Returns a seq of the last n items in coll. Depending on the type of coll …
`take-nth`	Returns a lazy seq of every nth item in coll. Returns a stateful transduce…
`take-while`	Returns a lazy sequence of successive items from coll while (pred item) …
`drop`	Returns a lazy sequence of all but the first n items in coll. Returns a stat…
`drop-last`	Return a lazy sequence of all but the last n (default 1) items in coll
`drop-while`	Returns a lazy sequence of the items in coll starting from the first item f…

Use (Iteration)

Sequences > Use (Iteration)

`map`	Returns a lazy sequence consisting of the result of applying f to the set …
`mapv`	Returns a vector consisting of the result of applying f to the set of first it…
`map-indexed`	Returns a lazy sequence consisting of the result of applying f to 0 and t…
`keep`	Returns a lazy sequence of the non-nil results of (f item). Note, this mea…
`keep-indexed`	Returns a lazy sequence of the non-nil results of (f index item). Note, thi…
`mapcat`	Returns the result of applying concat to the result of applying map to f a…
`reduce`	f should be a function of 2 arguments. If val is not supplied, returns the …
`reductions`	Returns a lazy seq of the intermediate values of the reduction (as per re…
`transduce`	reduce with a transformation of f (xf). If init is not supplied, (f) will be call…
`max-key`	Returns the x for which (k x), a number, is greatest. If there are multiple …
`min-key`	Returns the x for which (k x), a number, is least. If there are multiple suc…
`doall`	When lazy sequences are produced via functions that have side effects,…
`dorun`	When lazy sequences are produced via functions that have side effects,…

Sets

Co

Create

`hash-set`	Returns a new hash set with supplied keys. Any equal key
`set`	Returns a set of the distinct elements of coll.
`sorted-set`	Returns a new sorted set with supplied keys. Any equal ke
`sorted-set-by`	Returns a new sorted set with supplied keys, using the su

Use

`conj`	conj[oin]. Returns a new collection with the xs 'added'. (co
`disj`	disj[oin]. Returns a new set of the same (hashed/sorted) ty
`get`	Returns the value mapped to key, not-found or nil if key no

Transients

Collectior

Create

Transients > Cre

`transient`	Returns a new, transient version of the collection, in constant time.
`persistent!`	Returns a new, persistent version of the transient collection, in constar

Use (General)

Transients > Use (Gene

`conj!`	Adds x to the transient collection, and return coll. The 'addition' may h
`pop!`	Removes the last item from a transient vector. If the collection is empty
`assoc!`	When applied to a transient map, adds mapping of key(s) to val(s). Wh
`dissoc!`	Returns a transient map that doesn't contain a mapping for key(s).
`disj!`	disj[oin]. Returns a transient set of the same (hashed/sorted) type, that

Vectors

Create

`vec`	Creates a new vector containing the contents o
`vector`	Creates a new vector containing the args.
`vector-of`	Creates a new vector of a single primitive type t

Lists

Create

`list`	Creates a new list containing the items.

identity

"Returns its argument."

```
     Any -> Any
Int|Bool -> Int|Bool
     Int -> Int
    Bool -> Bool
  (eq 1) -> (eq 1)
```

/schema

`for all specs X,`

$$\begin{tikzcd} X \arrow[rr, bend right, dotted] & \text{->} & X \end{tikzcd}$$

typed.clj.spec

identity

"Returns its argument."

```
for all specs X,
    X -> X
```

```clojure
(s/def
  ::identity-poly
  (t/all :binder (t/binder :x (t/bind-tv))
         :body
         (s/fspec :args (s/cat :x (t/tv :x))
                  :ret (t/tv :x))))
```

https://tinyurl.com/typed-clj-spec

identity

"Returns its argument."

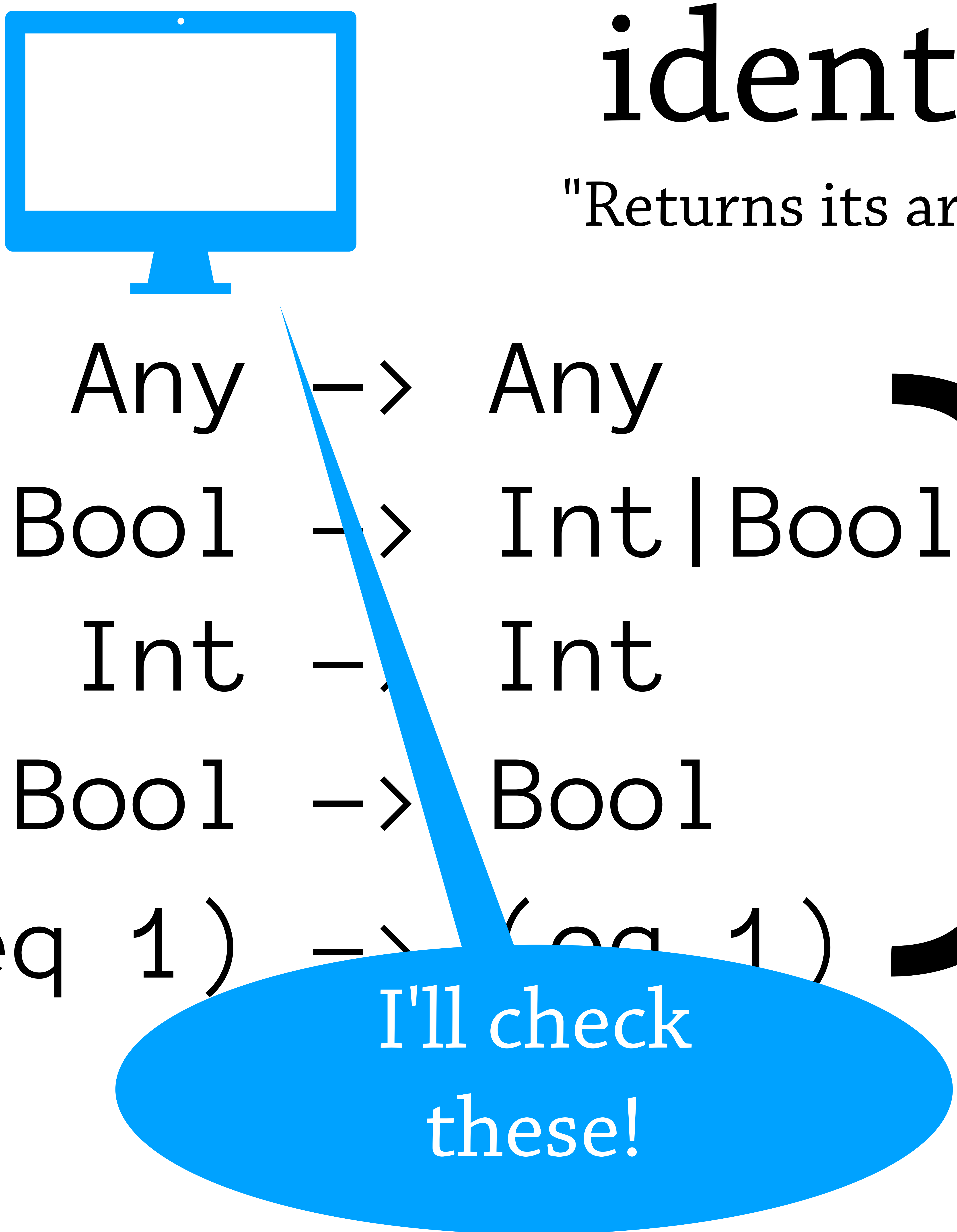
```
(tu/is-valid ::identity-poly identity)

(tu/is-invalid ::identity-poly (fn [x] nil))
```

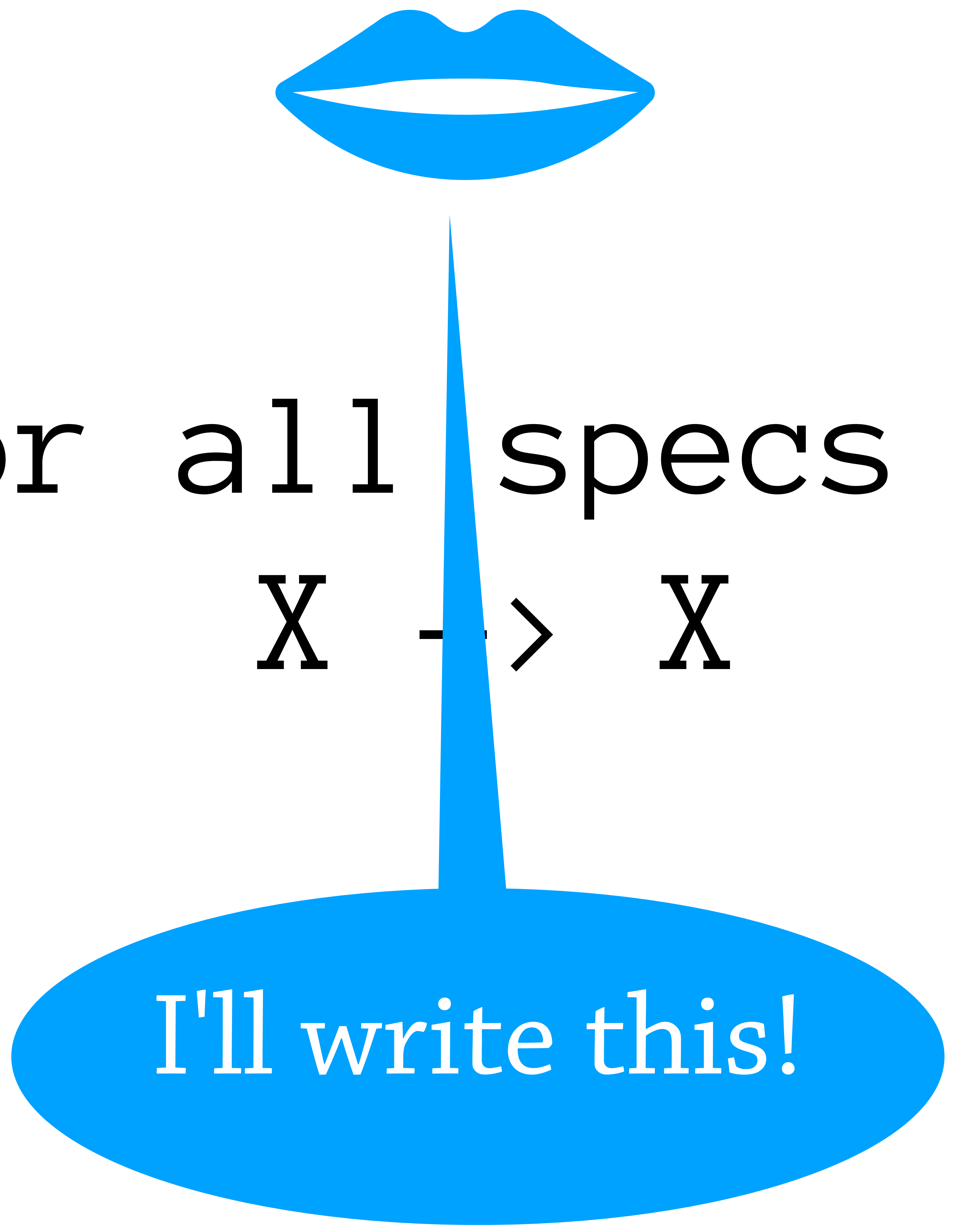
identity

"Returns its argument."

```
      Any -> Any
 Int|Bool -> Int|Bool
      Int -> Int
     Bool -> Bool
   (eq 1) -> (eq 1)
```

```
for all specs X,
    X -> X
```





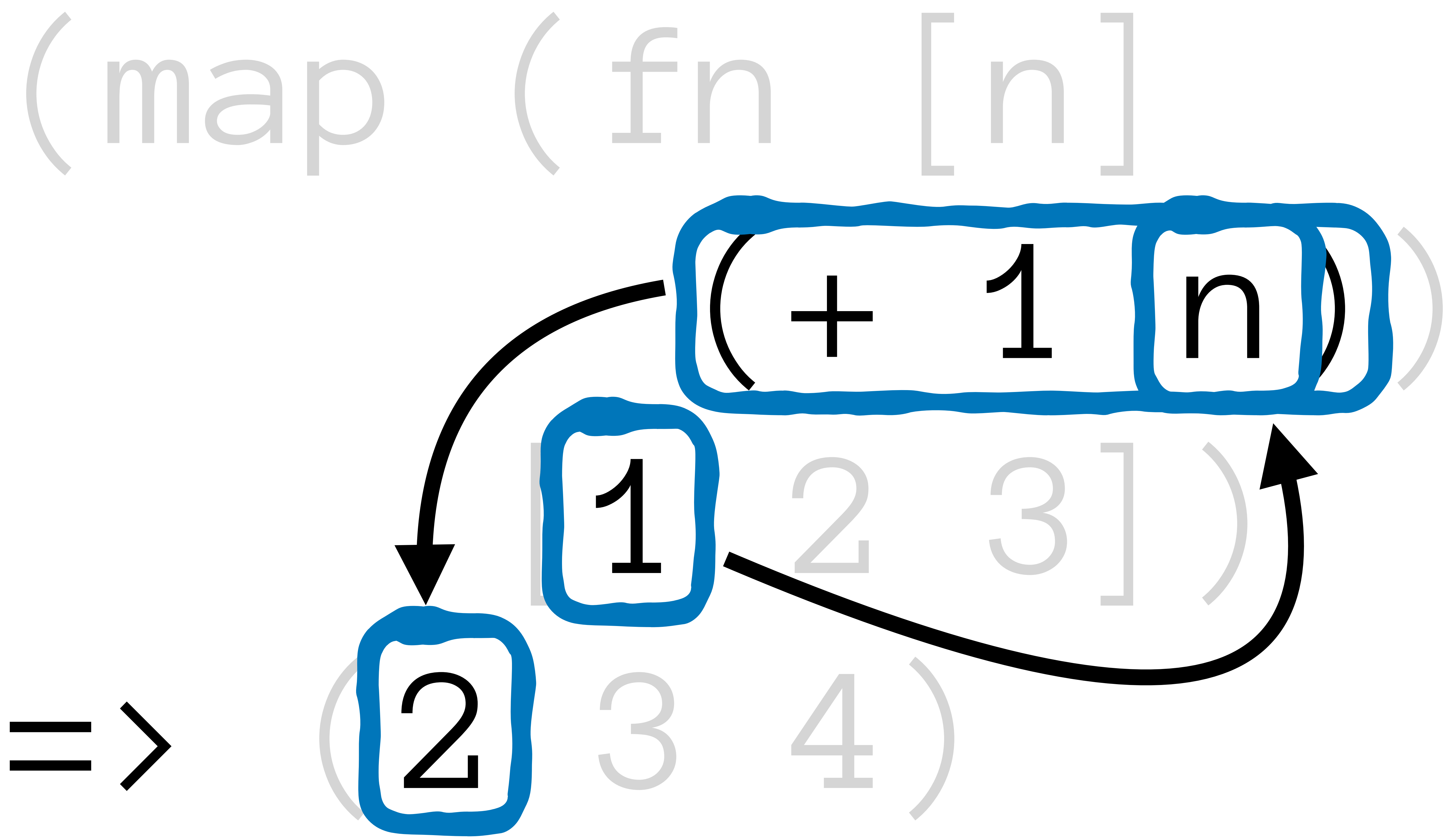
map

"Applies the function to each element of the collection."

```
(map (fn [n]
          (+ 1 n))
        [1 2 3])
=> (2 3 4)
```

map

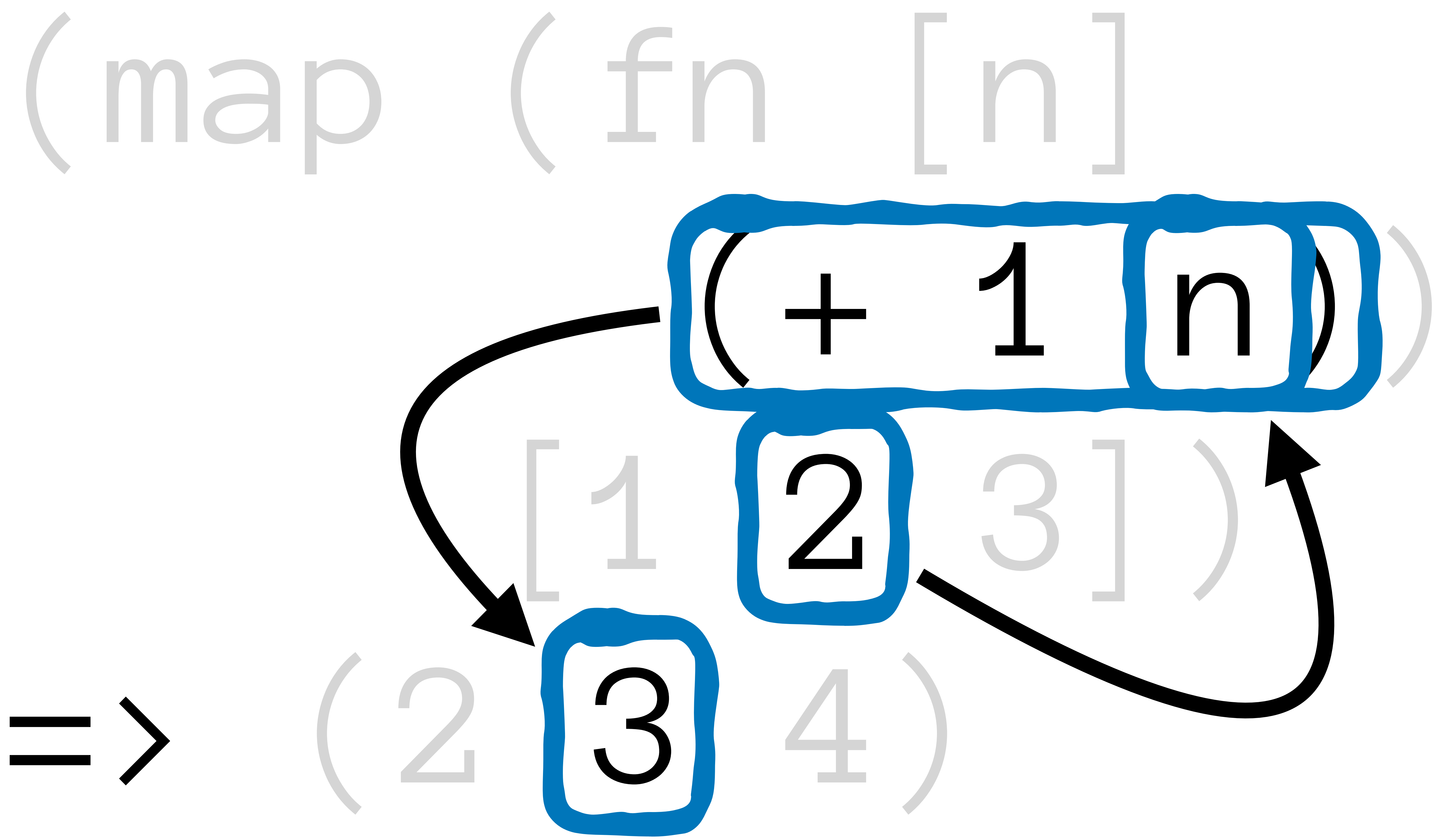
"Applies the function to each element of the collection."



```
(map (fn [n]
          (+ 1 n))
       [1 2 3])
=>  (2 3 4)
```

map

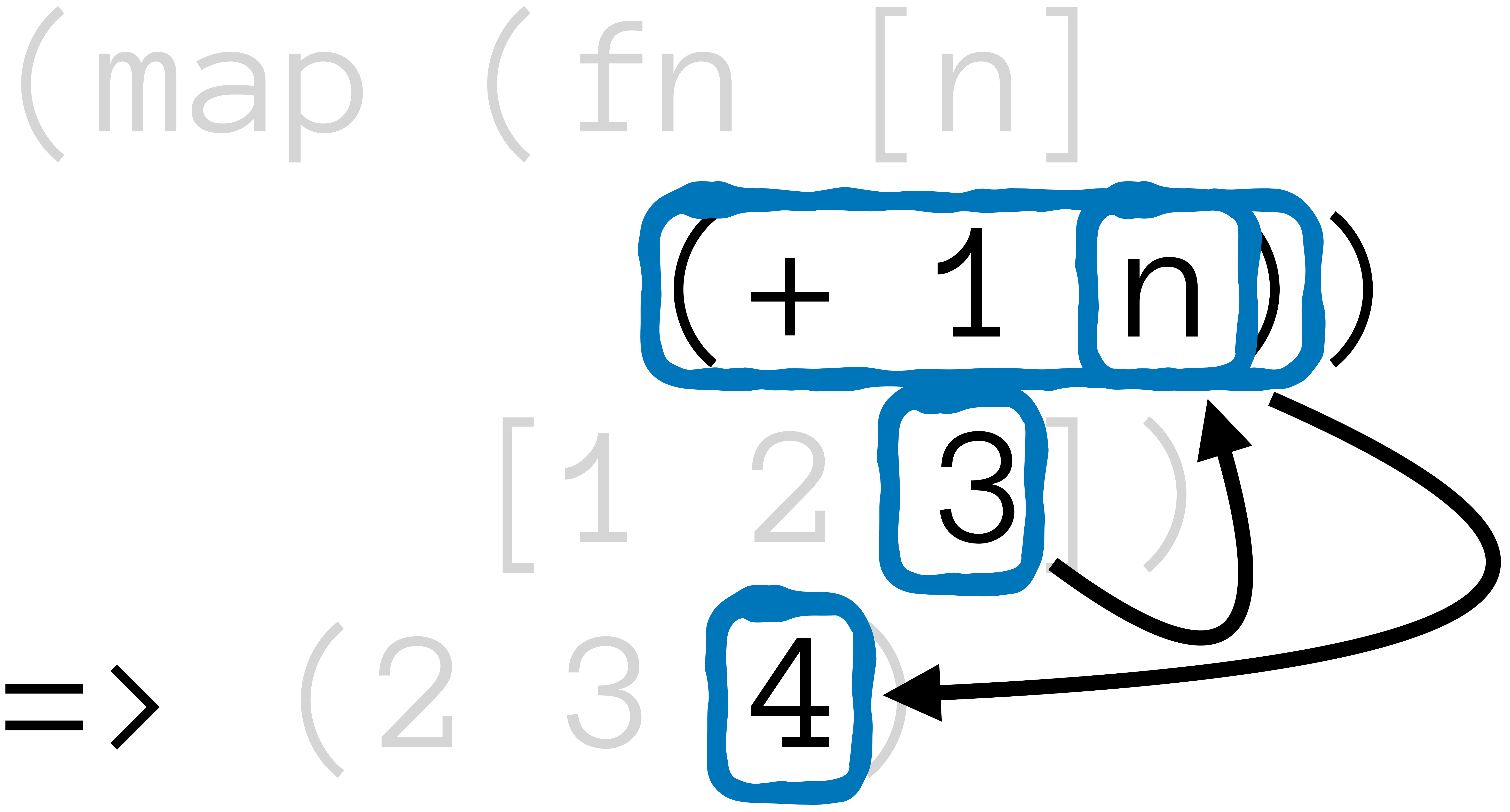
"Applies the function to each element of the collection."



```
(map (fn [n]
          (+ 1 n))
        [1 2 3])

=>  (2 3 4)
```

map

"Applies the function to each element of the collection."



```
(map (fn [n]
       (+ 1 n))
     [1 2 3])
=> (2 3 4)
```

map

"Applies the function to each element of the collection."

/schema	`(Any->Any) [Any] -> [Any]`
spec	`(any? -> any?) (every any?) -> (every any?)`
malli	`[:=> :any :any] [:sequential :any :any] -> [:sequential :any]`

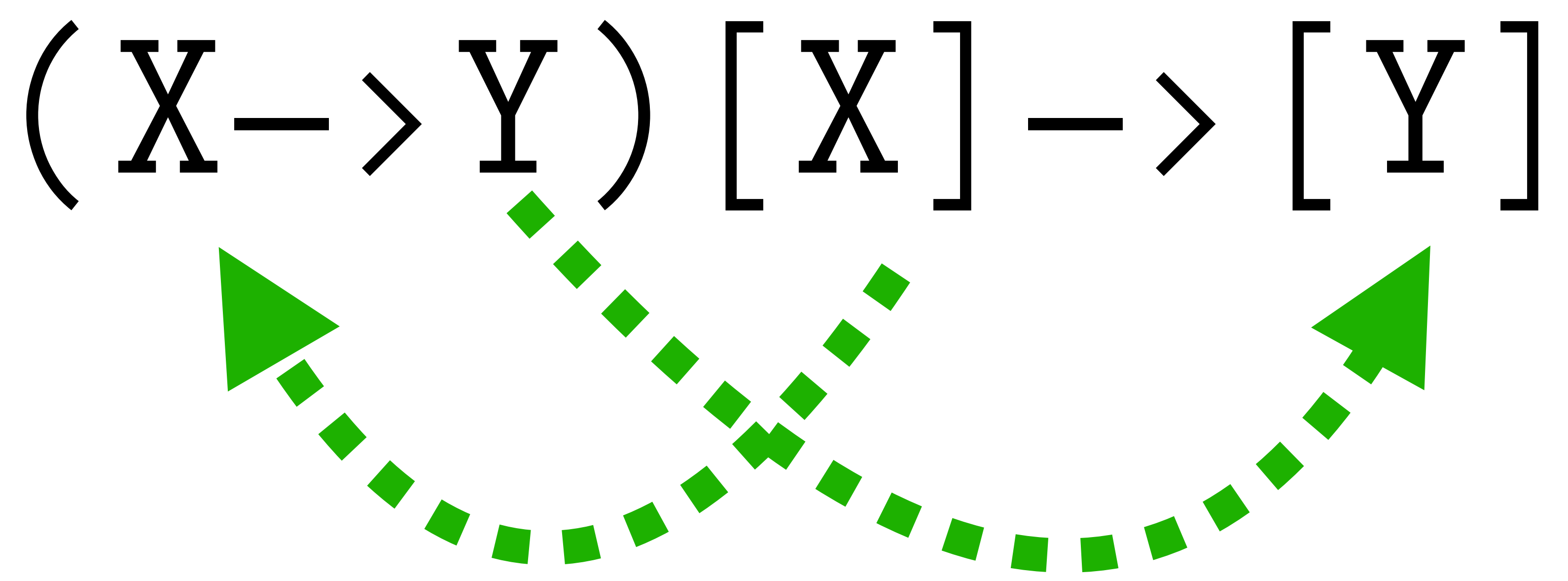
map

"Applies the function to each element of the collection."

```
(Any->Any)[Any]->[Any]

(Int->Str)[Int]->[Str]

(1->2)[1]->[2]
```

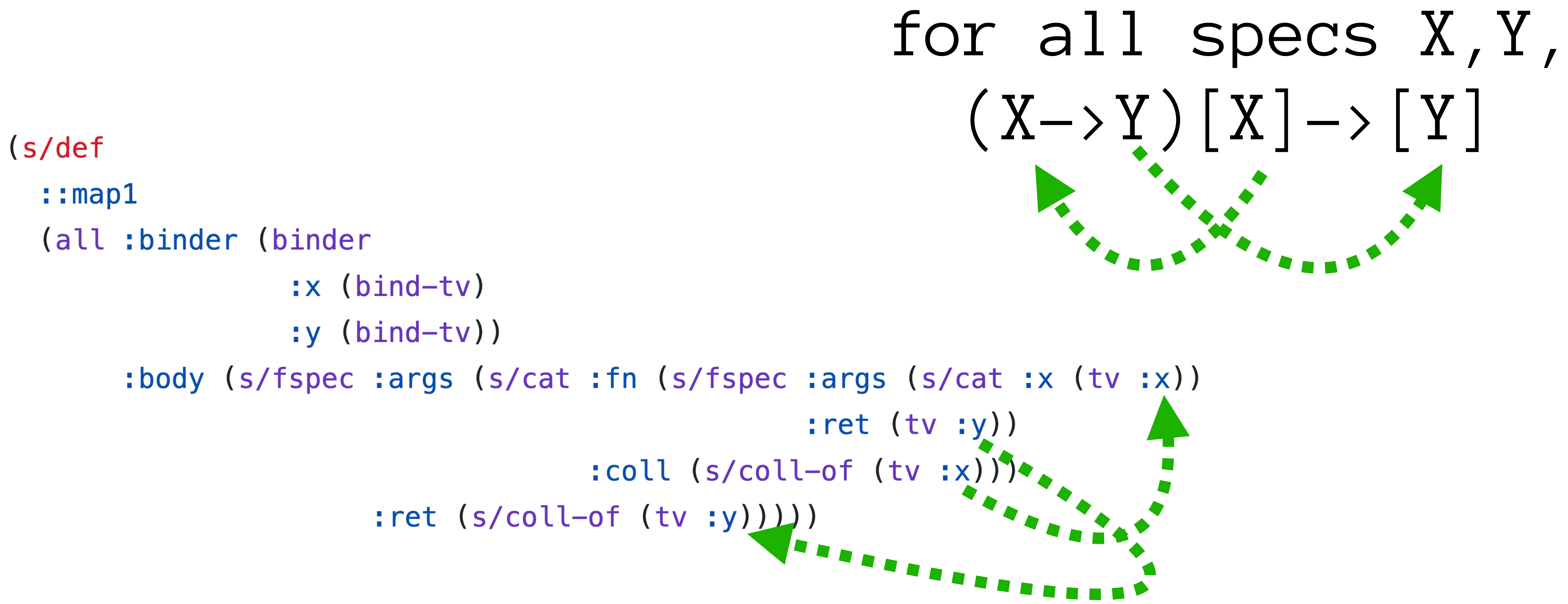
/schema

```
for all specs X,Y,
(X->Y)[X]->[Y]
```



map

"Applies the function to each element of the collection."



```
for all specs X,Y,
(X->Y)[X]->[Y]
```

```
(s/def
  ::map1
  (all :binder (binder
                 :x (bind-tv)
                 :y (bind-tv))
       :body (s/fspec :args (s/cat :fn (s/fspec :args (s/cat :x (tv :x))
                                                :ret (tv :y))
                                   :coll (s/coll-of (tv :x)))
                      :ret (s/coll-of (tv :y)))))
```

map

"Applies the function to each element of the collection."

```
(tu/is-valid ::map1 map)
```

✓

```
(tu/is-invalid ::map1 (comp #(map str %) map))
```

✗

map

"Applies the function to each element of the collection."

```
(Any->Any)[Any]->[Any]

(Int->Str)[Int]->[Str]

(1->2)[1]->[2]
```

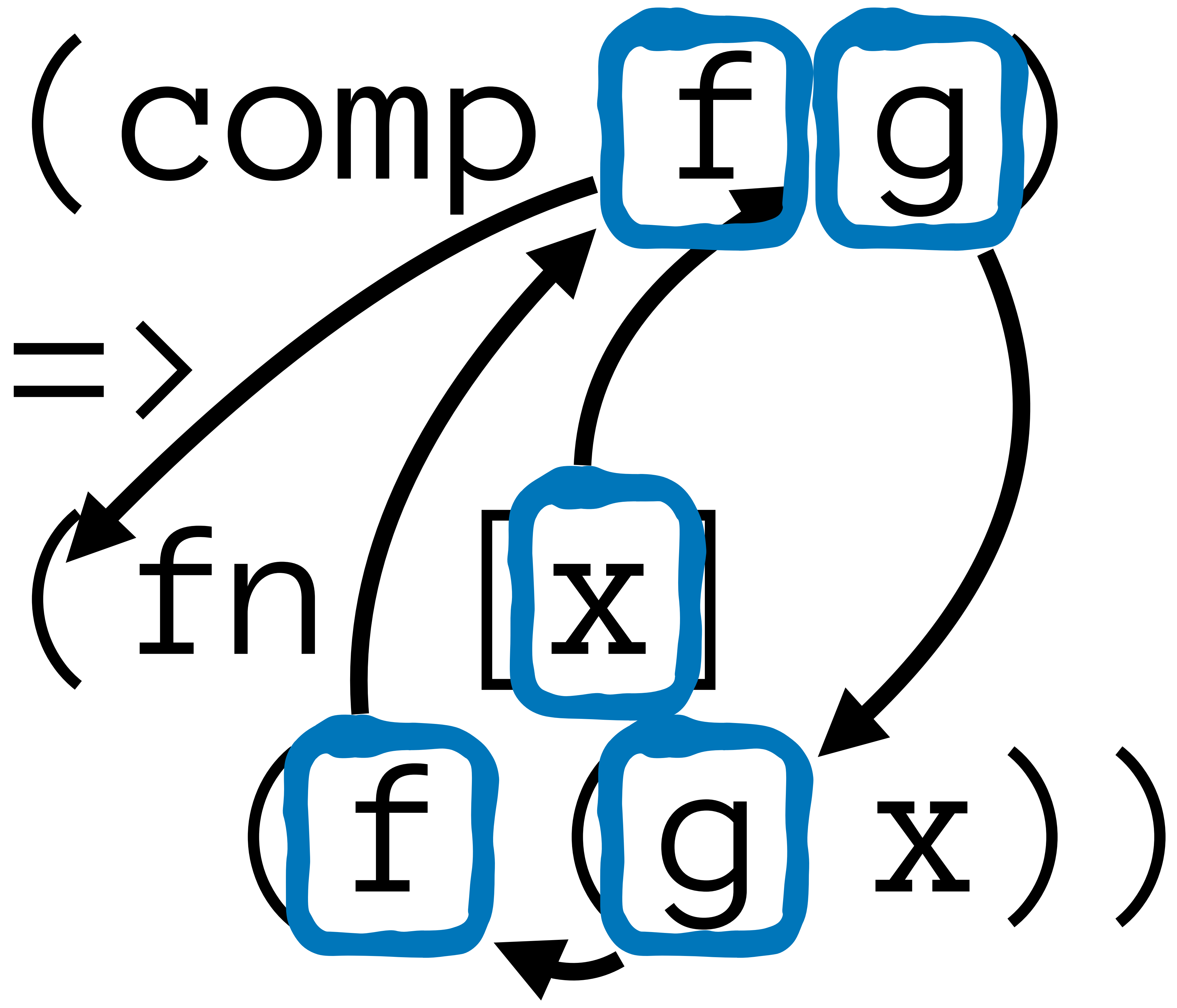
```
for all specs X,Y,
(X->Y)[X]->[Y]
```

I'll check these!

I'll write this!

comp

"Takes functions f and g, returning function applying g then f."



```
(comp f g)
=>
(fn [x]
  (f (g x)))
```

comp

"Takes functions f and g, returning function applying g then f."

/schema	`(Any->Any)(Any->Any)->(Any->Any)`
spec	`(any?->any?)(any?->any?)->(any?->any?)`
malli	`[:=> :any :any][:=> :any :any]-> [:=> :any :any]`

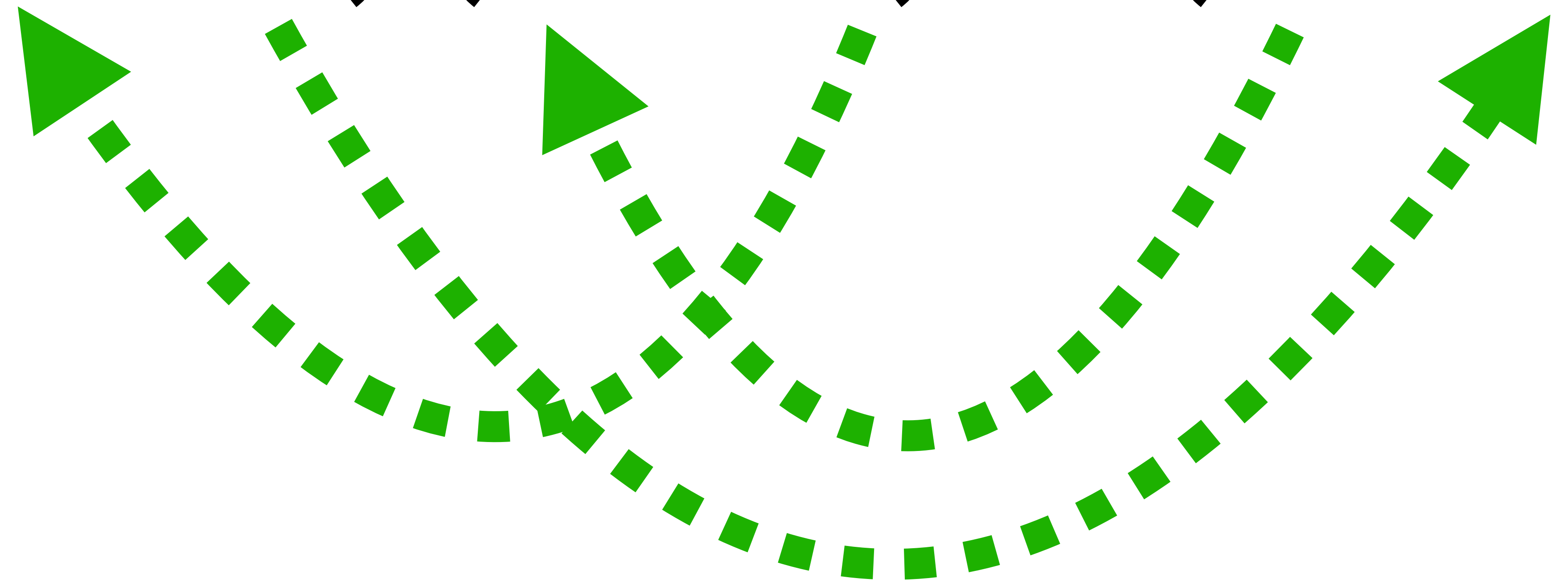
comp

"Takes functions f and g, returning function applying g then f."

/schema

```
  (Any->Any)(Any->Any)->
  (Any->Any)
(Bool->Str)(Int->Bool)->
(Int->Str)
                (2->3)(1->2)->
                (1->3)
```

```
for all specs X,Y,Z,
(Y->Z)(X->Y)->(X->Z)
```



comp

"Takes functions f and g, returning function applying g then f."

```
for all specs X,Y,Z,
(Y->Z)(X->Y)->(X->Z)
```

```
(s/def ::comp2
  (t/all :binder (t/binder
                   :a (t/bind-tv)
                   :b (t/bind-tv)
                   :c (t/bind-tv))
         :body
         (s/fspec :args (s/cat :f (s/fspec :args (s/cat :b (t/tv :b))
                                           :ret (t/tv :c))
                               :g (s/fspec :args (s/cat :a (t/tv :a))
                                           :ret (t/tv :b)))
                  :ret (s/fspec :args (s/cat :a (t/tv :a))
                                :ret (t/tv :c)))))
```

comp

"Takes functions f and g, returning function applying g then f."

```
(tu/is-valid ::comp-fspec-fn-gensym (fn [f g]
                                      #(f (g %))))
```

✓

```
(tu/is-invalid ::comp-fspec-fn-gensym (fn [f g] #(g (f %))))
```

✗

comp

"Takes functions f and g, returning function applying g then f."

```
(Any->Any)(Any->Any)->
(Any->Any)

(Bool->Str)(Int->Bool)->
(Int->Str)

(2->3)(1->2)->
(1
```

I'll check these!

```
for all specs X,Y,Z,
(Y->Z)(X->Y)->(X->Z)
```

I'll write this!

Specs for specs

↑

Spec

Leveling-Up Function Specs

https://github.com/typedclojure/typedclojure/blob/main/typed/clj.spec/README.md

https://tinyurl.com/typed-clj-spec

Now with Specs for Specs, I can help you find more mistakes!!

Thanks