The Hidden Data Flow in Types

Ambrose Bonnaire-Sergeant

Outline

Types?

Data flows?

Data flows in Types?

Data flows in Practice?

Type Systems

Static Type Checking

Specification

```
(ann app-a
  (All [a b]
    [[a -> b] ‘{:a a} -> b]))
```

Agree?

```
(defn app-a [f m]
  (f (:a m)))
```

Implementation

Type System Designs

- Hindley-Milner
- Unification-based type inference
- Let-polymorphism

(Global) Type Inference

(Local) Bidirectional Type Checking

How Global Type Inference Works

Global Type Inference (Simplified)

1. Associate type variables with each node of program

β_1 β_0 α_1 α_0

```
(defn app-a [f m] (f (:a m)))
```

$\alpha \to \beta = \beta_0$ $\alpha_0 = \{:a\ \alpha\}$ … …

2. Derive relationships (constraints) between nodes

3. Solve constraints via unification

app-a : $\alpha \to \beta, \{:a\ \alpha\} \to \beta$

How Bidirectional Type Checking Works

Bidirectional Type Checking

```
app-a : α→β,{:a α}→β
(defn app-a [f m] (f (:a m)))
```

▼ α→β,{:a α}→β

▲ α→β

▲ {:a α}

▲ α ▼ β

Infer ▲

Check ▼

What's Hard for Global Type Inference

Subtyping

Use Int as Num

Polymorphism

`Int≤Num`

`Int≤Object`

`{:a Int,:b Bool}≤{:a Int}`

Constraints + Subtyping

$\beta_0 = \beta$

Unification?

$\beta_0 \leq \beta$

Unification?

How to solve?

What's Hard for Bidirectional Type Checking

Bidirectional Inference

```
          ?
(map (fn [x] x)
     [1 2 3])
:(List ?)
```

Typed Racket

```
                Any
(map (fn [x] x)
     [1 2 3])
:(List Any)
```

TypeScript

```
             any
(map (fn [x] x)
     [1 2 3])
:(List any)
```

Gradual Typing

```
               Dyn
(map (fn [x] x)
     [1 2 3])
:(List Dyn)
```

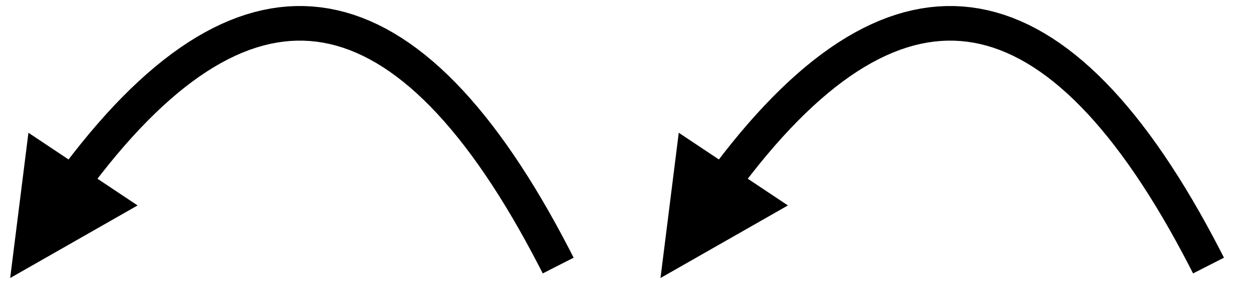
The Limitation

```
(map (fn [x] x)
     [1 2 3])
```

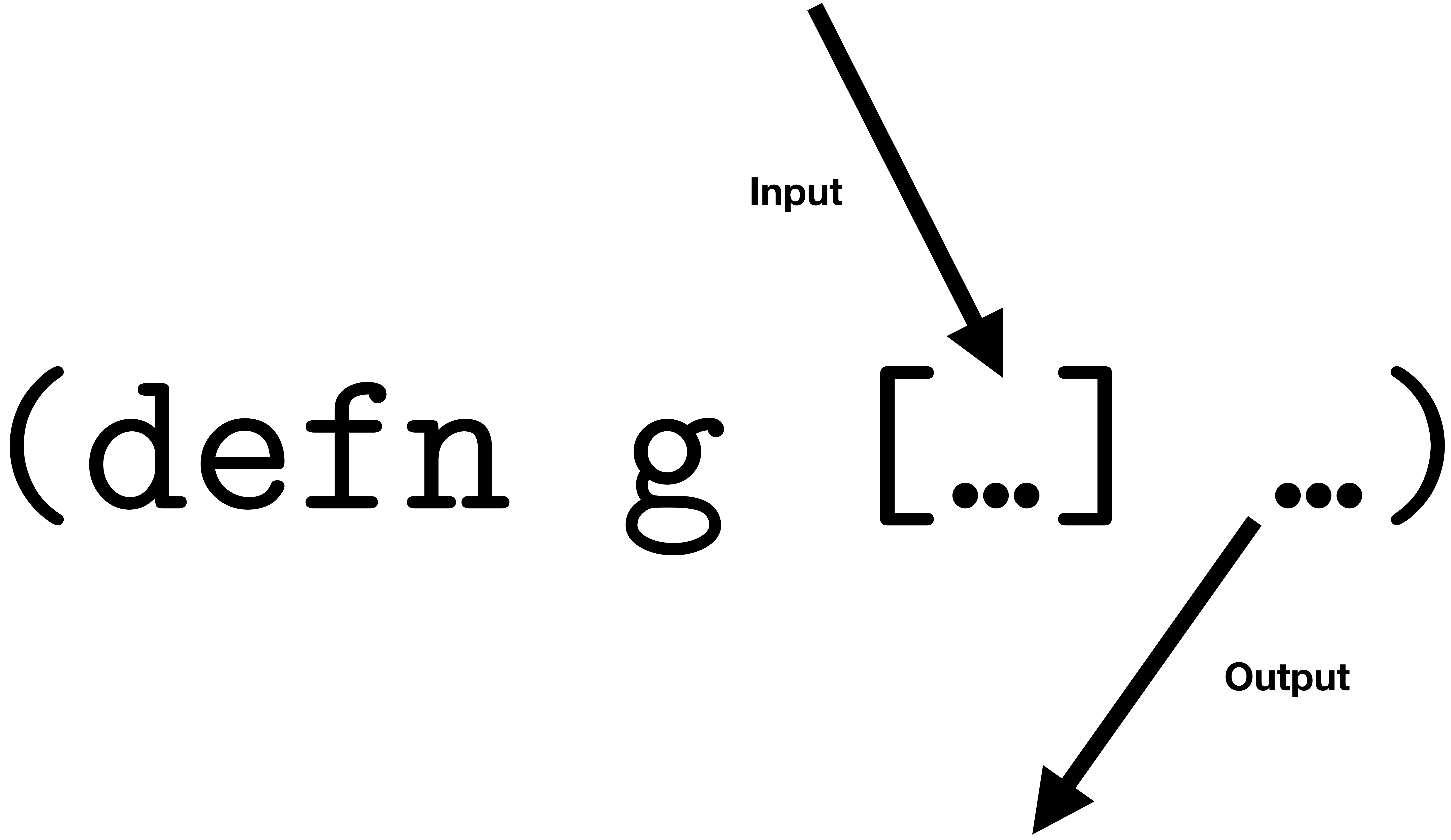
map : ∀α , β . α→β , α*→β*

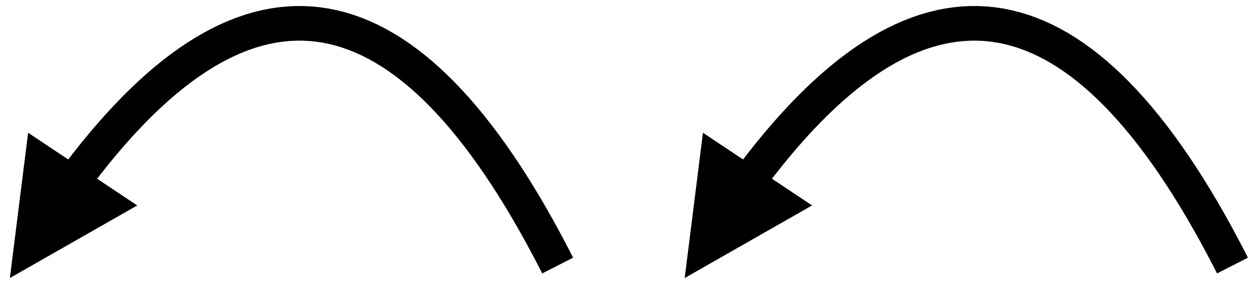
How to interleave?

Data flow



```
(f (g 42))
```


(defn g […] …)
Input
Output



```
(f (g 42))
```

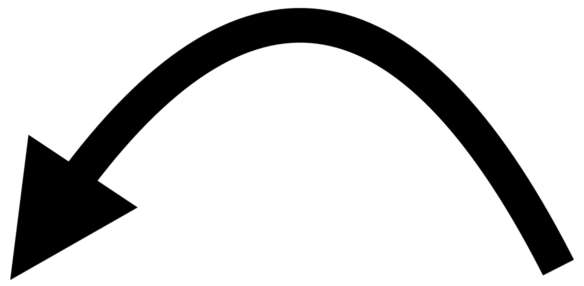
Input

```
(defn f […] …)
```

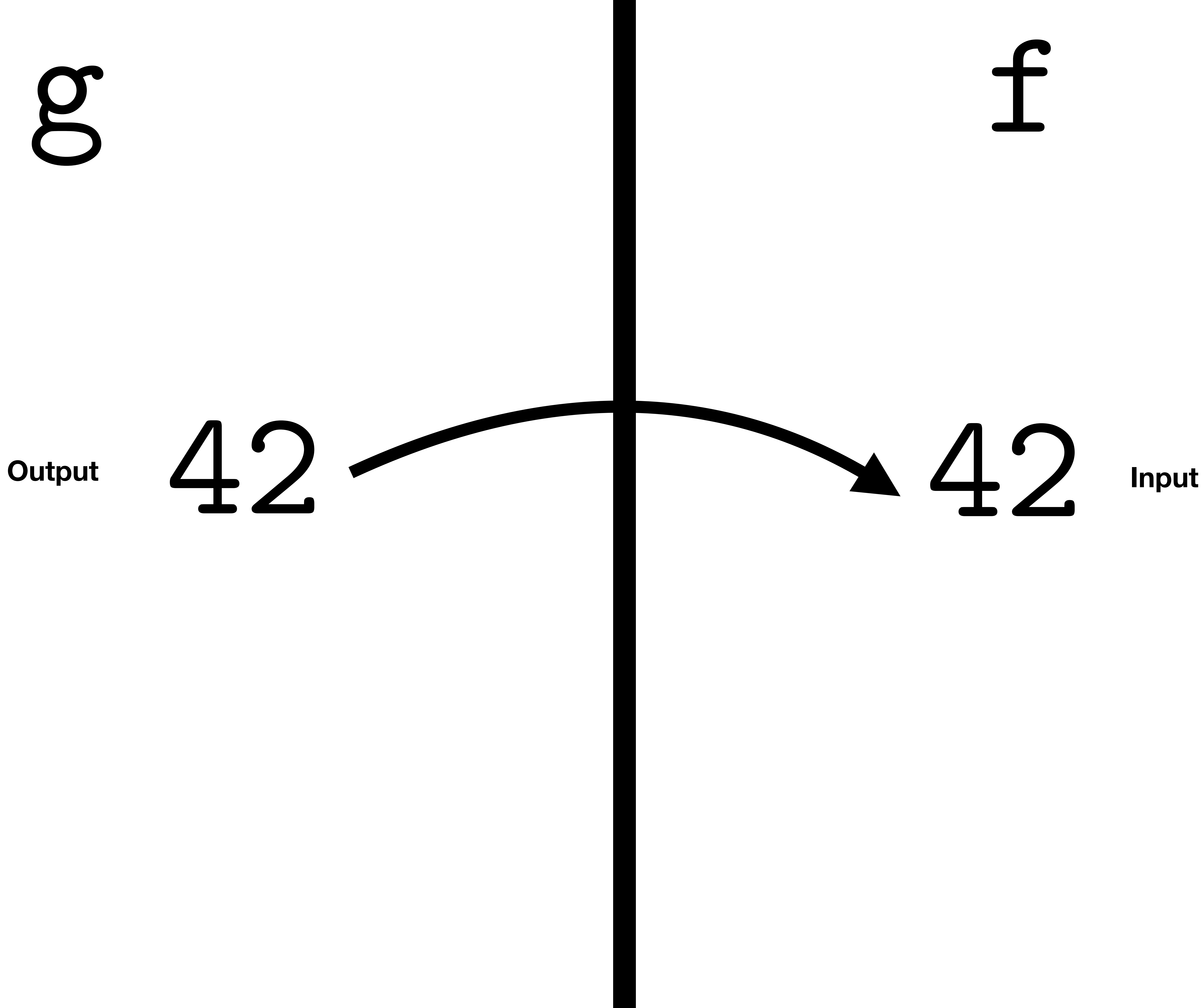
Output

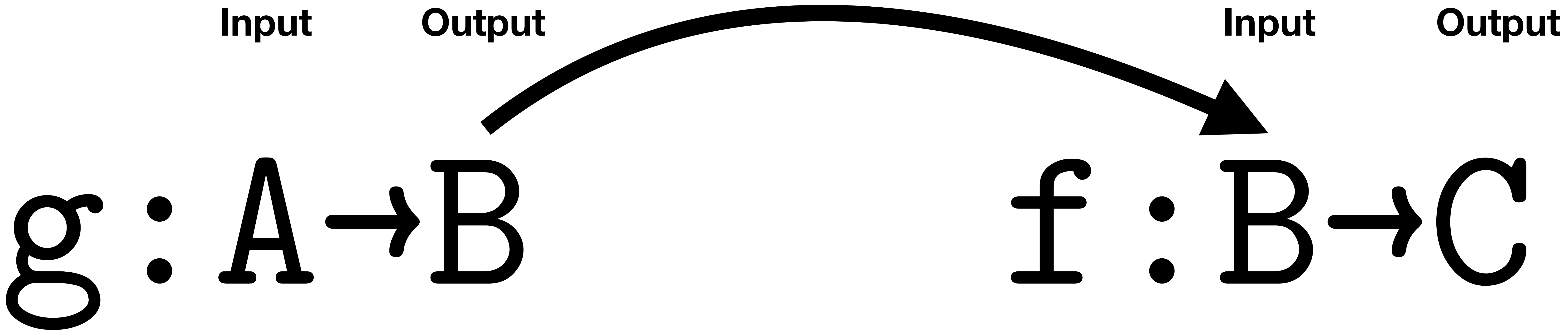
`f g`

Input **Output**



`(f (g 42))`

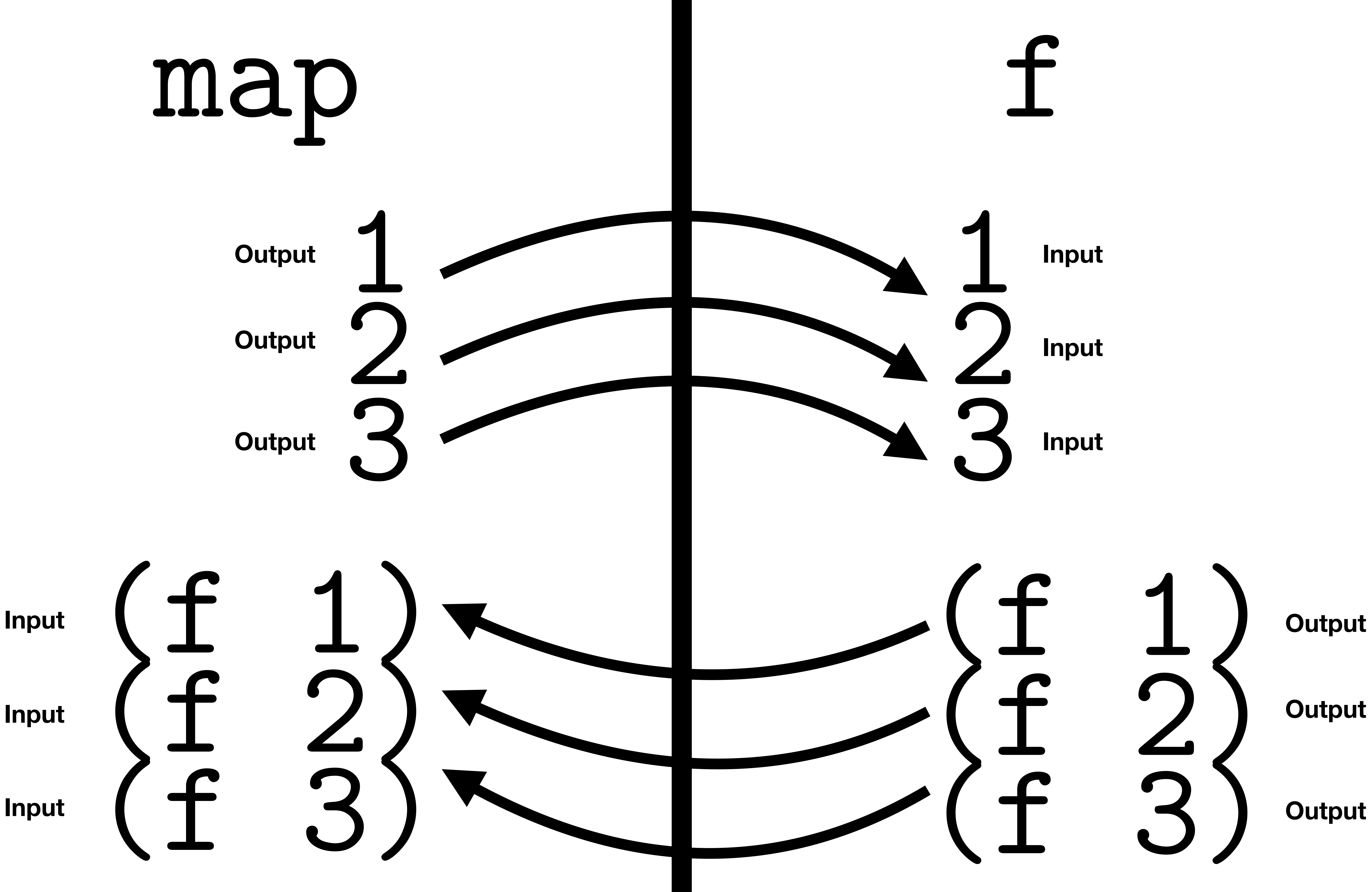

g
f
Output
42
42
Input


Input
Output
Input
Output
g:A→B
f:B→C

Higher-order Data flow

```
(map f [1 2 3])
```

Input Input



map
f
Output 1
Output 2
Output 3
1 Input
2 Input
3 Input
Input (f 1)
Input (f 2)
Input (f 3)
(f 1) Output
(f 2) Output
(f 3) Output

Input Output

```
map:(A→B),A*→B*
```

? ?

Input Output

$$A \to B$$

Output Input Output

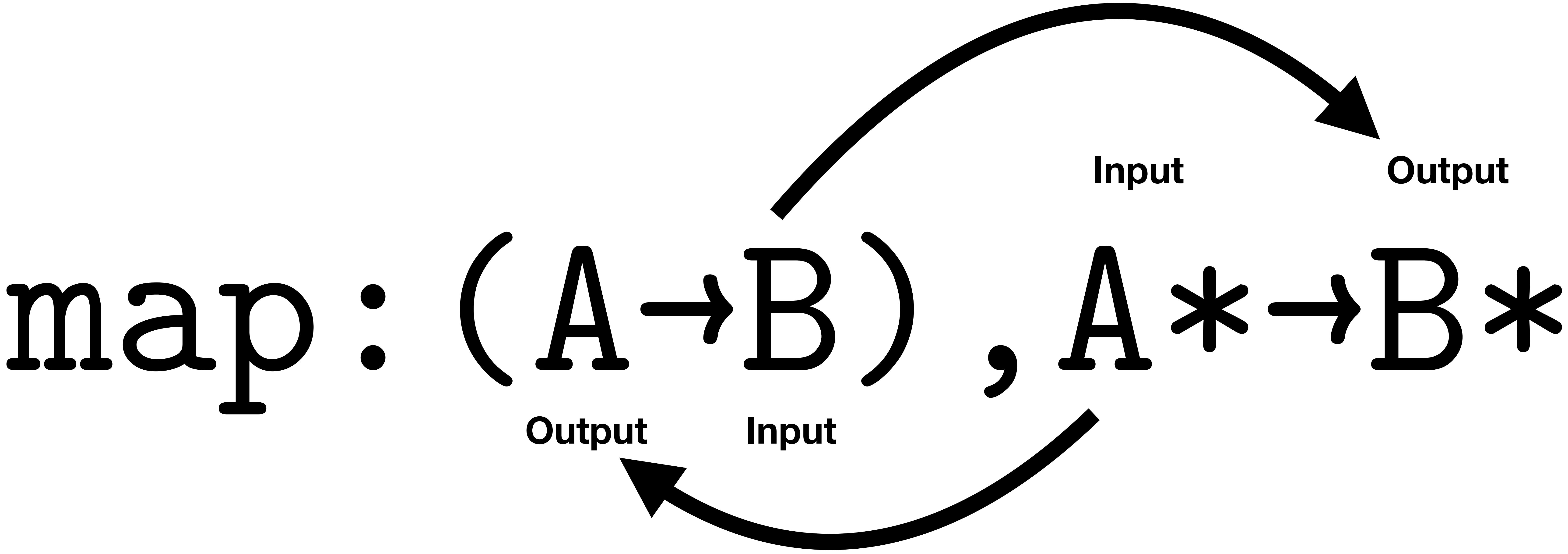
$$(A \to B) \to C$$

Input Output Input Output

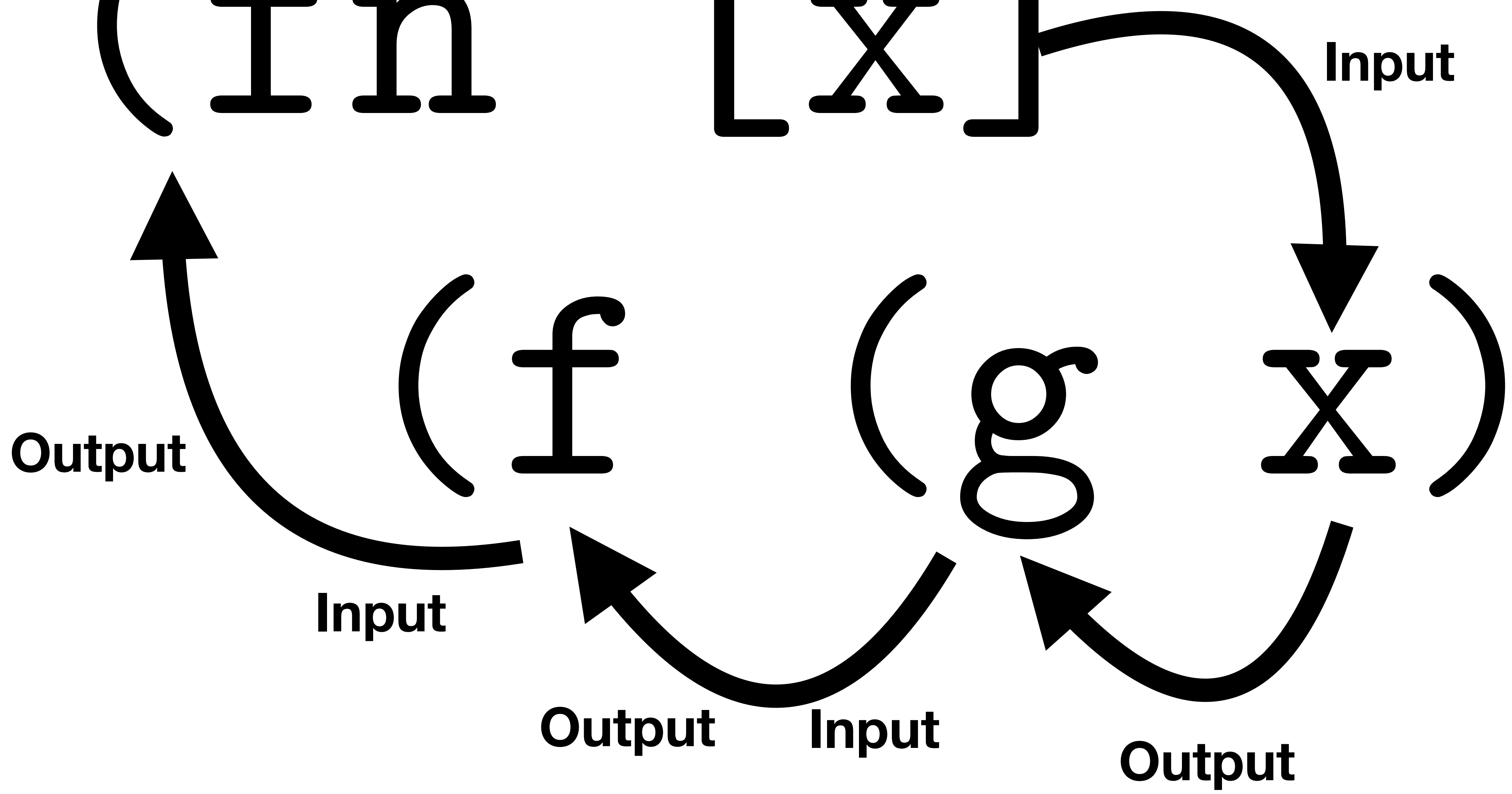
$$((A \to B) \to C) \to D$$

Output Input Output Input Output

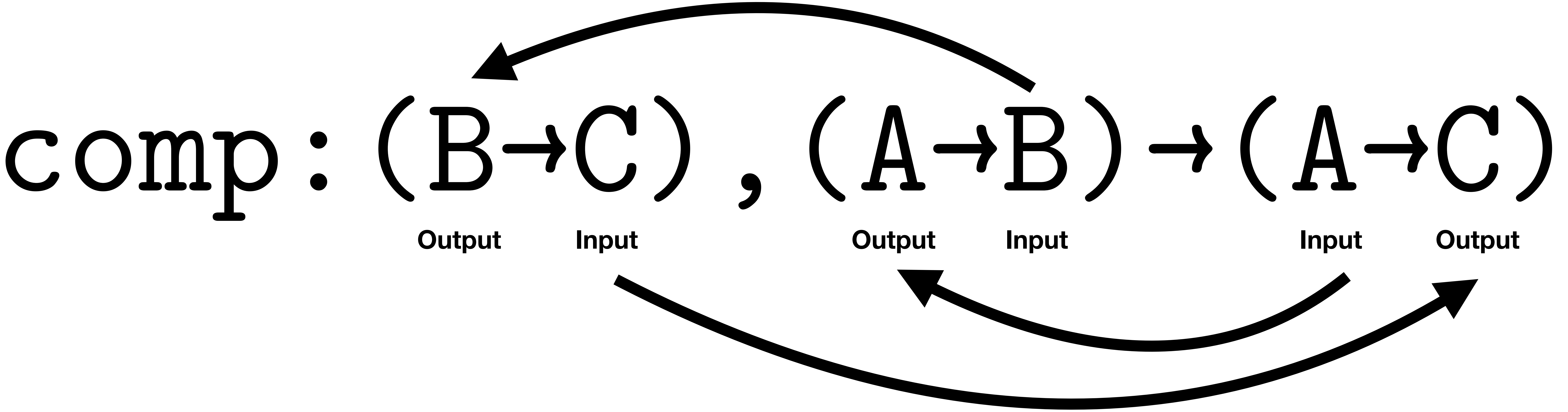
$$(((A \to B) \to C) \to D) \to E$$


Input
Output
map:(A→B),A*→B*
Output
Input

```
(defn comp [f g]
  (fn [x]
    (f (g x))))
```



comp
f
g
Output
x
x
Input
Input
(g x)
(g x)
Output
Output
(g x)
(g x)
Input
(f(g x))
(f(g x))
Input
Output


comp:(B→C),(A→B)→(A→C)
Output
Input
Output
Input
Input
Output

Data Flows helping Global Type Inference

MLsub = HM + Subtyping

Unification => Biunification

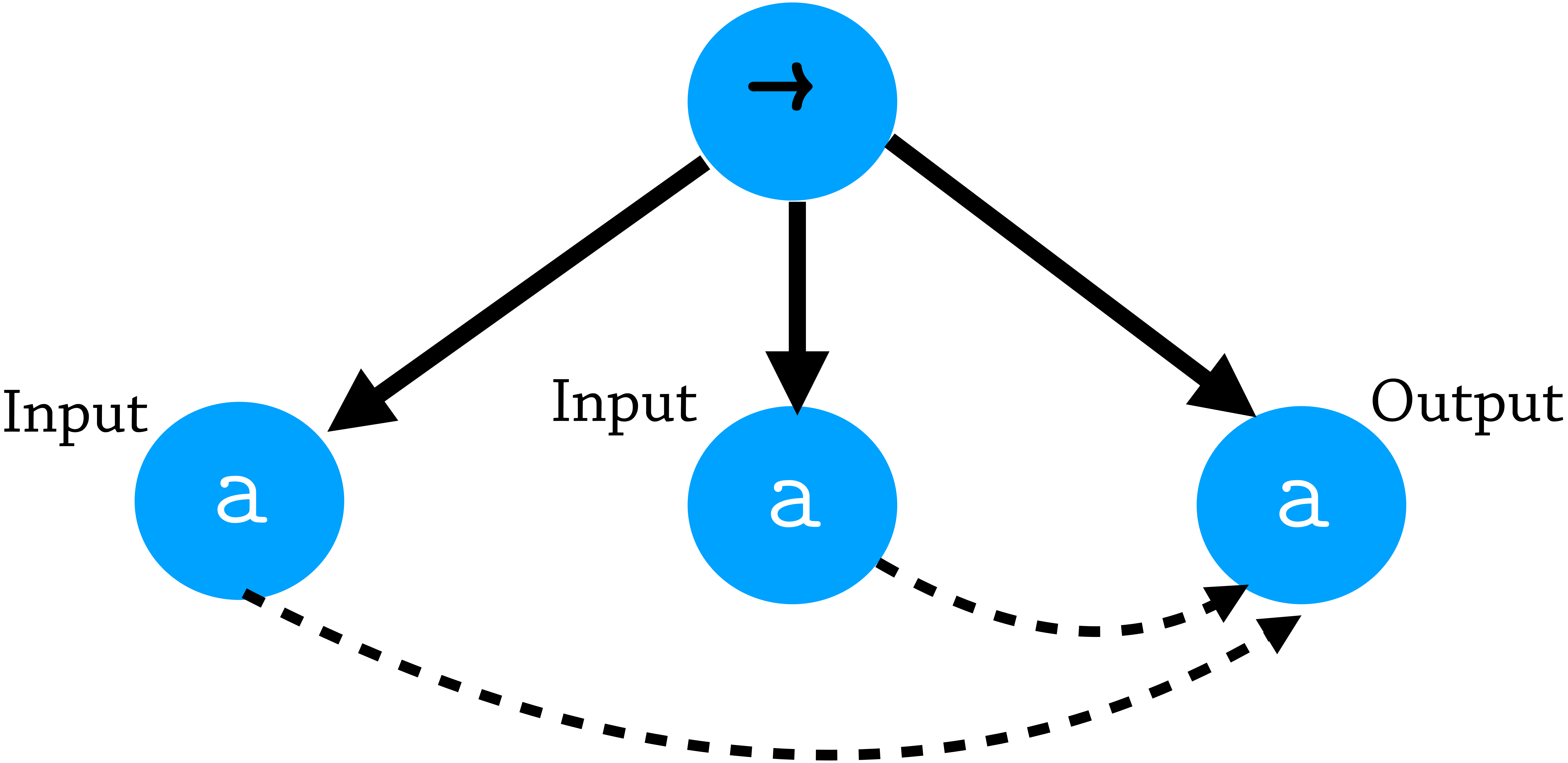
Substitutions => Bisubstitutions

Types => Finite State Machines

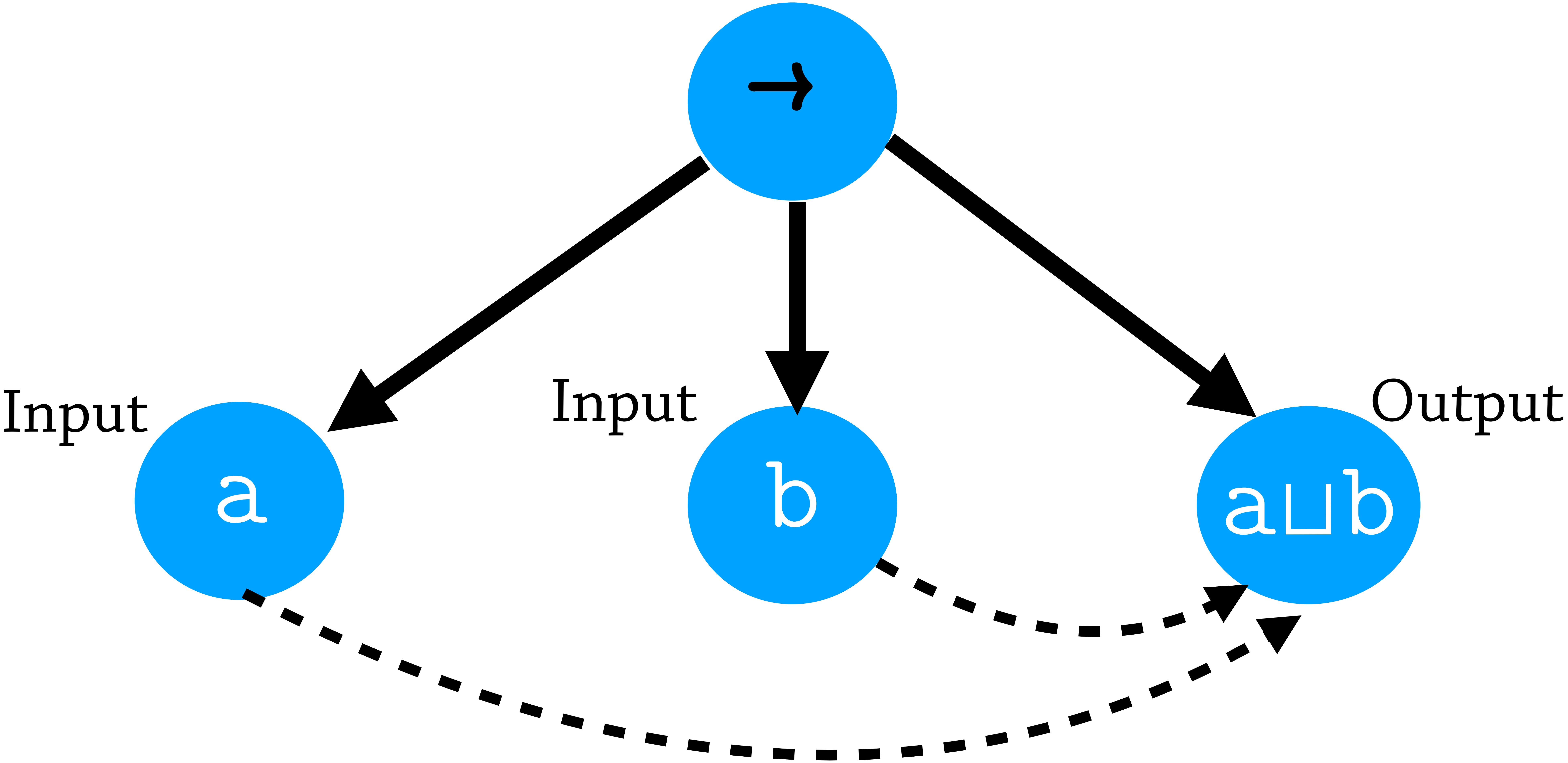
```
choose : a,a → a
choose : a,b → a⊔b
```

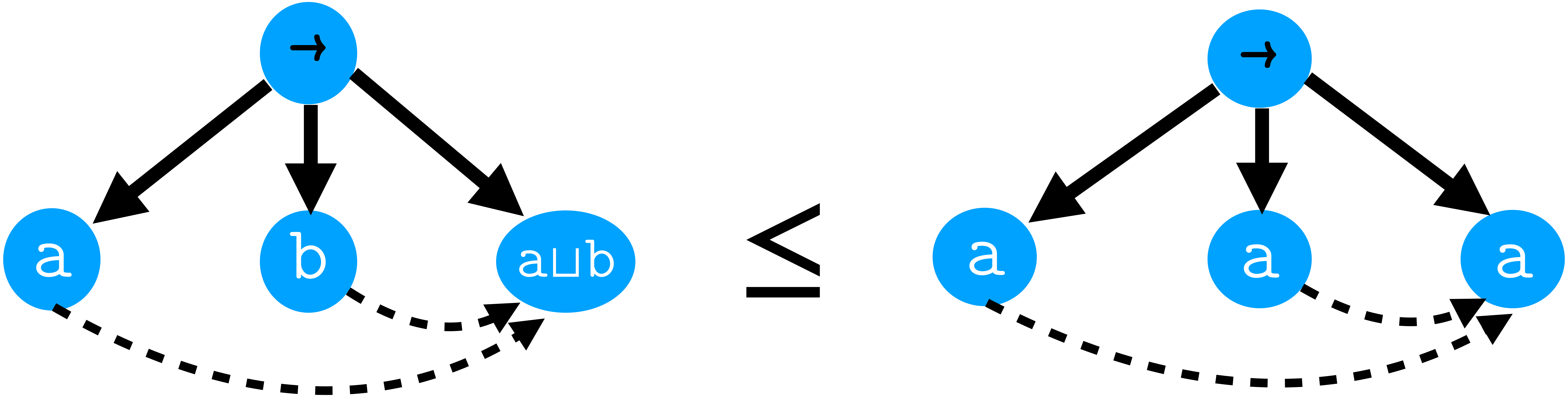
Returns either first or second argument.

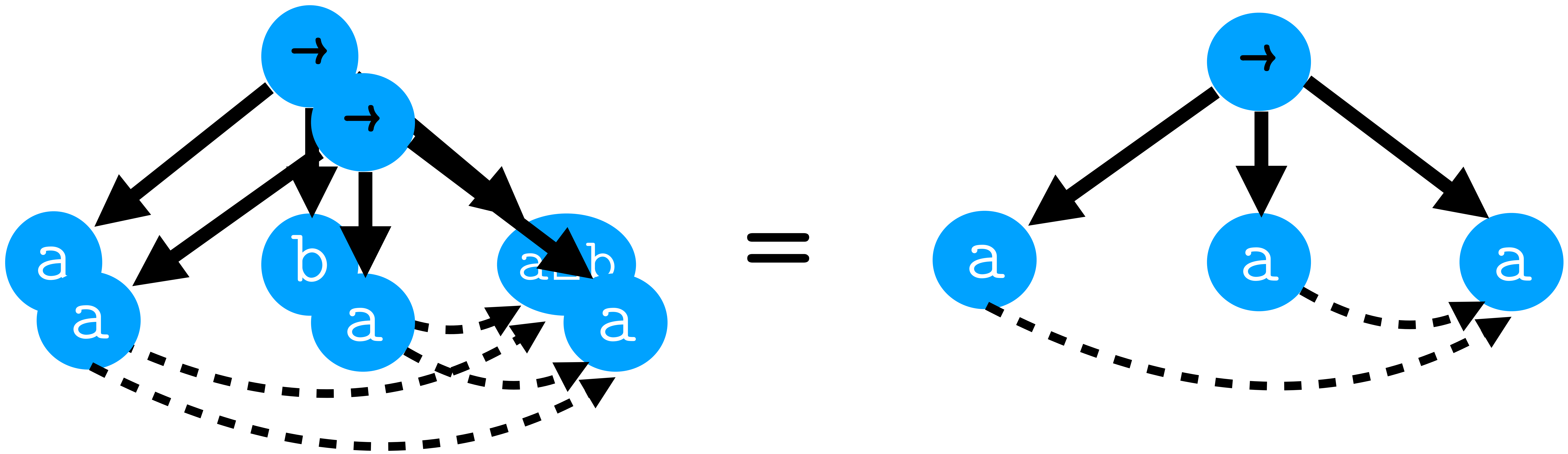
choose : a,a → a



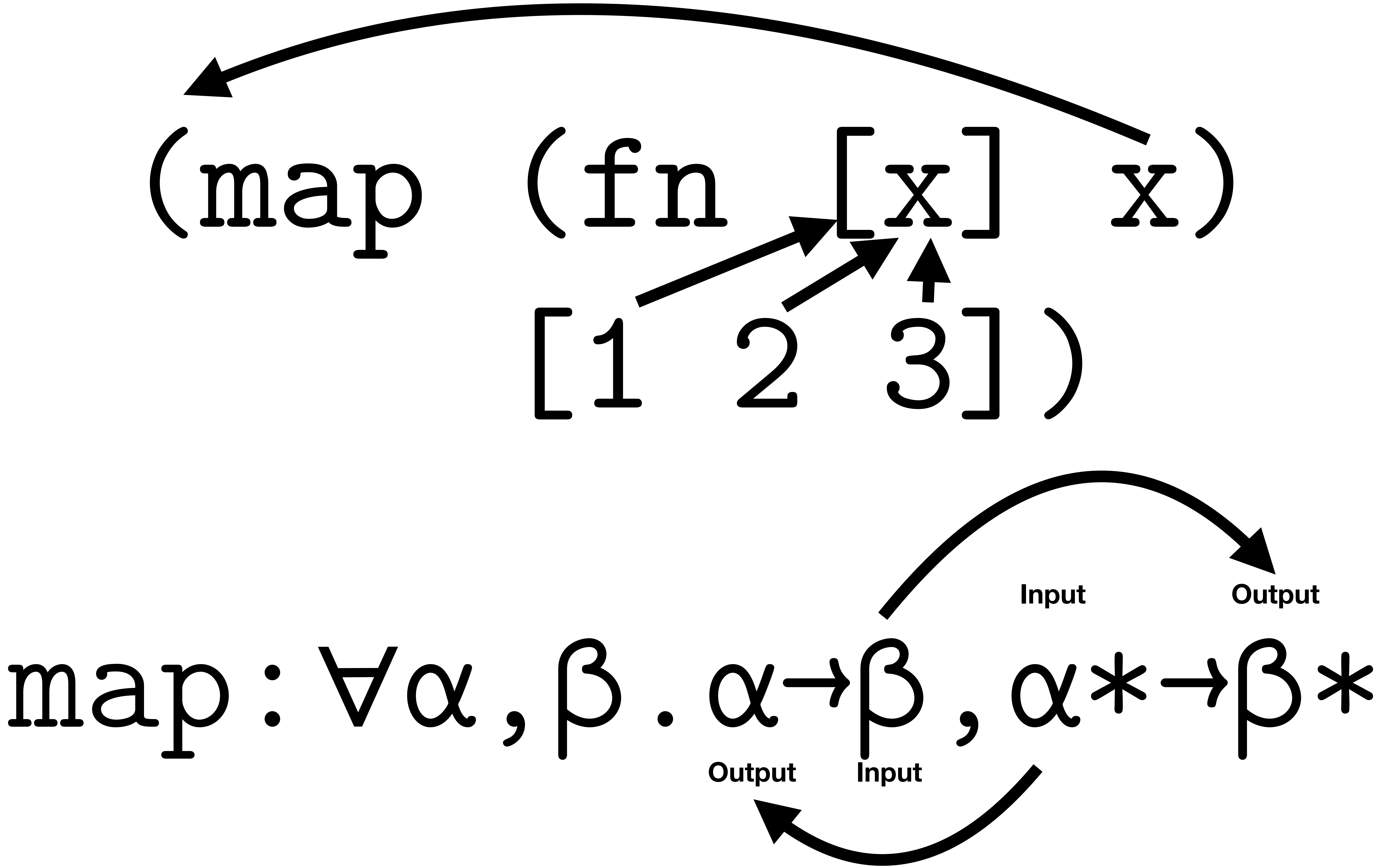
choose : a,b → a⊔b




→
a
b
a⊔b
≤
→
a
a
a


→
→
a
a
b
a
a
b
a
=
→
a
a
a

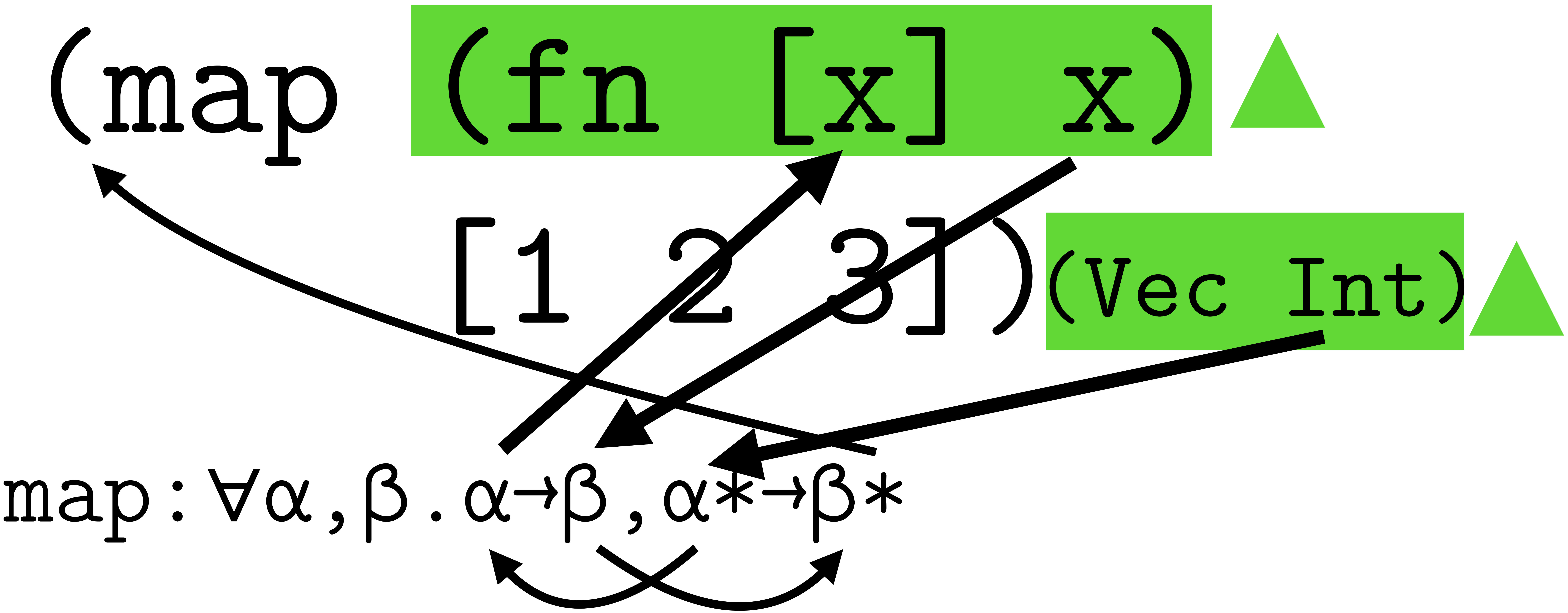
Data Flows helping Bidirectional Type Checking


(map (fn [x] x)
[1 2 3])
Input
Output
map:∀α,β.α→β,α*→β*
Output
Input

The Limitation

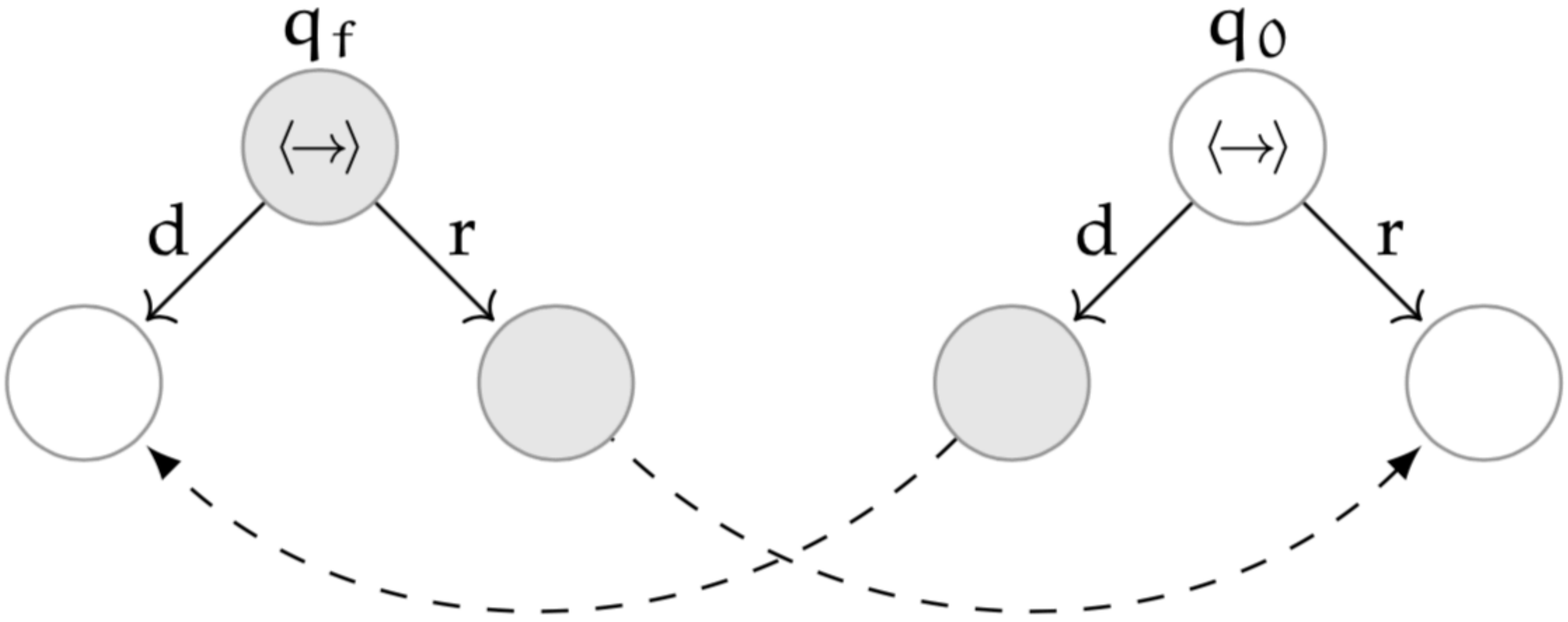
```
(map (fn [x] x)
     [1 2 3])
```

map : ∀α, β. α→β, α*→β*


(map (fn [x] x)
[1 2 3])
(Vec Int)
map:∀α,β.α→β,α*→β*

References

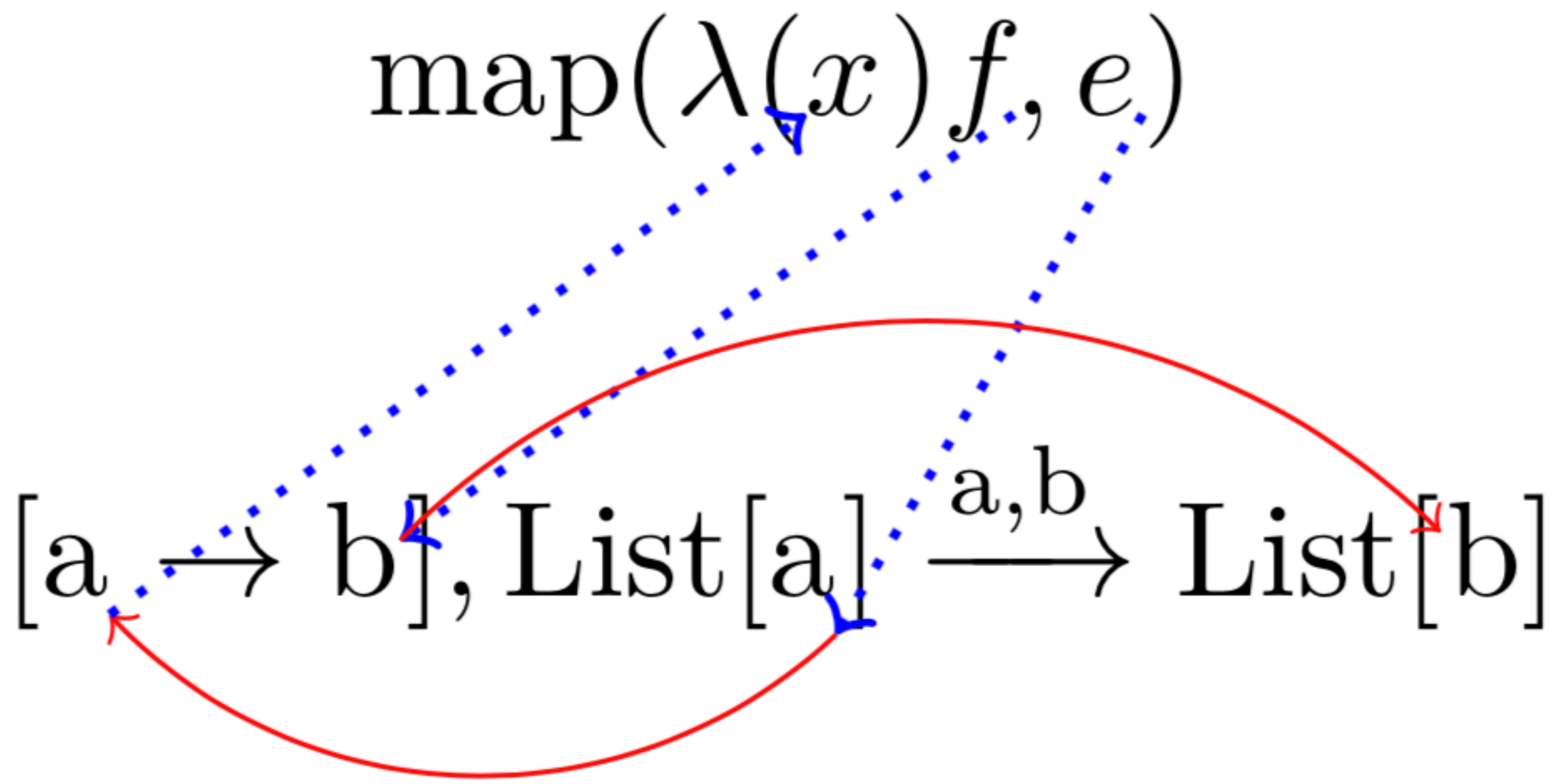
Global Type Inference + Subtyping



(a) $[f : \alpha \to \beta]\alpha \to \beta$

Algebraic Subtyping, Stephen Dolan, PhD Thesis (2016)

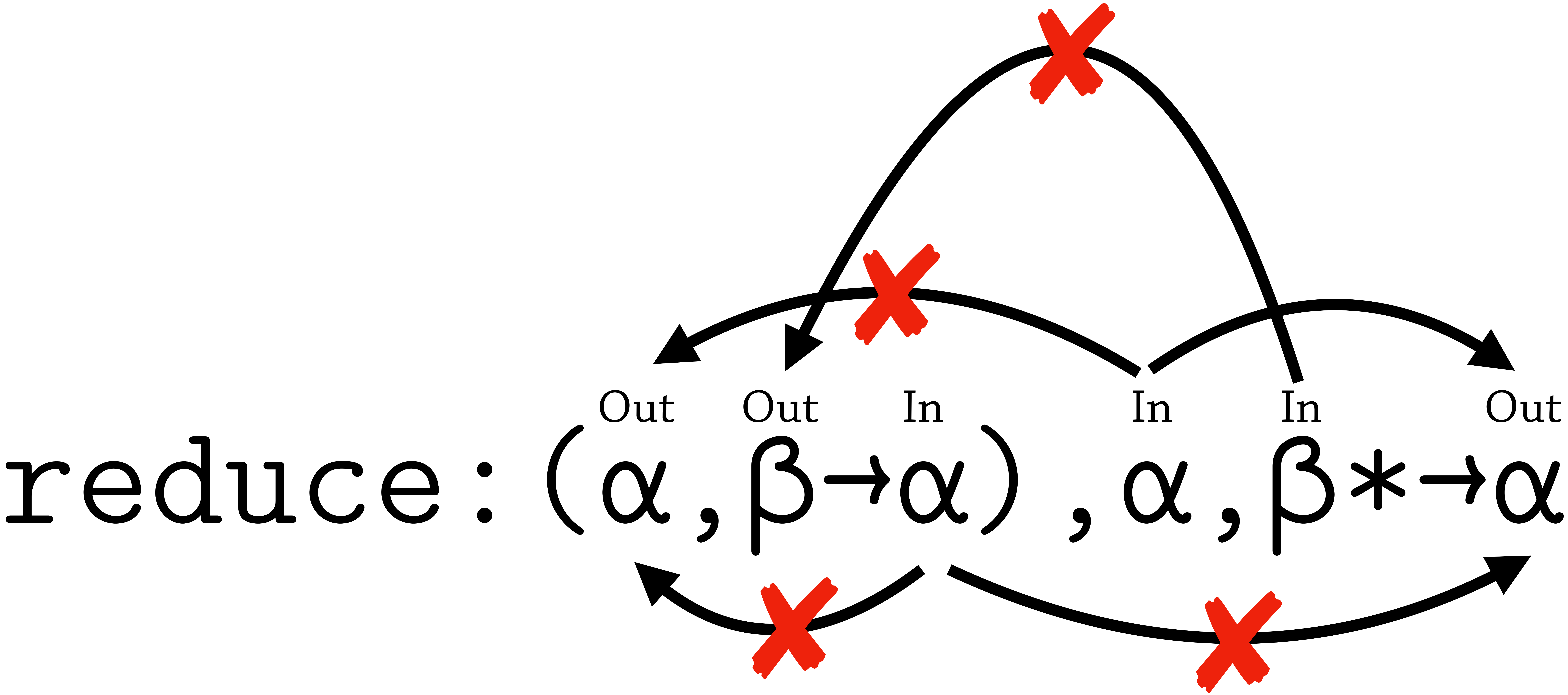
Bidirectional Checking + Inference



Typed Clojure in Theory and Practice,
Ambrose Bonnaire-Sergeant, PhD Thesis (2019)

Conclusion

$$\texttt{map}: \alpha \to \beta, \alpha^* \to \beta^*$$



Thank you!

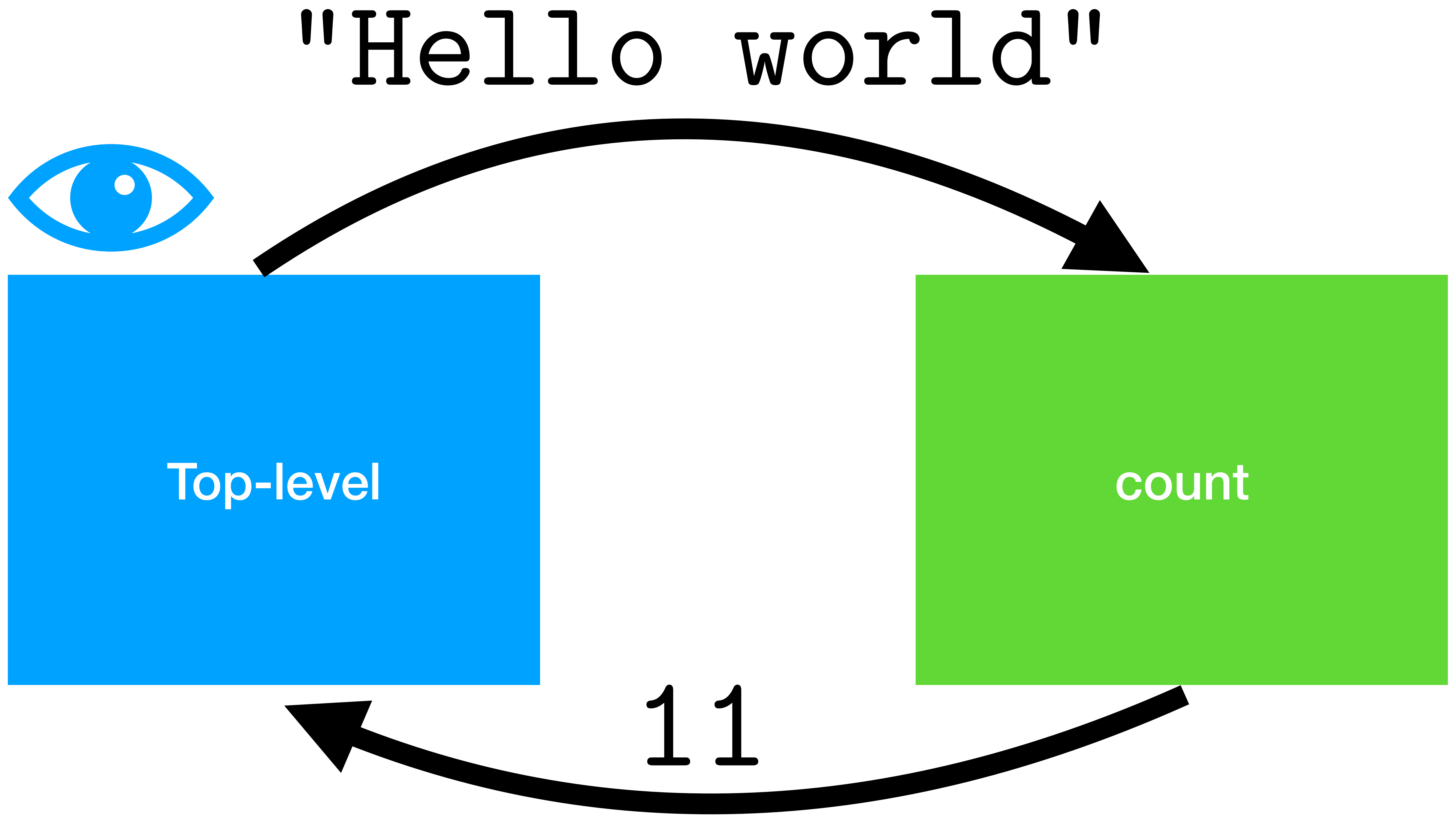
@ambrosebs

Extra slides

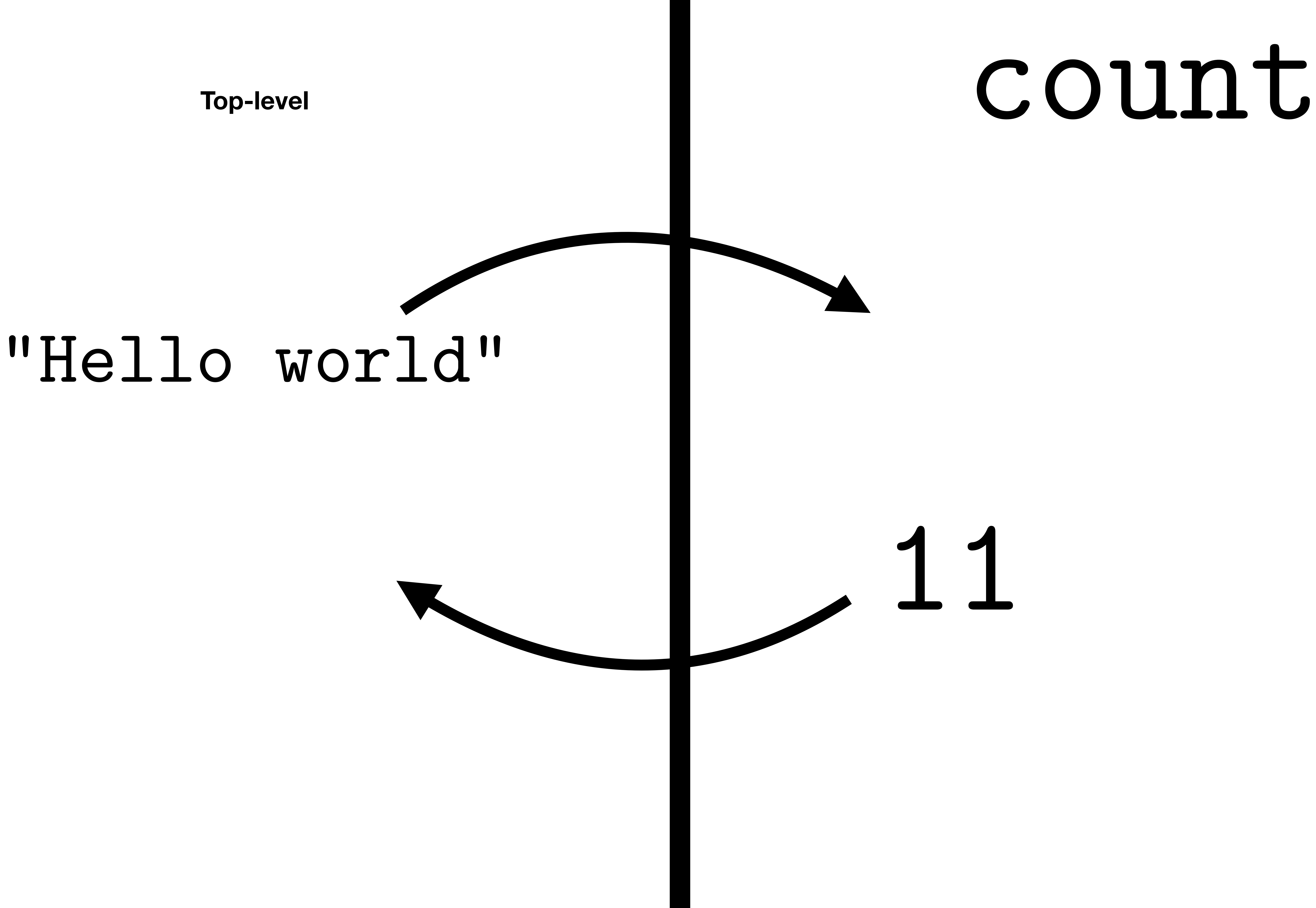
```
first : a → b → a
first : a → ⊤ → a
```

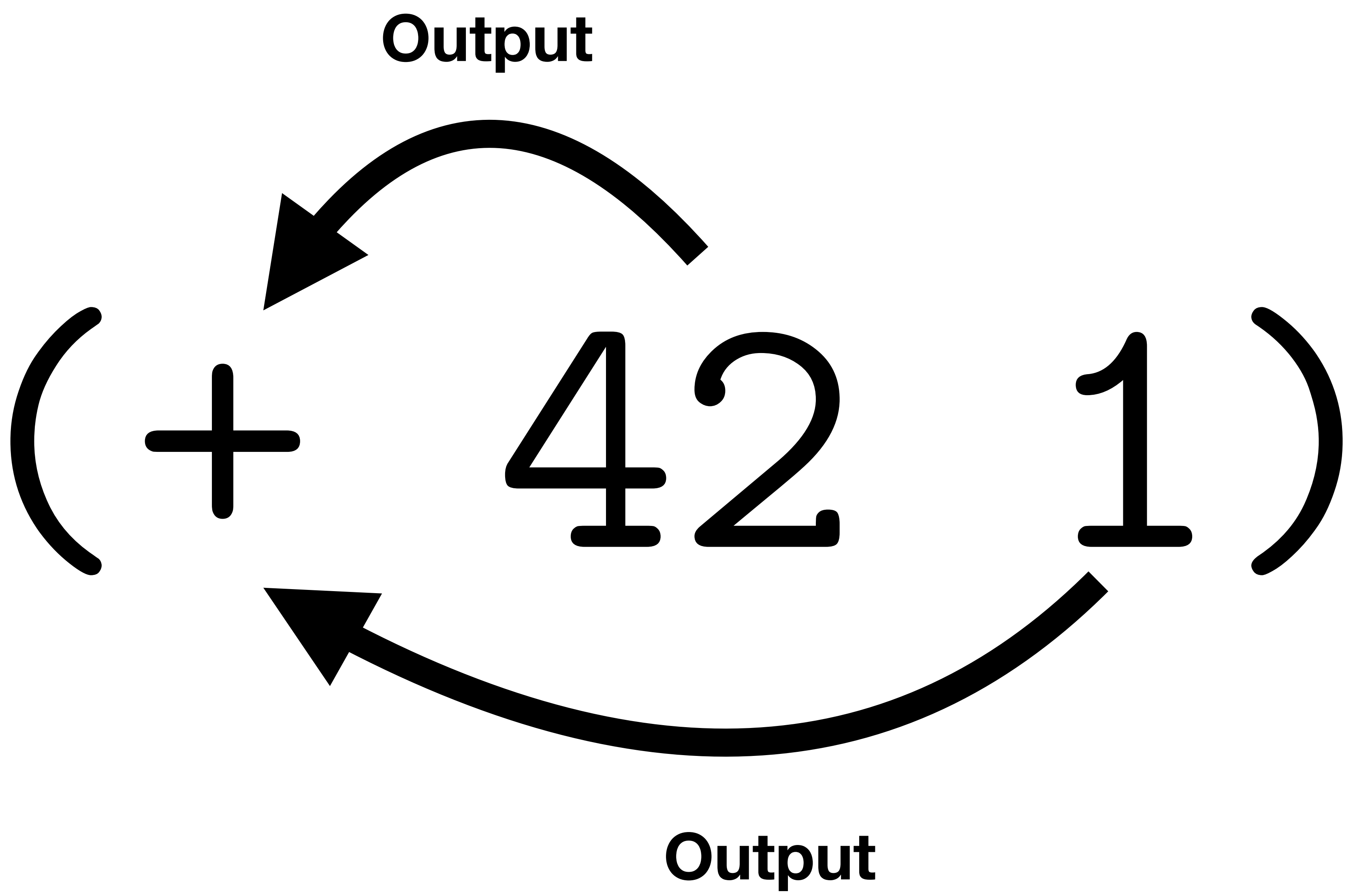
```
choose : a → a → a
choose : a⊔b → a⊔b → a⊔b
choose : a → a⊔b → a⊔b
choose : a → b → a⊔b
```

```
(count "Hello world")
;=> 11
```



```
(count "Hello world")
```

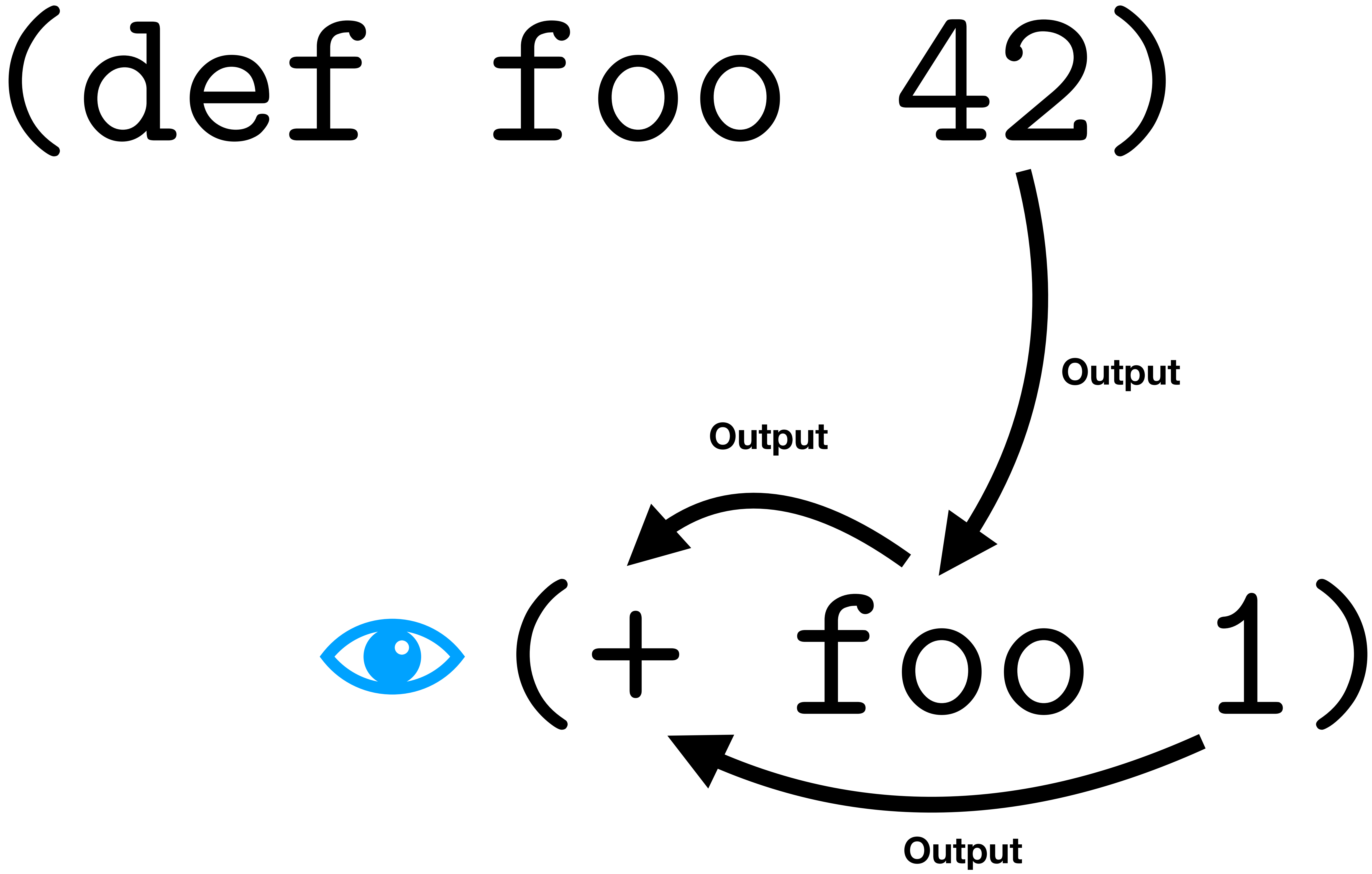

Top-level
count
"Hello world"
11

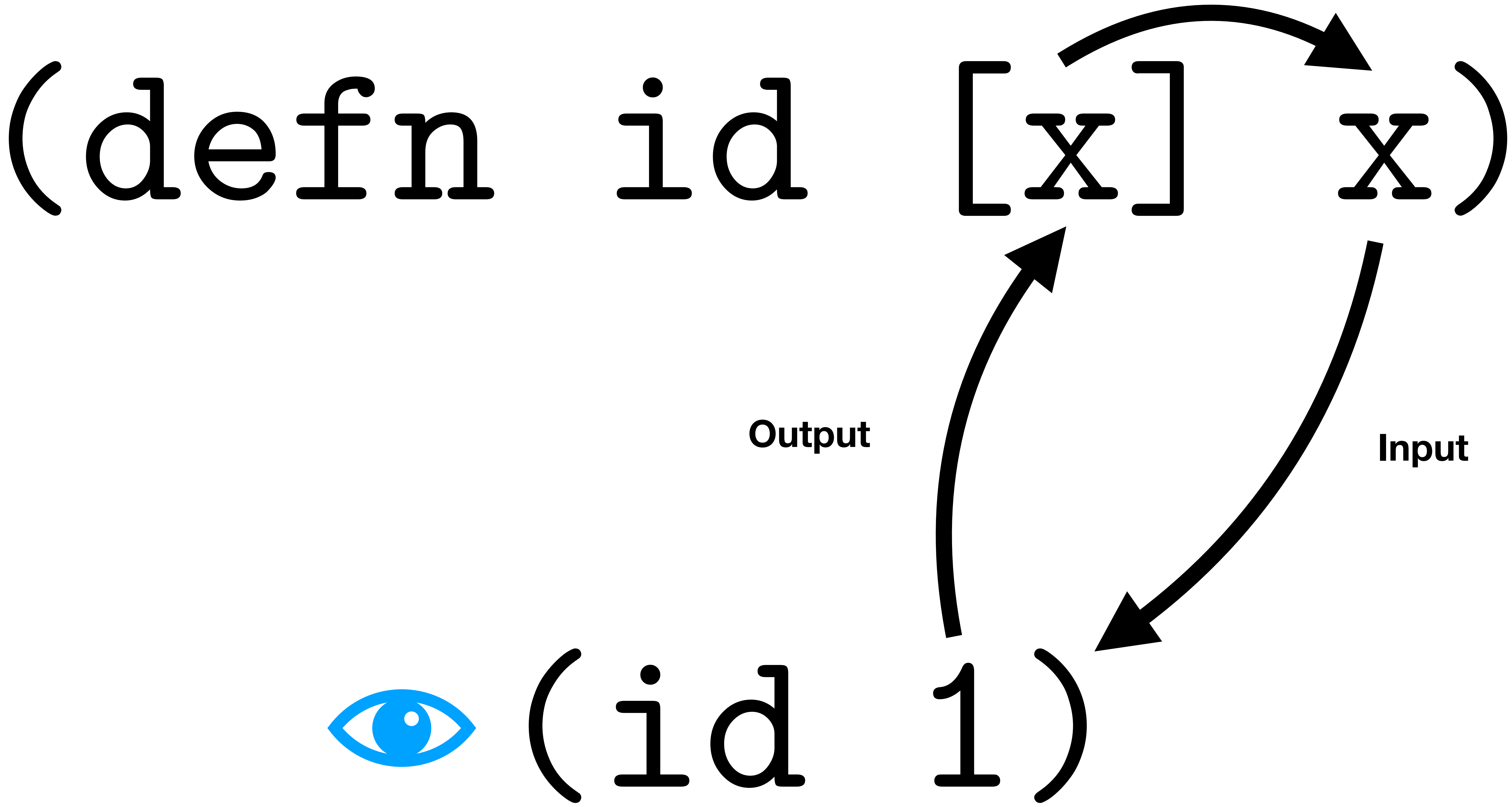

Output
(+ 42 1)
Output

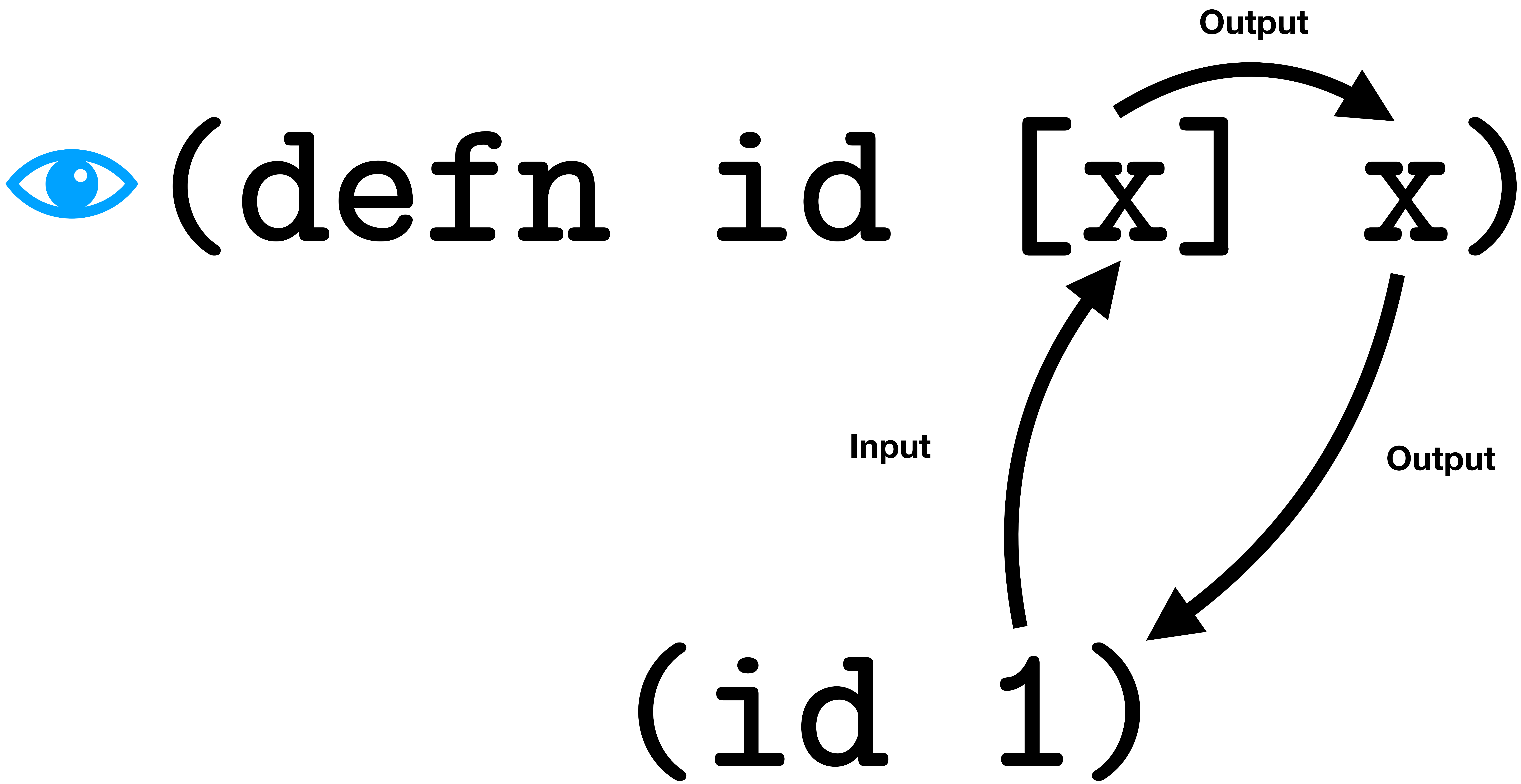
```
(def foo 42)
```

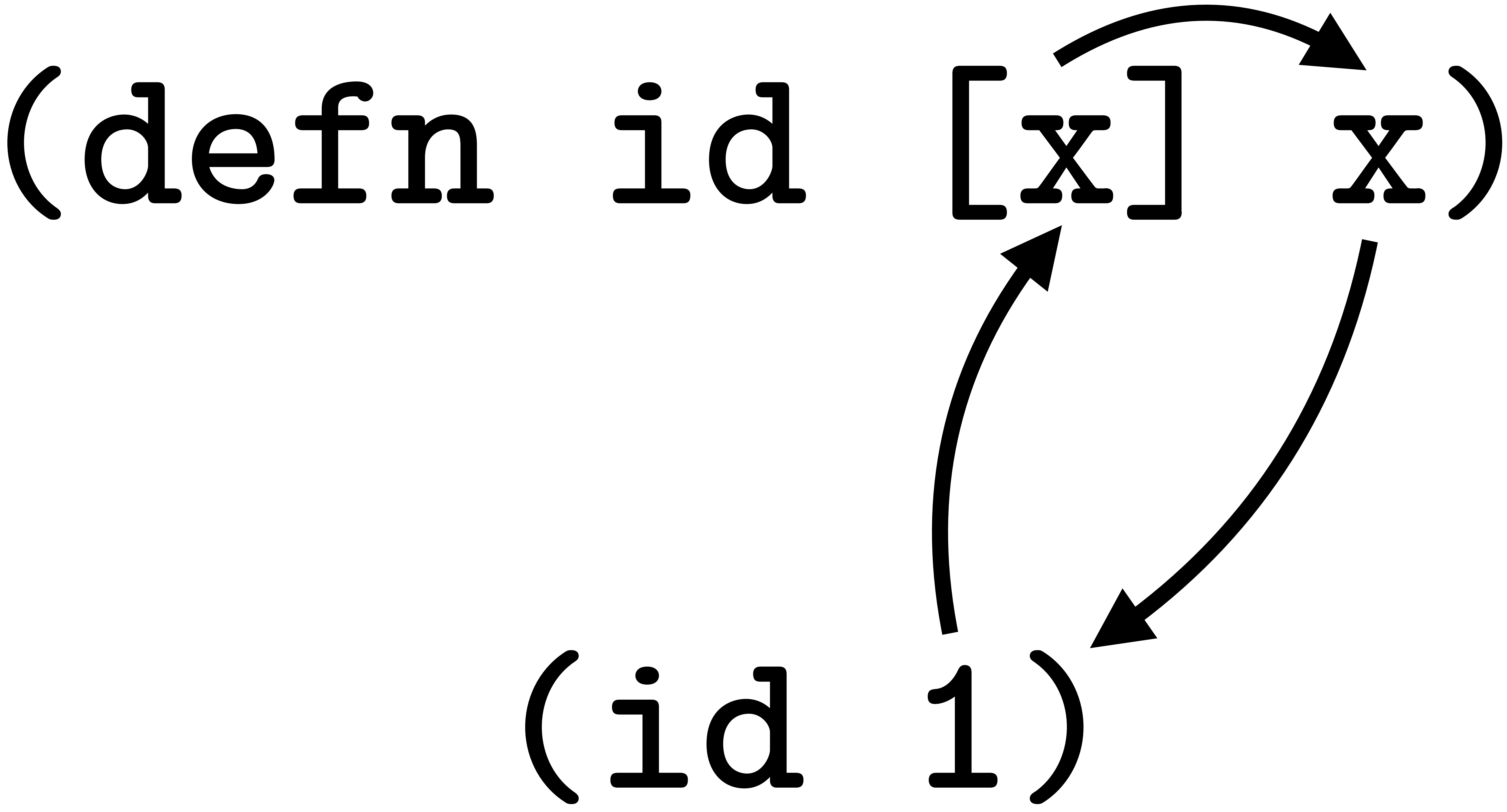
Output

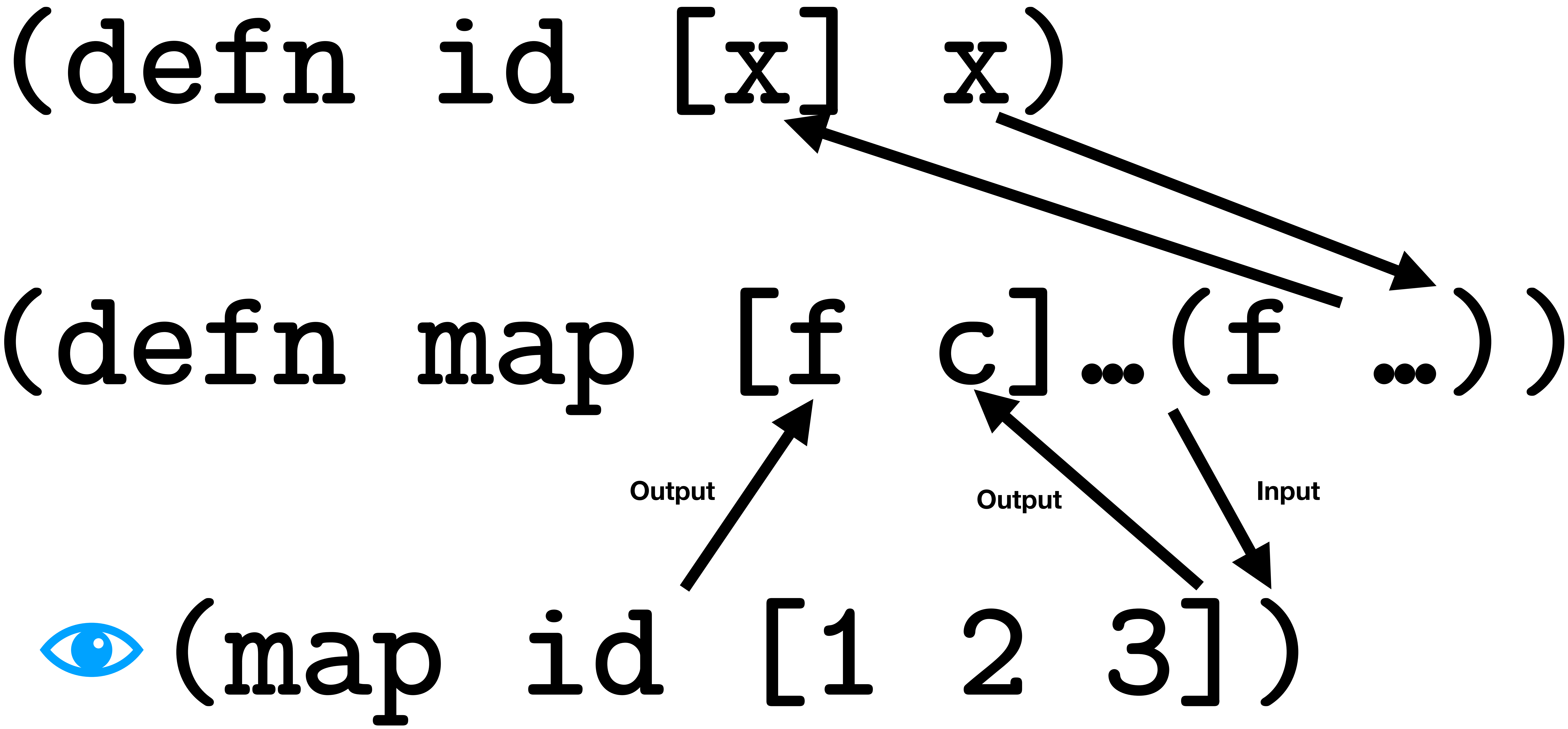
```
(+ foo 1)
```


(def foo 42)
Output
Output
(+ foo 1)
Output


(defn id [x] x)
Output
Input
(id 1)


(defn id [x] x)
Output
Input
Output
(id 1)


(defn id [x] x)
(id 1)


(defn id [x] x)
(defn map [f c]…(f …))
(map id [1 2 3])
Output
Output
Input

Top-level
map
id
id
[1 2 3]
id
1
2
3


(defn id [x] x)
Input
Output
(defn map [f c]…(f …))
Input
Input
Output
(map id [1 2 3])


(defn id [x] x)
Output
Input
(defn map [f c]…(f …)))
Output
(map id [1 2 3])