Typed Clojure

INDIANA UNIVERSITY

Space-Efficient Runtime Tracking

Inferring type annotations via runtime observations

Ambrose Bonnaire-Sergeant

Type annotations

```
1  +(ann comma-sep [(Seqable Expr) -> (Seqable (U Expr String))])
2   (defn comma-sep [xs]
3     (interpose "," xs))
4
5  +(defalias Expr
6  +  (U '{:op ':list :items (Vec Expr)}
7  +     '{:op ':val :val Any}
8  +     ...))
9  +
10 +(ann emit-list ['{:op ':list :items (Vec Expr)} -> String])
11  (defn emit-list [{:keys [items]}]
12    (if (empty? items)
```

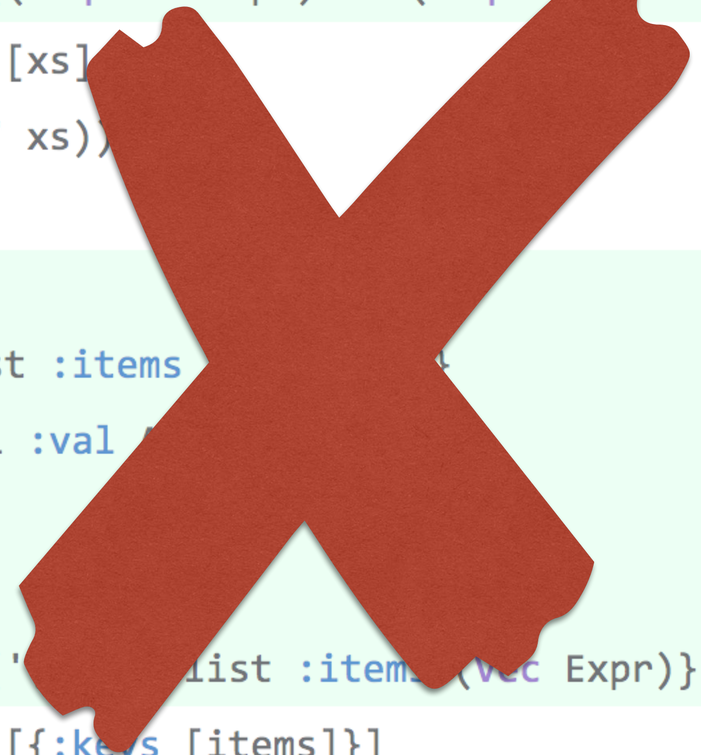
Problem:

```
1  +(ann comma-sep [(Seqable Expr) -> (Seqabl  U Expr String))])
2   (defn comma-sep [xs]
3     (interpose "," xs))
4
5  +(defalias Expr
6  +  (U '{:op ':list :items
7  +      '{:op ':val :val
8  +      ...))
9  +
10 +(ann emit-list ['      list :items (vec Expr)} -> String])
11  (defn emit-list [{:keys [items]}]
12    (if (empty? items)
```

Problem:

Someone wrote this manually

```
1  +(ann comma-sep [(Seqable Expr) -> (Seqabl  U Expr String))])
2   (defn comma-sep [xs]
3     (interpose "," xs))
4
5  +(defalias Expr
6  +  (U '{:op ':list :items
7  +      '{:op ':val :val
8  +      ...))
9  +
10 +(ann emit-list ['  list :item  (vec Expr)} -> String])
11  (defn emit-list [{:keys [items]}]
12    (if (empty? items)
```

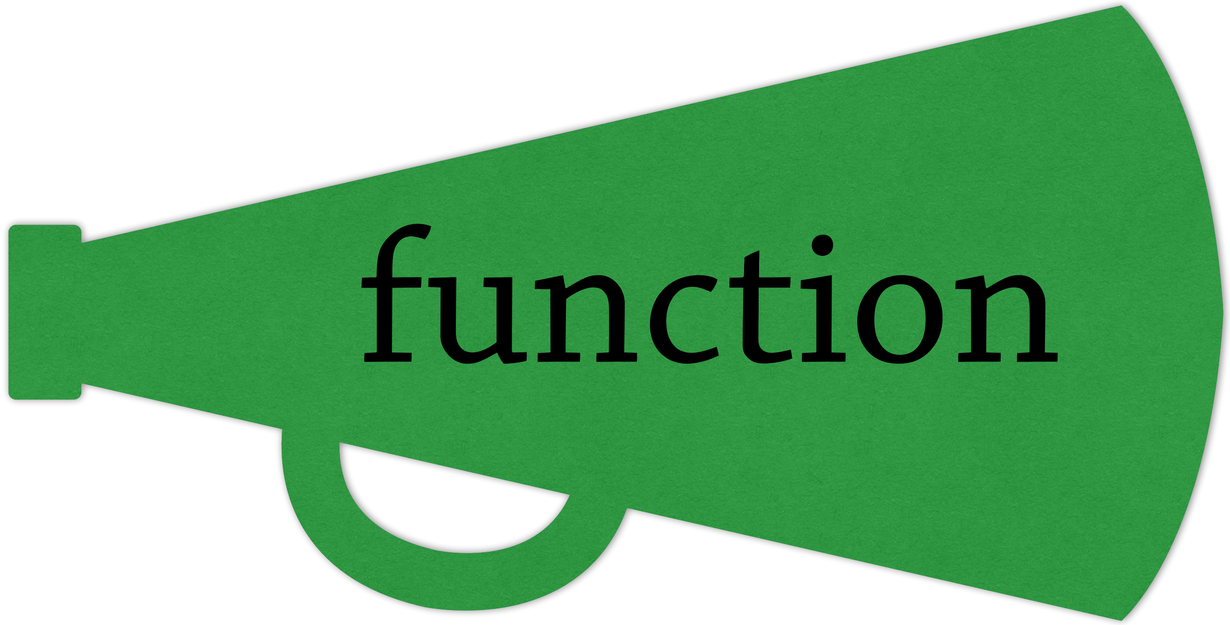


Solution:
Automatic Annotations

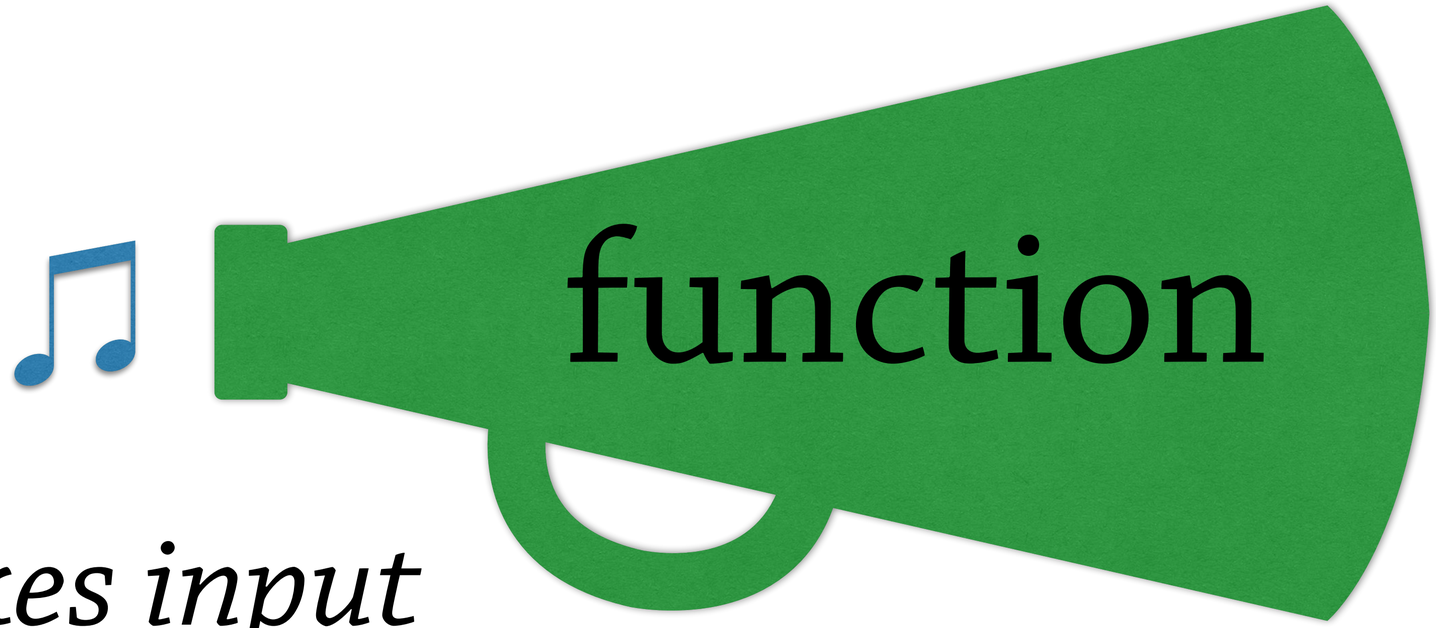
Our approach:

1. Instrument program
2. Observe running program
3. Summarize execution

Example: A function



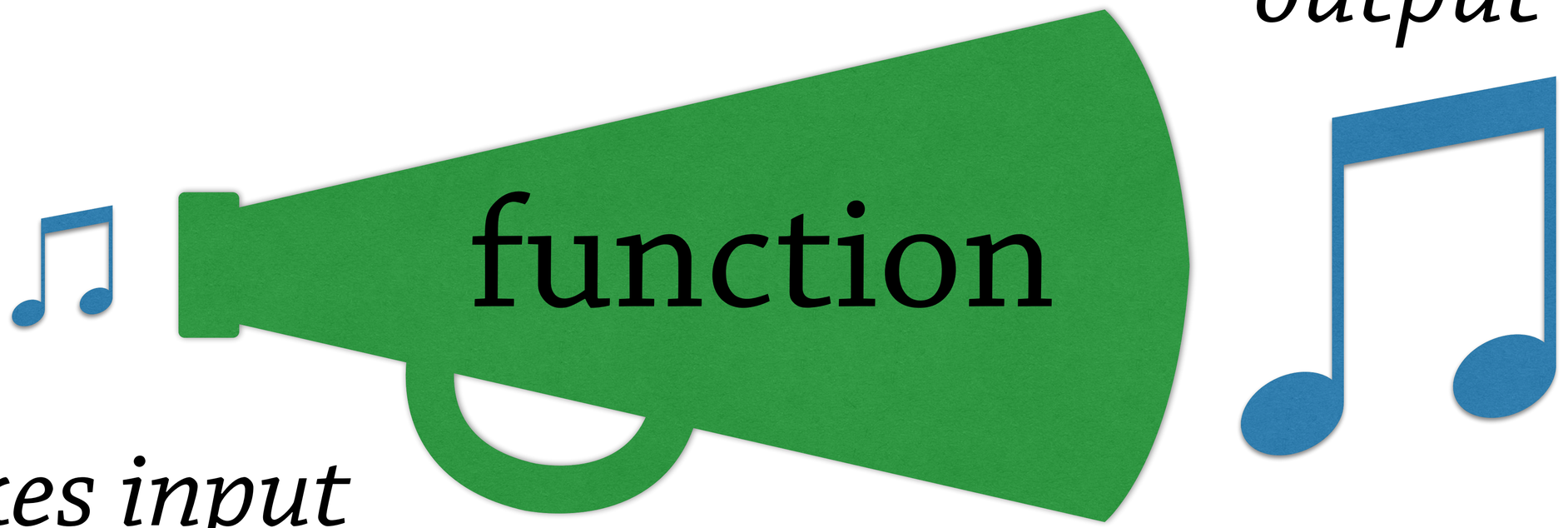
Example: A function



Takes input

Example: A function

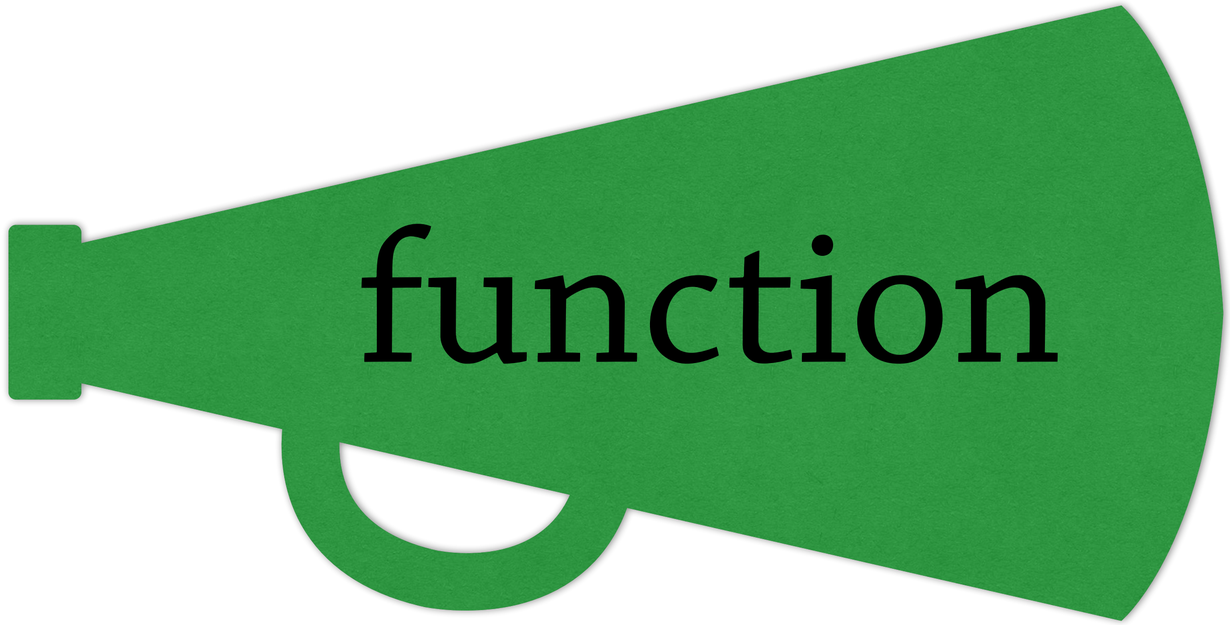
Produces output



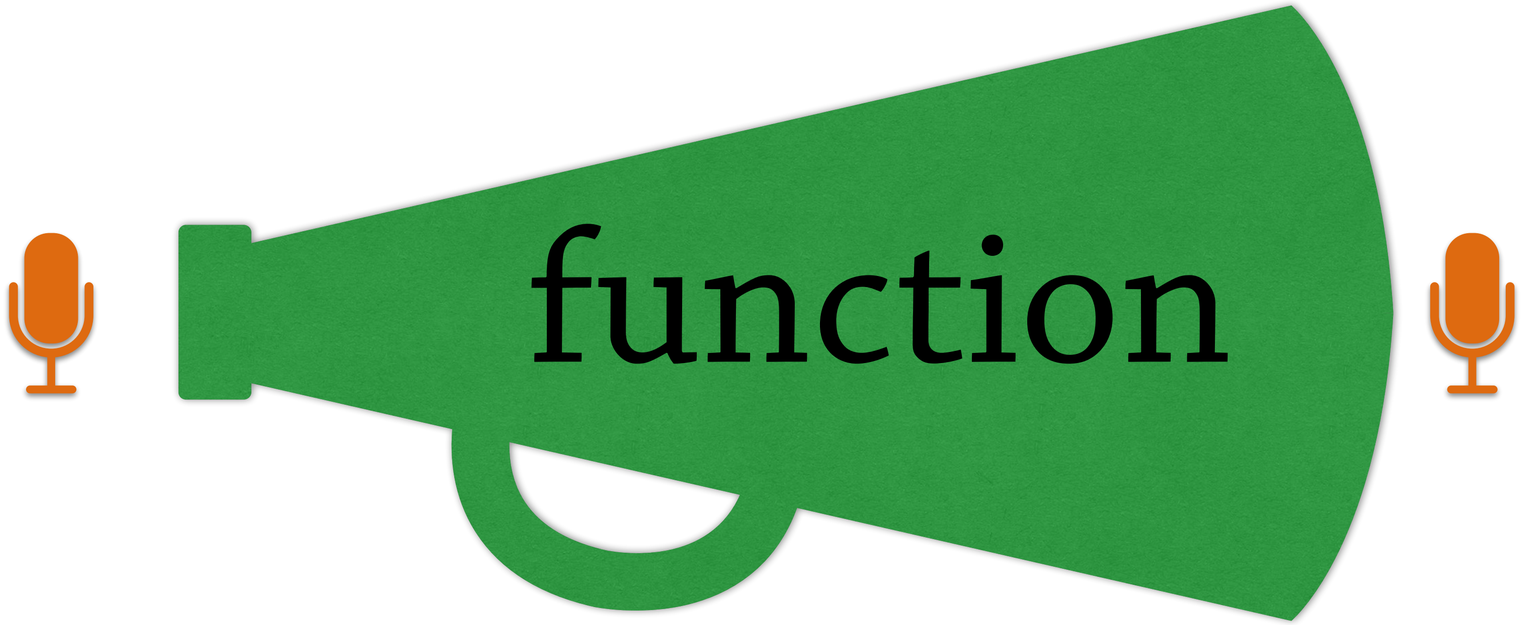
Takes input

Our approach

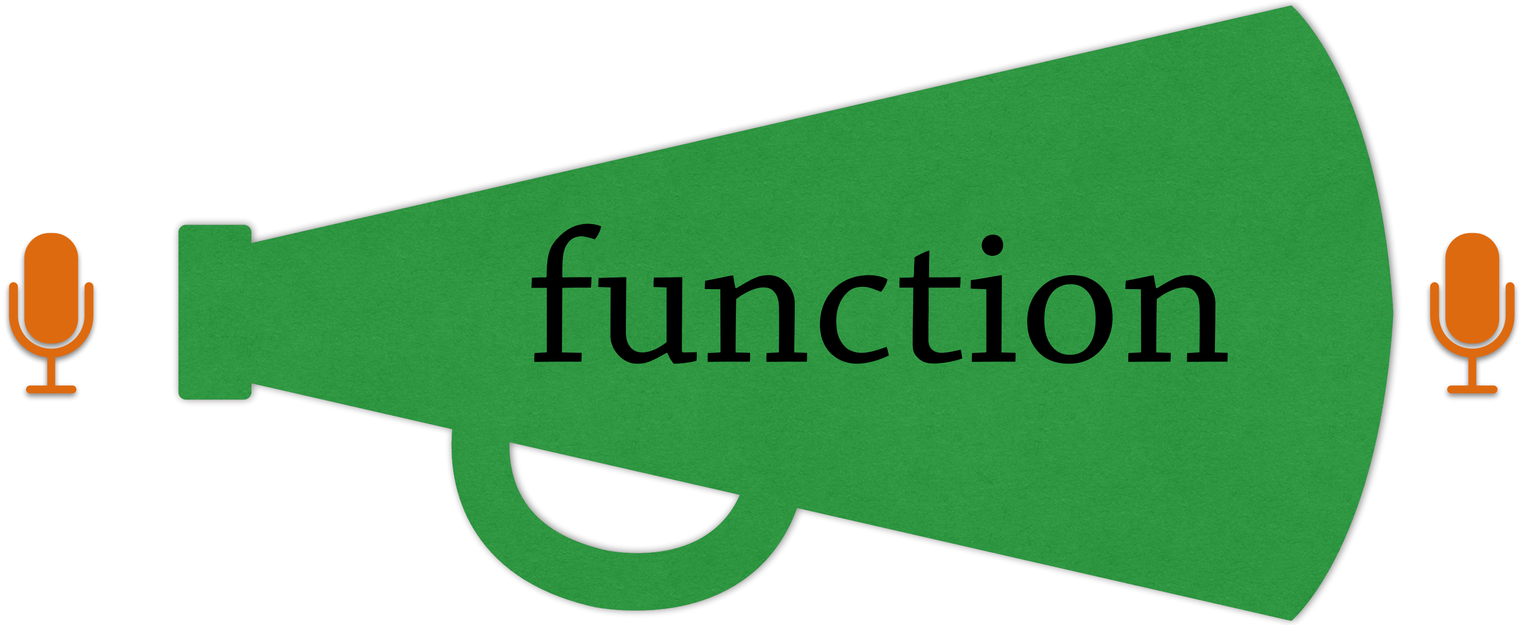
Step 1:
Instrumentation



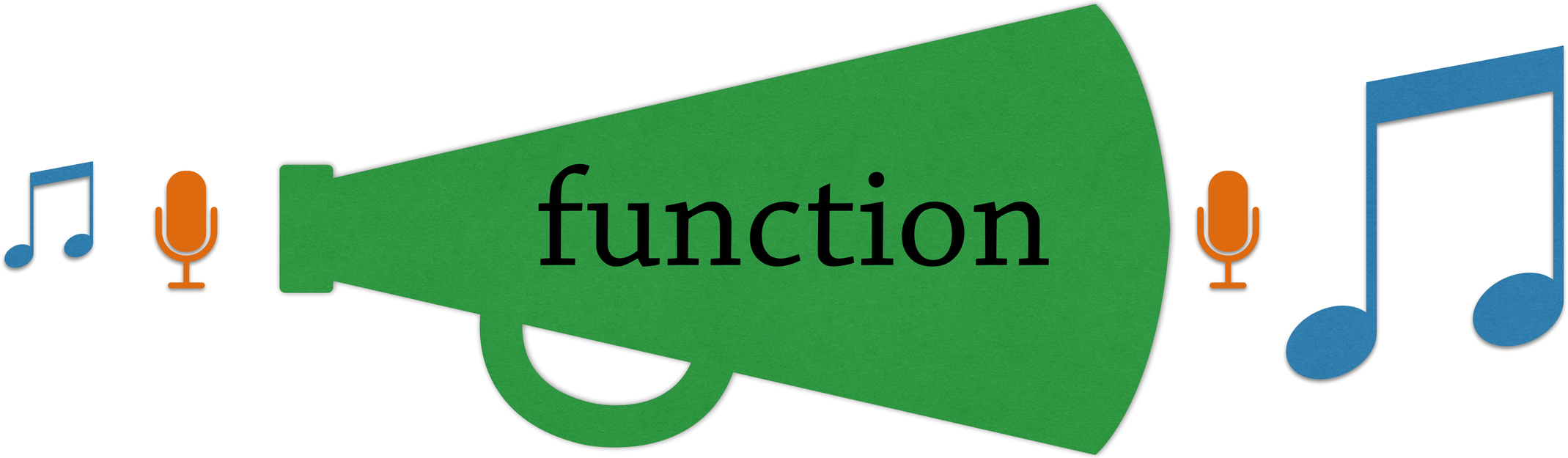
Step 1:
Instrumentation



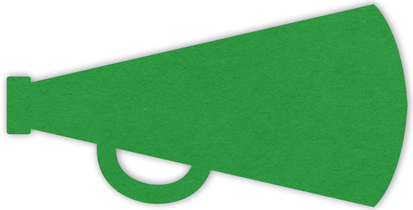
Step 2:
Observe running program



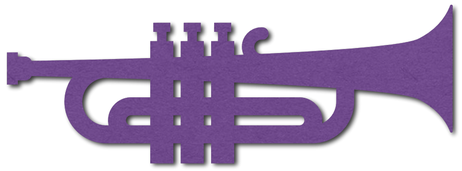
Step 2:
Observe running program



Step 3:
Summarize execution

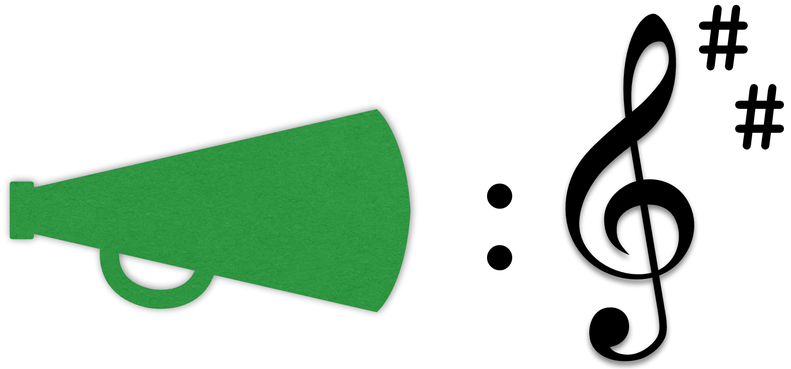


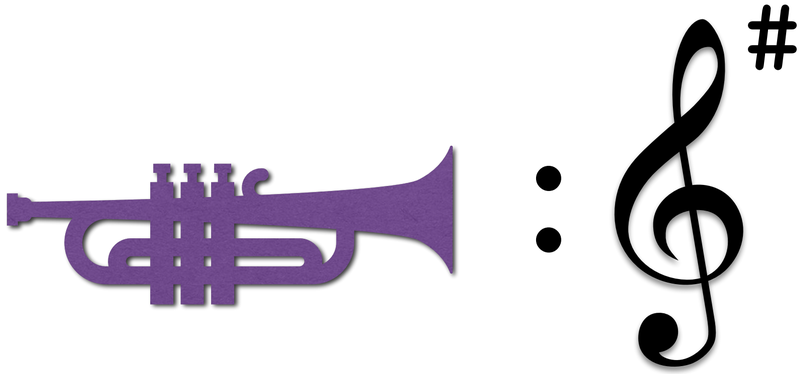
:



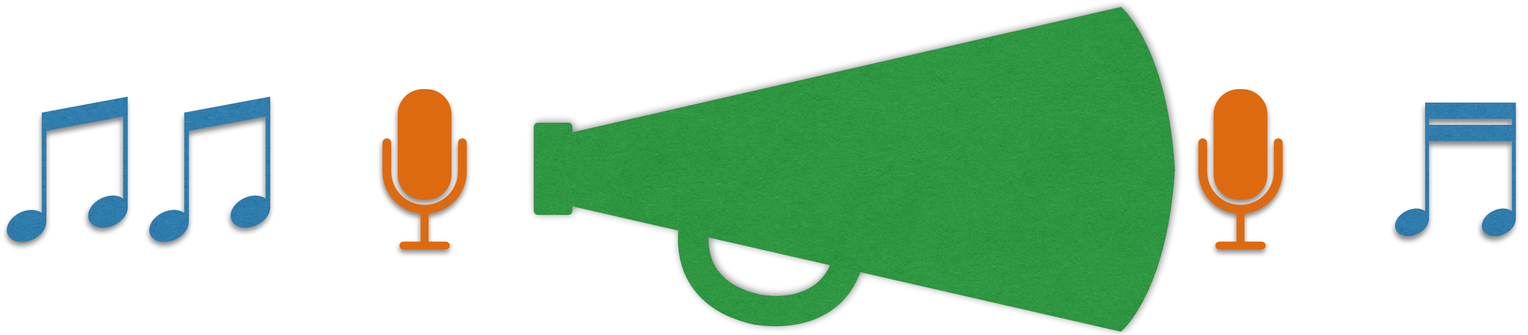
:

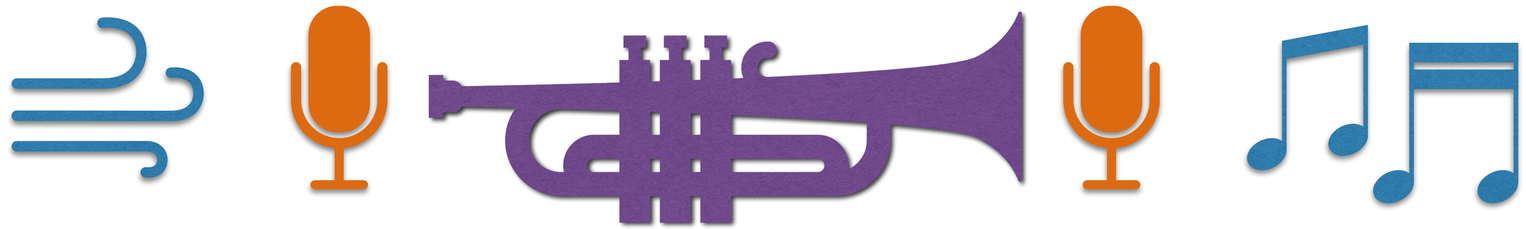
Step 3:
Summarize execution



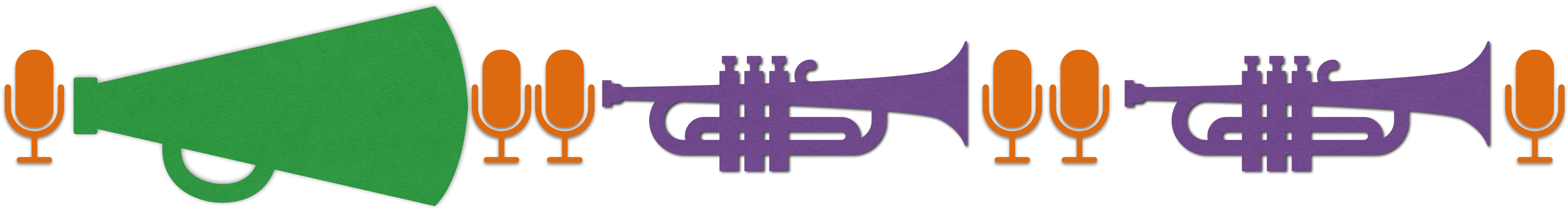


This talk





This talk



...Something completely different

ClojureScript Compilation

Compilation

Clojure	*JavaScript*
`1`	`1`
`2`	`2`
`‘(1 2)`	`cljs.core.list(1,2)`

Compilation Lifecycle

Clojure Form	*AST*	*JS (string)*
`1`	`{:op :val :val 1}`	`"1"`
`2`	`{:op :val :val 2}`	`"2"`
`'(1 2)`	`{:op :list :items [{:op :val :val 1} {:op :val :val 2}]}`	`"cljs.core.list(1,2)"`

Compilation Lifecycle

Analyze

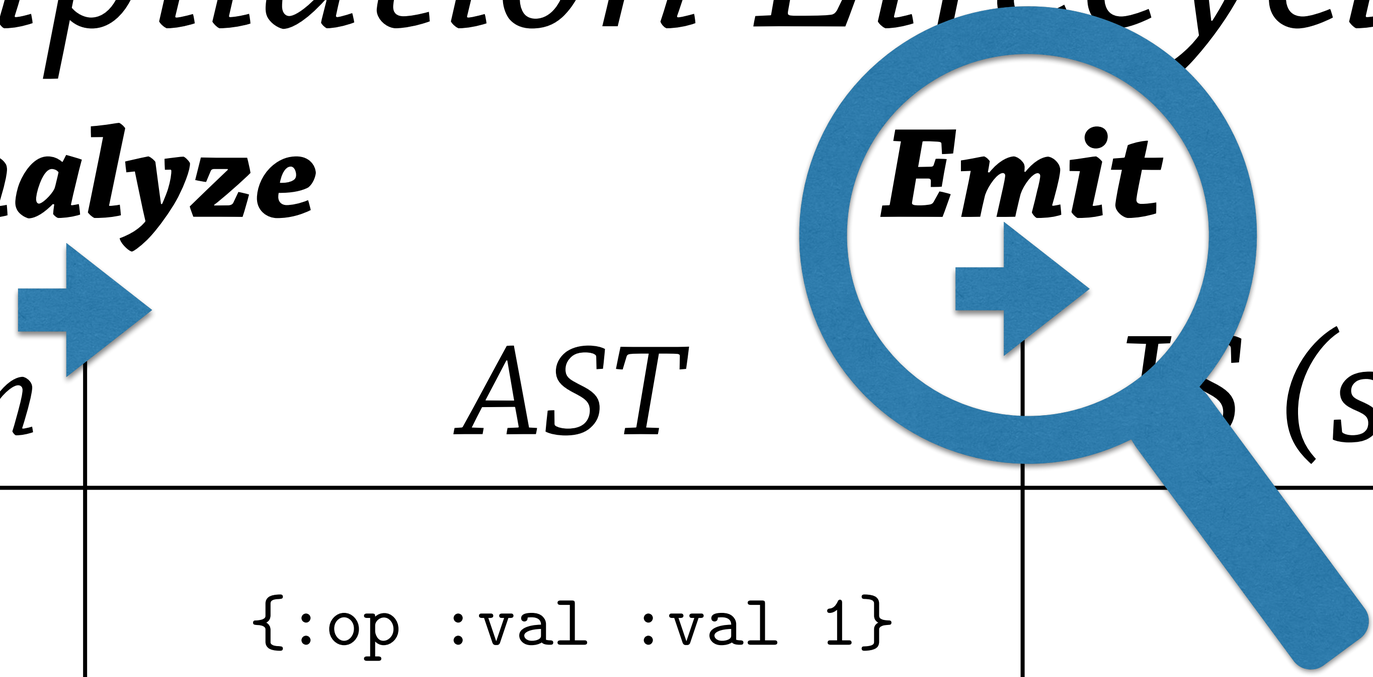
Clojure Form	AST	JS (string)
1	{:op :val :val 1}	"1"
2	{:op :val :val 2}	"2"
'(1 2)	{:op :list :items [{:op :val :val 1} {:op :val :val 2}]}	"cljs.core.list(1,2)"

Compilation Lifecycle

Analyze → **Emit** →

Clojure Form	AST	JS (string)
`1`	`{:op :val :val 1}`	`"1"`
`2`	`{:op :val :val 2}`	`"2"`
`'(1 2)`	`{:op :list :items [{:op :val :val 1} {:op :val :val 2}]}`	`"cljs.core.list(1,2)"`

Compilation Lifecycle



Clojure Form	AST	JS (string)
1	{:op :val :val 1}	"1"
2	{:op :val :val 2}	"2"
'(1 2)	{:op :list :items [{:op :val :val 1} {:op :val :val 2}]}	"cljs.core.list(1,2)"

The emitter

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Instrumentation

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Paths

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

[emit (arg)]

[emit (ret)]

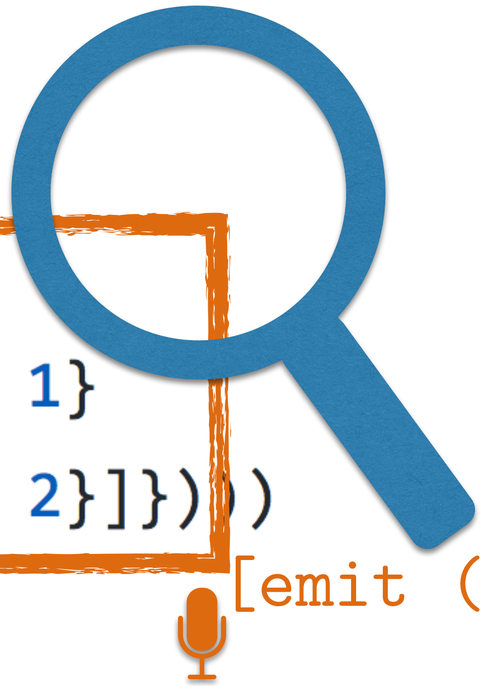
[emit (ret)]

Sample execution

```
(is (= "cljs.core.list(1,2)"
       (emit {:op :list
              :items [{:op :val :val 1}
                      {:op :val :val 2}]})))
```

Sample execution

```
(is (= "cljs.core.list(1,2)"
       (emit {:op :list
              :items [{:op :val :val 1}
                      {:op :val :val 2}]})))
```



[emit (arg)]

Instrumentation results

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

[emit (arg)]

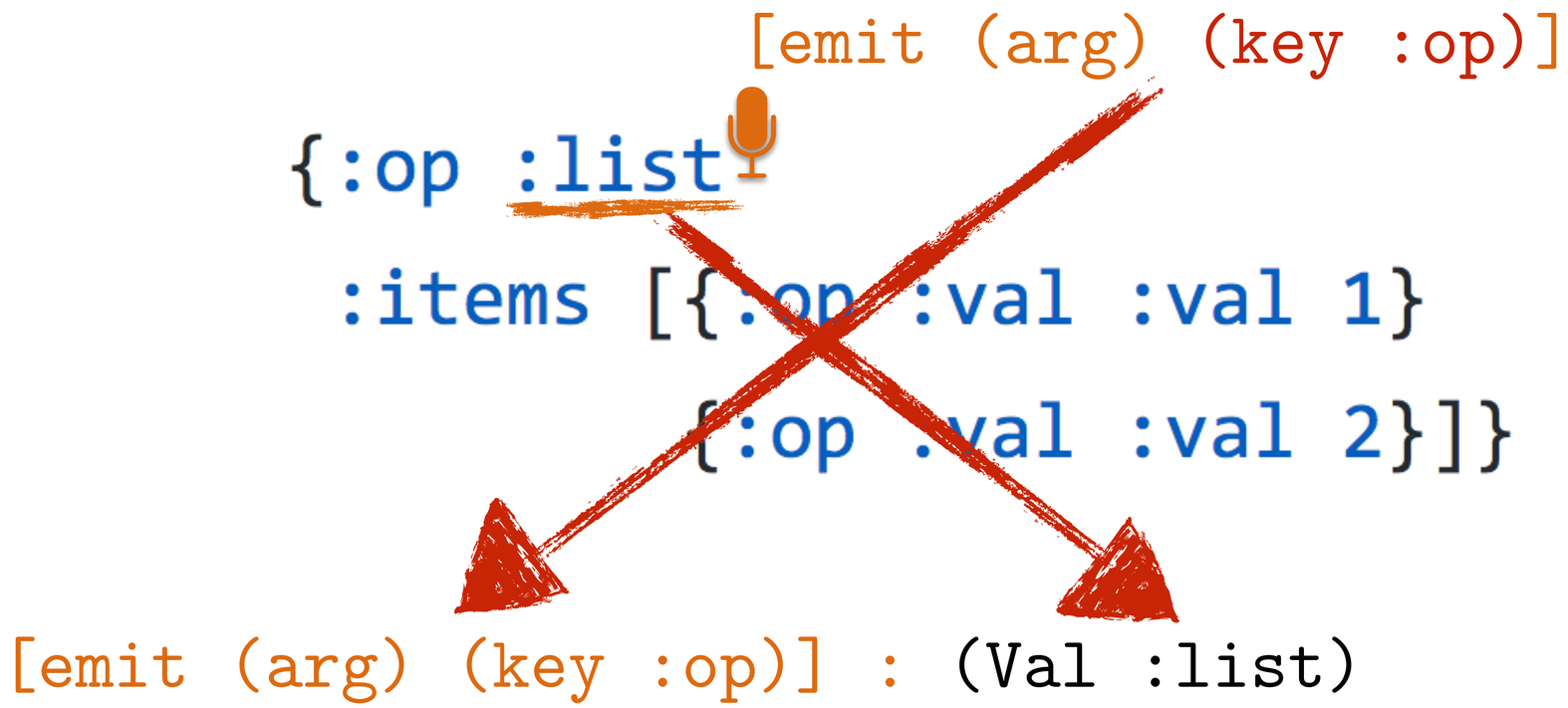
Instrumentation results

```
                [emit (arg)          ]
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Instrumentation results

```
                [emit (arg) (key :op)]
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Instrumentation results



Instrumentation results

```
                    [emit (arg) (key :op)]
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results

```
{:op :list
          [emit (arg) (key :items)]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results

```
{:op :list
          [emit (arg) (key :items)        ]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results

```
{:op :list
          [emit (arg) (key :items) (vec)]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :items) (vec)                ]
```

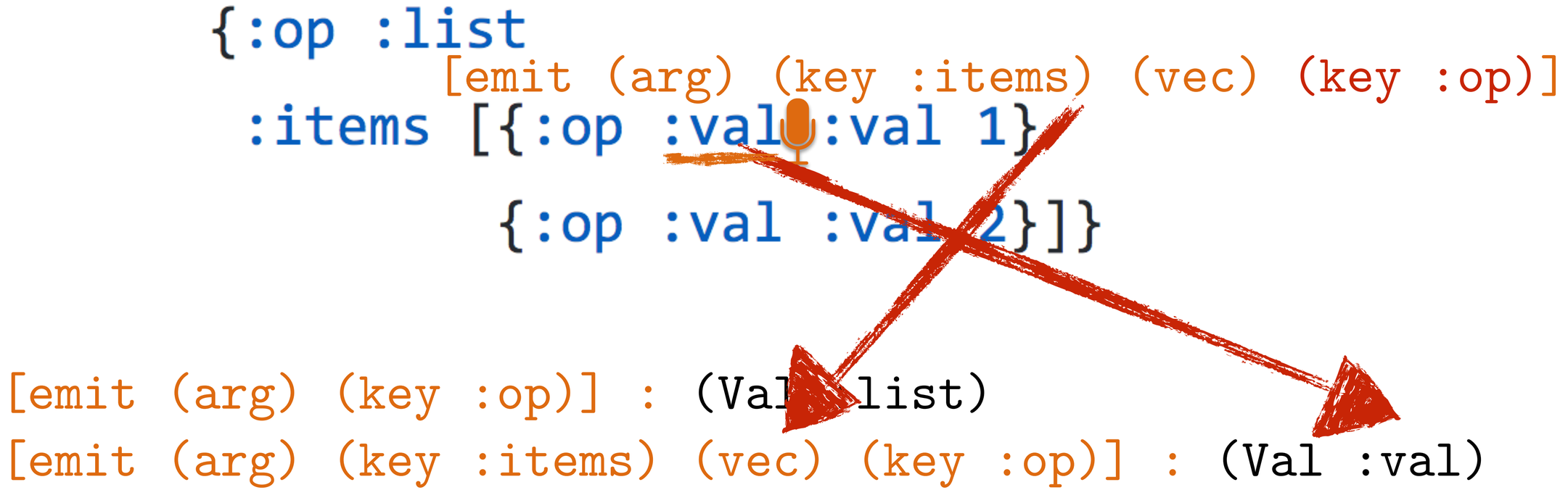
```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results

```
{:op :list
        [emit (arg) (key :items) (vec) (key :op)]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
```

Instrumentation results



Instrumentation results

```
{:op :list
          [emit (arg) (key :items) (vec) (key :op)]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
[emit (arg) (key :items) (vec) (key :op)] : (Val :val)
```

Instrumentation results

```
{:op :list
          [emit (arg) (key :items) (vec)                ]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

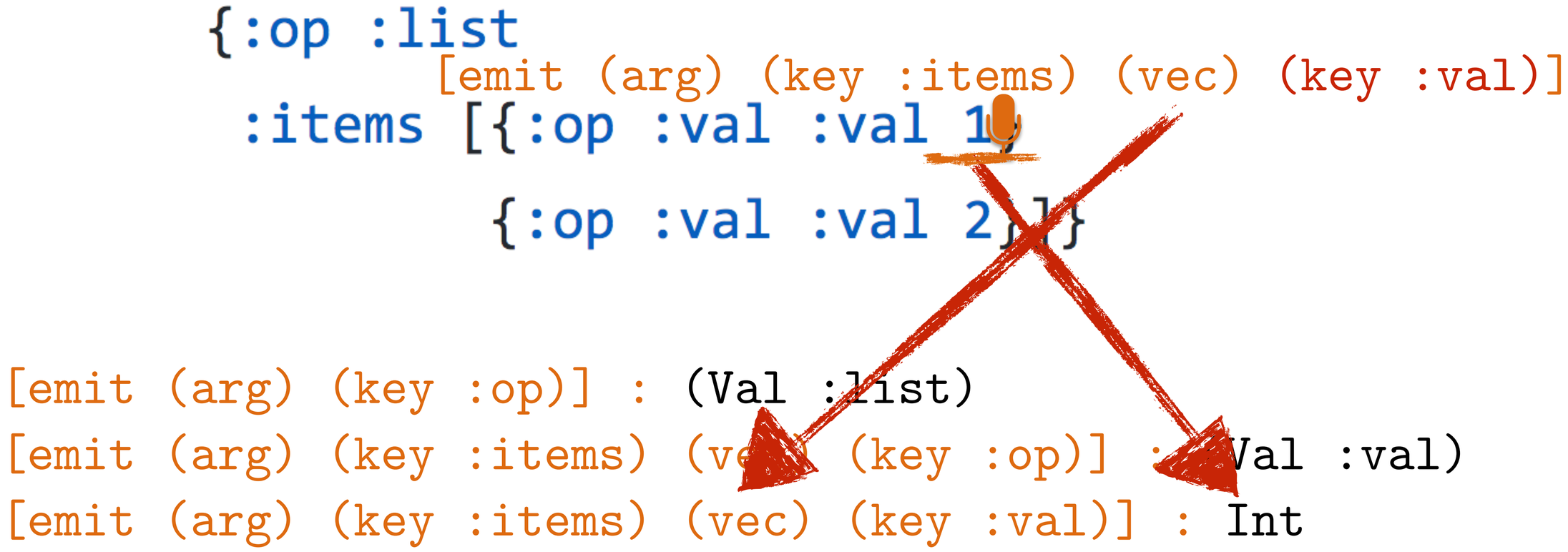
```
[emit (arg) (key :op)] : (Val :list)
[emit (arg) (key :items) (vec) (key :op)] : (Val :val)
```

Instrumentation results

```
{:op :list
          [emit (arg) (key :items) (vec) (key :val)]
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)] : (Val :list)
[emit (arg) (key :items) (vec) (key :op)] : (Val :val)
```

Instrumentation results



Problem

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

[emit (arg)]

Problem

Walk
the same data,
twice …

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

[emit (arg)]

[emit-val (arg)]

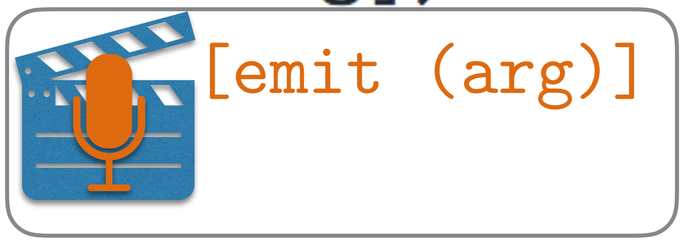
Recording Approach	*Tradeoffs*
Eager 	• *Rich recordings* • *Significant runtime cost*

Solution: Lazy recording

Solution: Lazy recording

Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```



Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

```
[emit (arg)]
```

Apply wrapper

Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

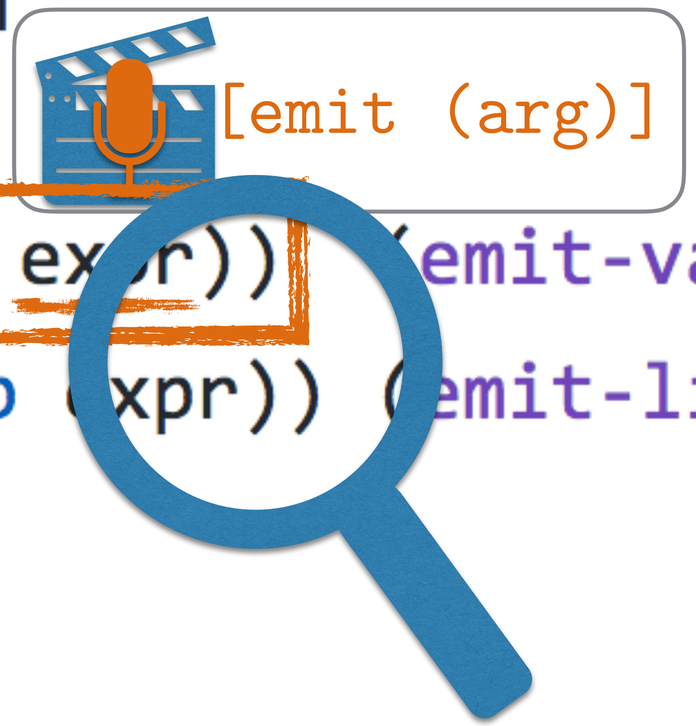
Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Lookup

Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr)) (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```



Lazy Instrumentation results

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

```
[emit (arg)]
```

Lazy
Instrumentation results


```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

Lazy Instrumentation results


```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

Lazy Instrumentation results


```
[emit (arg) (key :op)] : (Val :list)
```

Lazy Instrumentation results

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit (arg) (key :op)]
```

```
[emit (arg) (key :op)] : (Val :list)
```

Done (lazy!)

Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

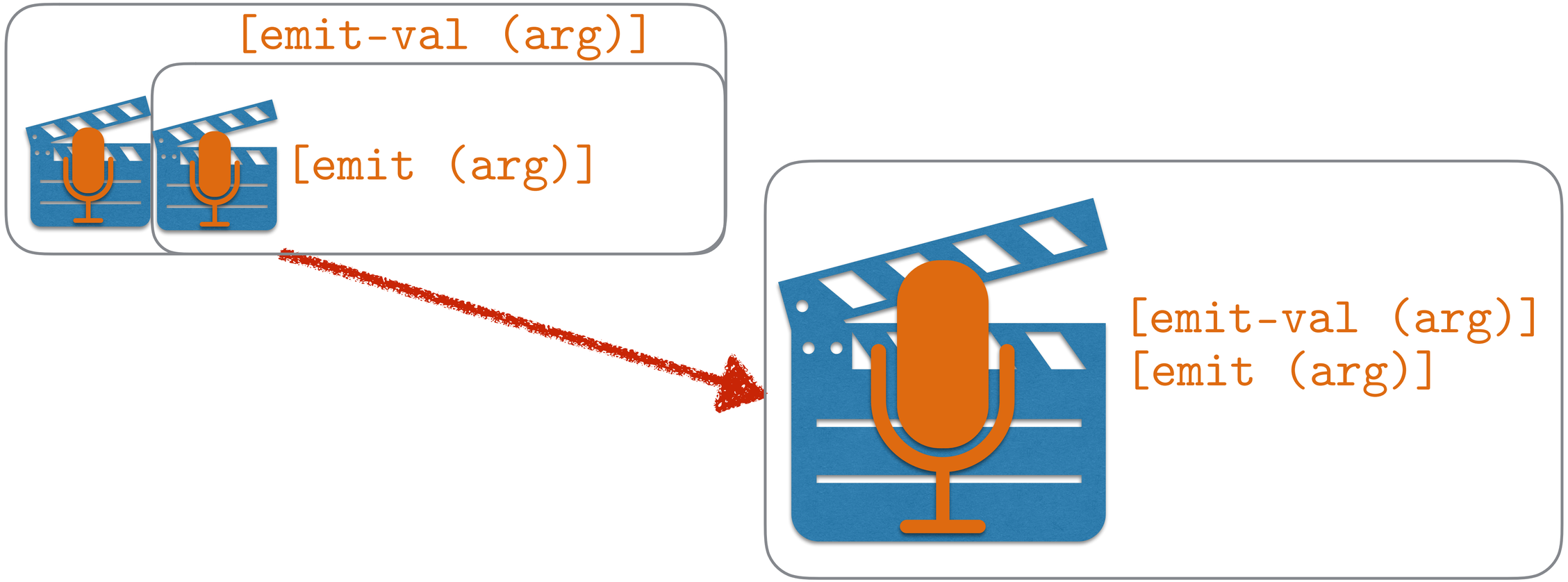
Solution: Lazy recording

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (=                    pr))  (emit-val expr)
    (=                  expr)) (emit-list expr)
    ...))
```

Space inefficient wrapping

Recording Approach	*Tradeoffs*
Eager 	• *Rich recordings* • *Significant runtime cost*
Lazy 	• *Sparse recordings* • *Low runtime cost* • *Redundant rerecording*

Space-efficient wrapping



Space-efficient wrapping

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :                )  (emit-val expr)
    (= :l               pr)) (emit-list expr)
    ...))
```

Combine…

Space-efficient wrapping

```
(ann emit [Expr -> String])
(defn emit [expr]
  (cond
    (= :val (:op expr))  (emit-val expr)
    (= :list (:op expr)) (emit-list expr)
    ...))
```

```
[emit-val (arg)]
[emit (arg)]
```

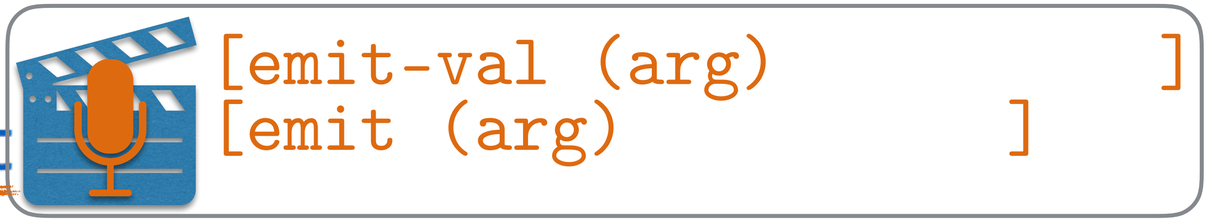
Space-efficient Lookup

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

```
[emit-val (arg)]
[emit (arg)]
```

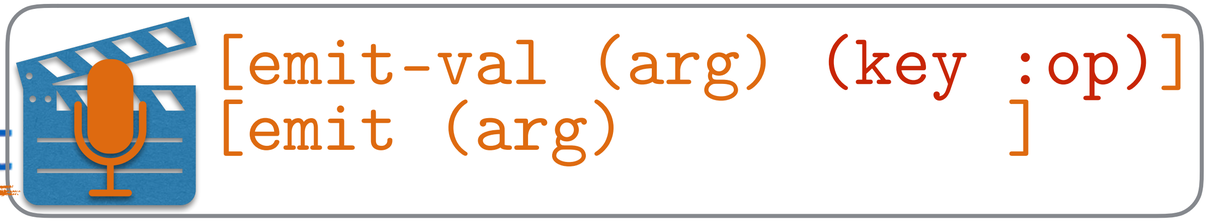
Space-efficient Lookup




```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

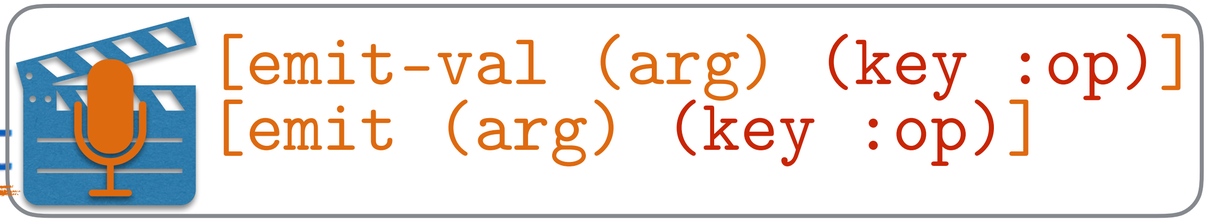
Space-efficient Lookup




```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Lookup :op

Space-efficient Lookup




```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

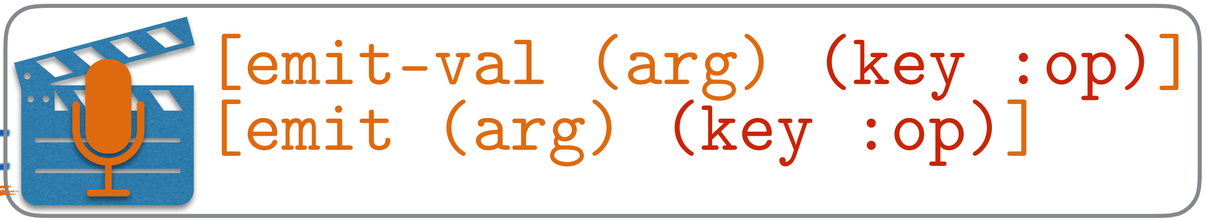
Lookup :op

Space-efficient Lookup


```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

```
[emit-val (arg) (key :op)] : (Val :list)
[emit (arg) (key :op)] : (Val :list)
```

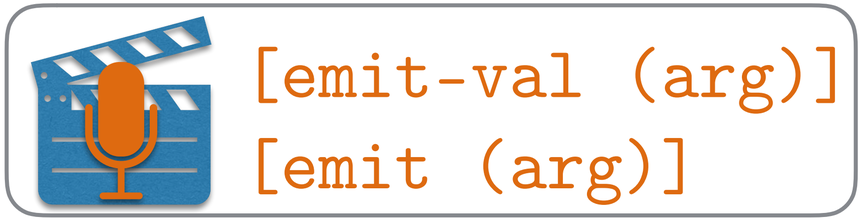
Space-efficient Lookup



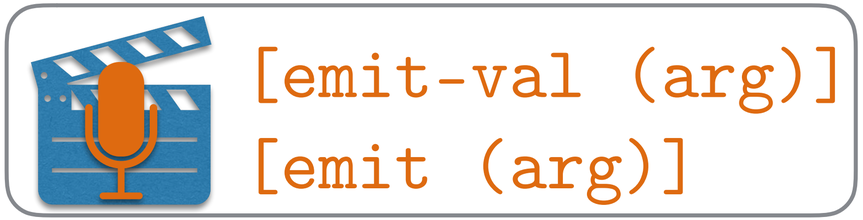

```
{:op :list
 :items [{:op :val :val 1}
         {:op :val :val 2}]}
```

Single pass over data!

```
[emit-val (arg) (key :op)] : (Val :list)
[emit (arg) (key :op)] : (Val :list)
```

Recording Approach	*Tradeoffs*
Eager 	• *Rich recordings* • *Significant runtime cost*
Lazy 	• *Sparse recordings* • *Low runtime cost* • *Redundant rerecording*
Lazy + ***Space-efficient*** 	• *Sparse recordings* • *Low runtime cost* • *Less rerecording*

Thanks

Recording Approach	*Tradeoffs*
Eager 	• *Rich recordings* • *Significant runtime cost*
Lazy 	• *Sparse recordings* • *Low runtime cost* • *Redundant rerecording*
Lazy **+** ***Space-efficient*** 	• *Sparse recordings* • *Low runtime cost* • *Less rerecording*