Programming with Values in Clojure

Alan Dipert
@alandipert

relevance

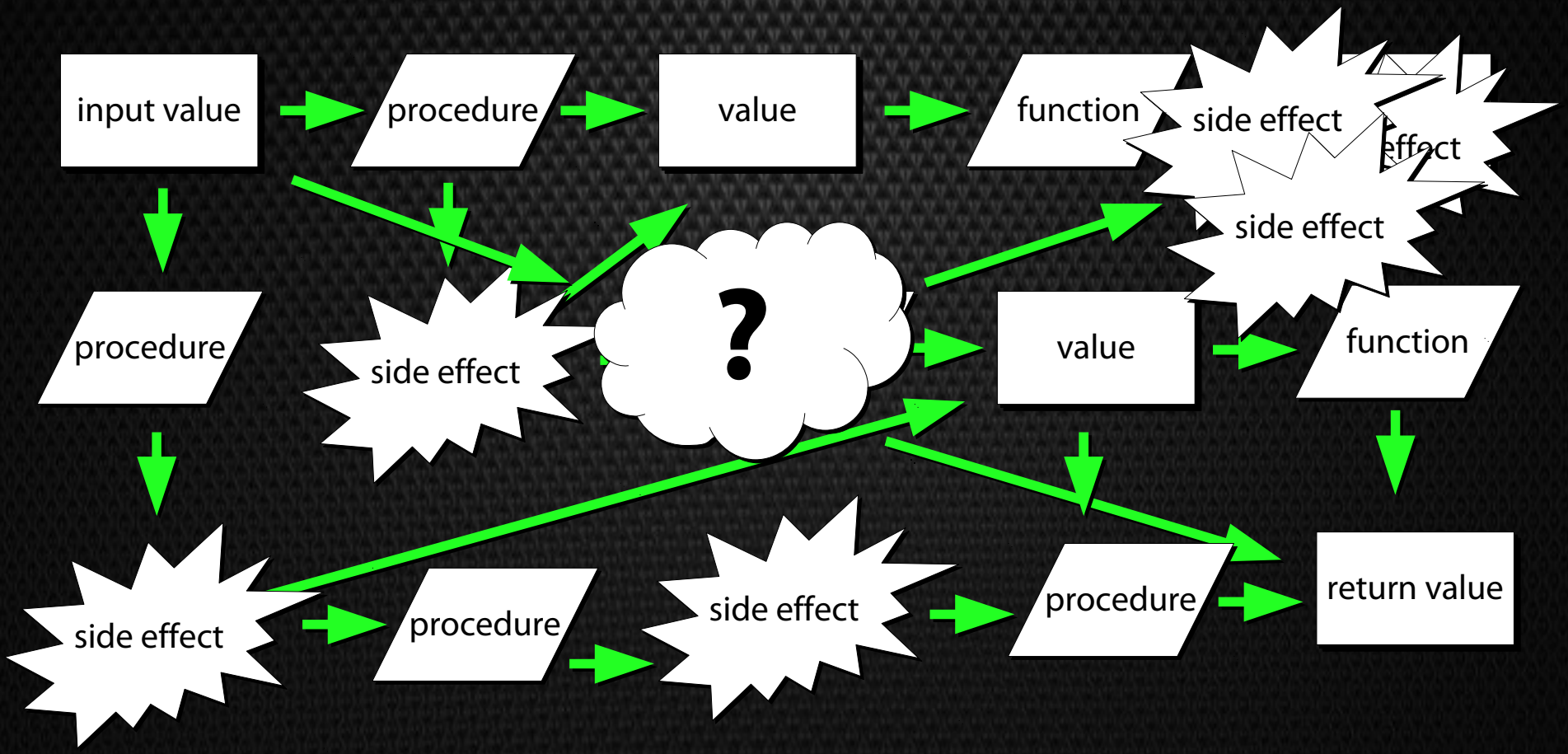
Programming with...

Integrity

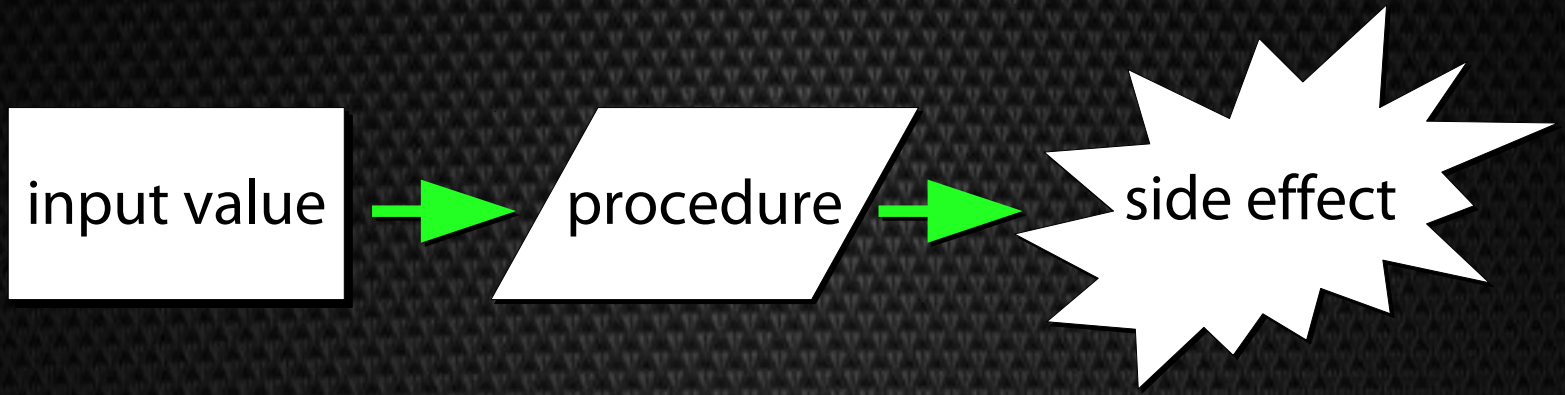
Courage

Respect

Problem


input value
procedure
value
function
side effect
side effect
procedure
side effect
?
value
function
side effect
procedure
side effect
procedure
return value

Procedure



- Optionally takes input
- Performs side effects
- Not referentially transparent
- Weak to compose

Function

- Optionally takes input
- Referentially transparent
- Composable
- Aids in debugging

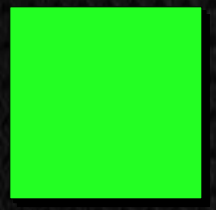
Prog alues

input value → f n → return value

procedure

Values

- Numbers, strings, symbols, etc.
  - `1, “clojure”, \x, pants, :soccer`
- Associative



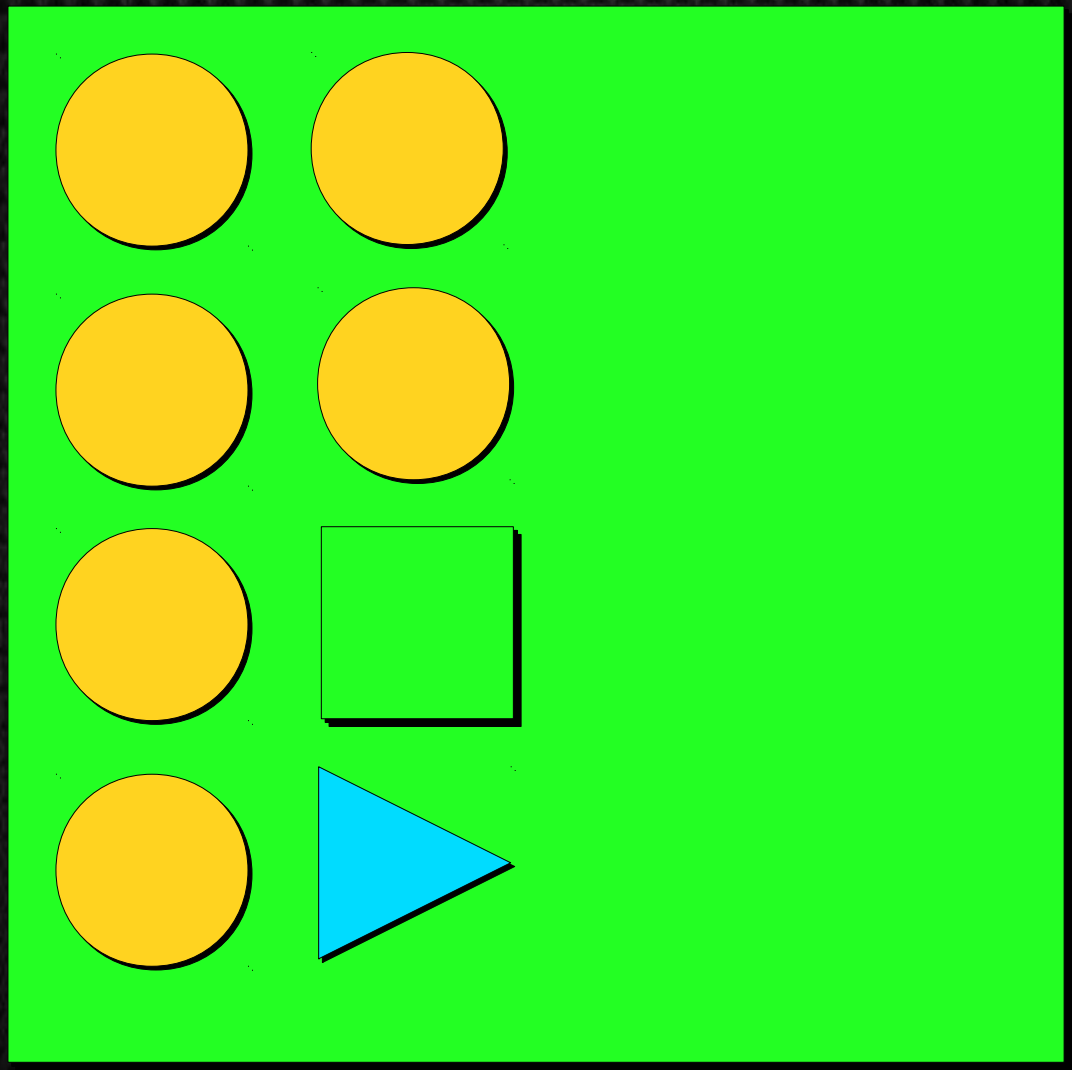
  - `{:name “Bob” :age 34}`
  - `#{:vanilla, :strawberry}`
- Sequential

  - `[1, 3, 5]`
  - `(x y z)`

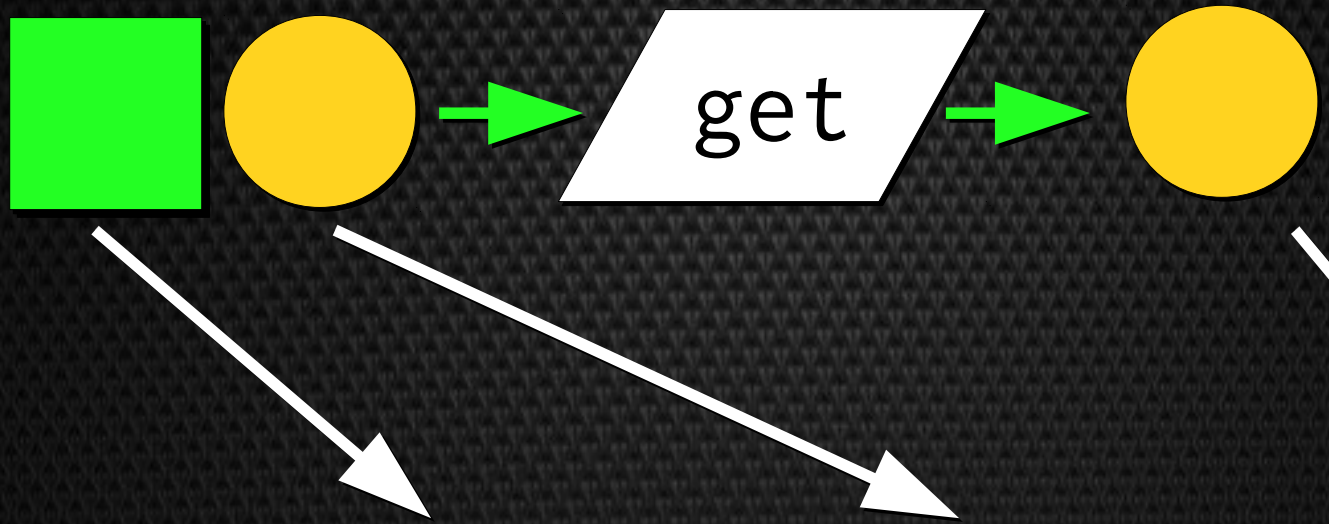
Composite Values

```
{:name “Bob”
 :age 34
 :favorites #{:blueberry :chocolate}
 :goals [“skydive” “sail”]
 :address {:street “123 Penny Lane”
           :city “London”}}
```

Composite Values

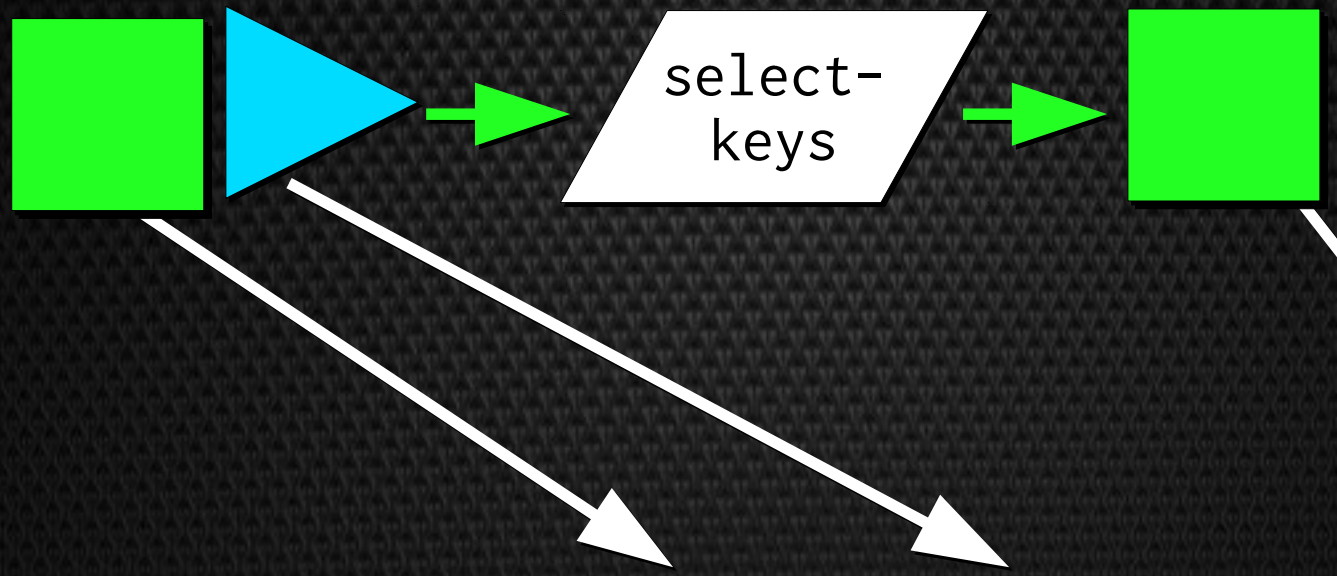


Shape Conversion



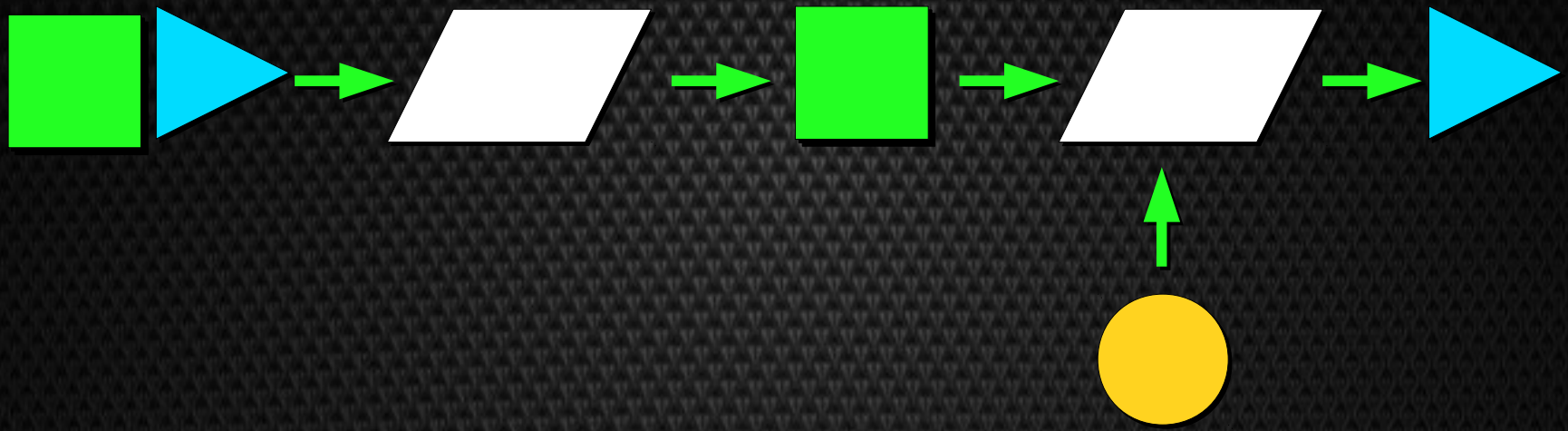

```
(get {:name “Bob”} :name) ;=> “Bob”
```

Shape Conversion




```
(select-keys {:a 1 :b 2} [:a]) ;=> {:a 1}
```

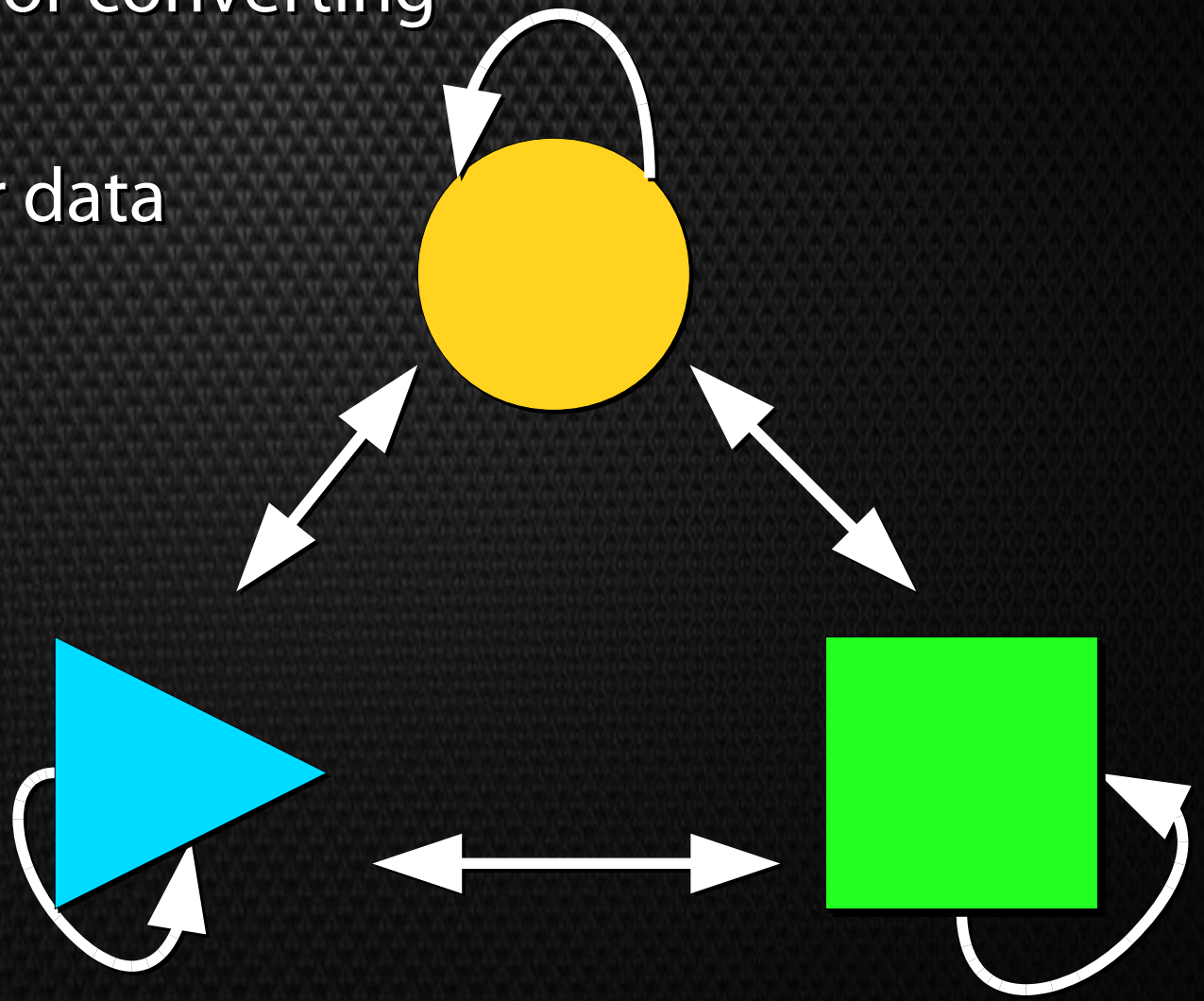
Your Program



- Takes some shape of input value
- Performs intermediate conversions
- Returns final result value

Clojure

- Functions/idioms for converting shapes
- Pick based on your data
- Wire together



Working with Sequences

- Sequential, possibly infinite things
- Clojure provides comprehensive support
- Functions can be wired together to yield
  - New sequences
  - New shapes
- Arities support syntactical composition
- Function composition helps too

->>

- “Thread last” macro
- Useful because seq functions take seq as last arg
- Easy to comment out intermediate results

```
(apply str
       (interpose \space
                  (filter allowed coll)))

(->> coll
     (filter allowed)
     (interpose \space)
     (apply str))
```

Complement and Compose

- `complement` to create the "opposite" function
- `comp` to compose functions

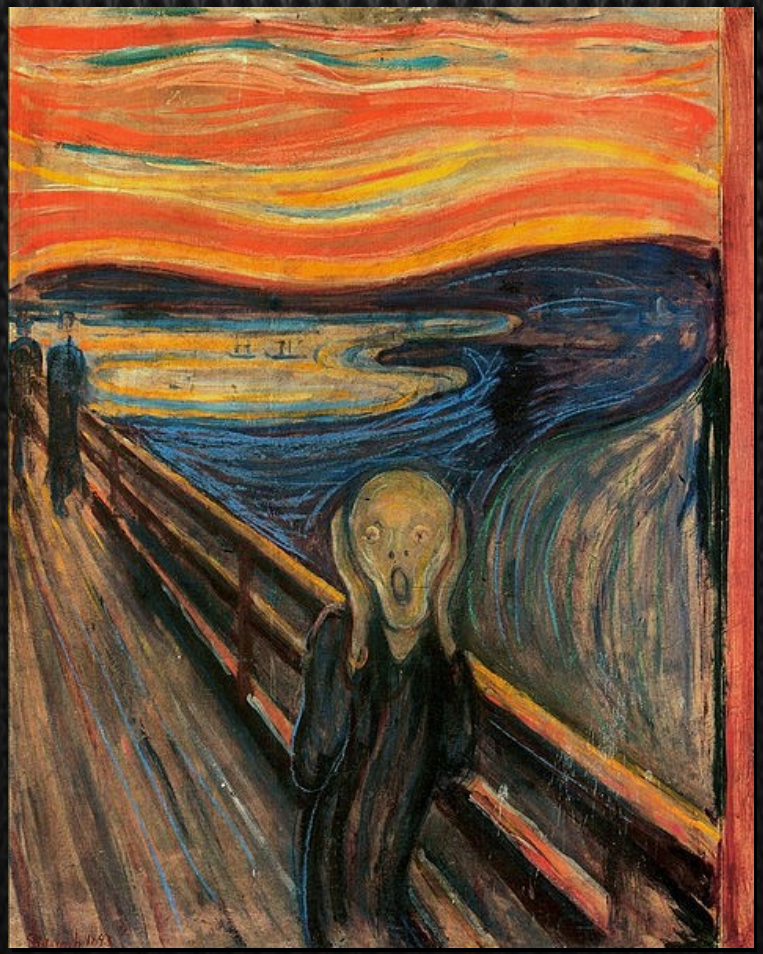
```
(filter (complement zero?) (range 5))

;=> (1 2 3 4)

(map (comp inc :n) [{:n 1} {:n 2}])

;=> (2 3)
```

keep = map + filter

```
(map first
     (filter (comp #{:a :b} first)
             [[:a] [:d]]))

;=> (:a)

(keep (comp #{:a} first) [[:a] [:b]])

;=> (:a)
```

contains?

- “How do I find an element in a list. Contains?”



some

- `contains?` looks for key in indexed collections
- `some` does linear search

```
(some #{:b} [:a :b :c])
```

```
;=> :b
```

- Linear search? Maybe you're working with the wrong shape...

Associative Shapes

- Key/Value (or just distinct elements)
- Arities support syntactical composition and interoperation with STM
- Core functions support path-based inspection, update in nested associative data
- Maps and sets are functions of their keys
- Records made with `defrecord` are map-like

Map in, Map out

```
(select-keys {:a 10 :b 20} [:a])

;=> {:a 10}

(merge {:a 10} {:b 20})

;=> {:a 10 :b 20}

(defrecord Foo [a])
(assoc (Foo. 1) :b 2)

;=> #user.Foo{:a 1, :b 2}
```

A Bit about Records

- Use maps first
- Then, “graduate” to records when domain solidifies or you need polymorphism
- Your code should stay the same
- Consider using “constructors” from the start
  - Hides construction of map or record objects
  - Seamless map to record conversion
  - Place to hang validations or constraints

Constructing Records

```
(defrecord Drinker [age])

(defn make-drinker [age]
  {:pre [(>= age 21)]}
  (Drinker. age))

; user=> (make-drinker 4)
; AssertionError Assert failed:
; (>= age 21)  user/make-drinker
```

->

- “Thread first” macro
- Useful for maps, interop
- Mix functions and methods in same “pipeline”

```
(-> "hi"
    (.charAt 0)
    int)
```

Path-based Lookup/Update

- `get-in` and `update-in`: path-based lookup/update
- "Drill" into nested associative structures
  - Without pulling them apart

get-in

```
(def m {:name "Alan"
        :address {:zip 27701}})

(get-in m [:address :zip])

;=> 27701
```

update-in

```
(def m {:name "Alan"
        :address {:zip 27701}})

(update-in m [:address :zip] inc)

;=> {:name "Alan", :address {:zip 27702}}
```

update-in + fnil

```
(def m {:name "Alan"})

(update-in m [:address :zip] inc)

; NullPointerException
; clojure.lang.Numbers.ops
; (Numbers.java:942)
```

update-in + fnil

```
(def m {:name "Alan"})

(update-in m
           [:address :zip]
           (fnil inc 10000))

;=> {:name "Alan", :address {:zip 10001}}
```

Mad Science

- Because not all of your Programs with Values need to be important.

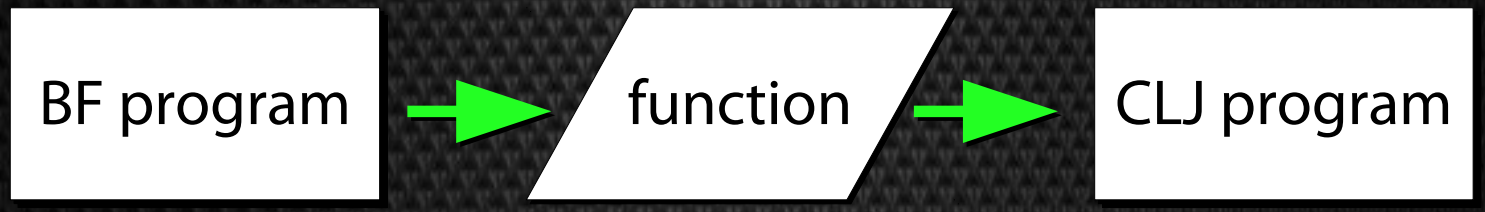
Brainf***

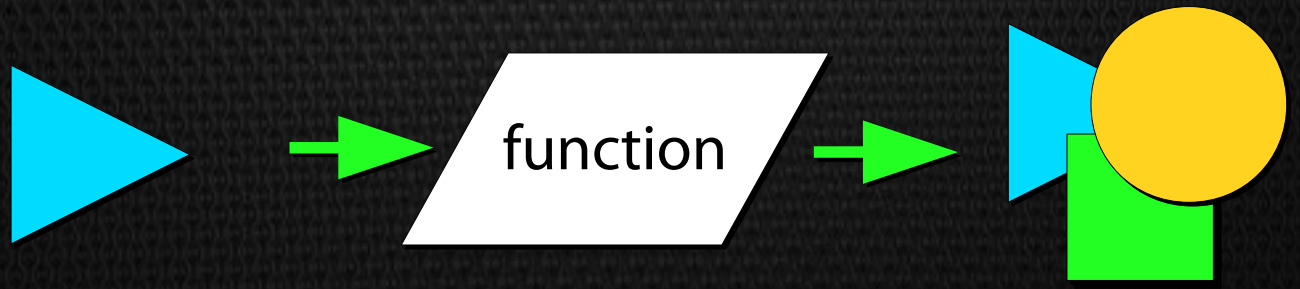
- Program:

```
+++++[->+<]
```

- Memory tape:

BF to Clojure Compiler





BF to Clojure Compiler

“++++++[->+<]”

“+ + + + + [- > + <]”

```
(read-string “(+ + + + + [ - > + < ])”)
```

```
(defn bfc
  [program]
  (let [allowed #{\+ \- \< \> \[ \] \.}
        src (->> program (filter allowed)
                 (interpose \space) (apply str))
        fns (zipmap '(- + < > . ?) (repeatedly gensym))]
    (letfn [(bfc* [s]
              (if (vector? s)
                `(while (not (~(fns '?))) ~@(map bfc* s))
                `(~(fns s))))]
      `(let [tape# (int-array 60000 0)
             pointer# (atom 0)
             ~(fns '-) #(aset tape# @pointer# (dec (aget tape# @pointer#)))
             ~(fns '+) #(aset tape# @pointer# (inc (aget tape# @pointer#)))
             ~(fns '<) #(swap! pointer# dec)
             ~(fns '>) #(swap! pointer# inc)
             ~(fns '.) #(print (char (aget tape# @pointer#)))
             ~(fns '?) #(zero? (aget tape# @pointer#))]
         ~@(map bfc* (read-string (str "(" src ")"))) nil))))
```

Tips

- Recognize the shapes in your domain
- Pick the Clojure functions and idioms you need to work with them
- Separate functions and procedures
- Think of your program as a transition of valuess
- Mostly pure functions, some explicit procedures for side effects

Thank you!

Alan Dipert
@alandipert

relevance