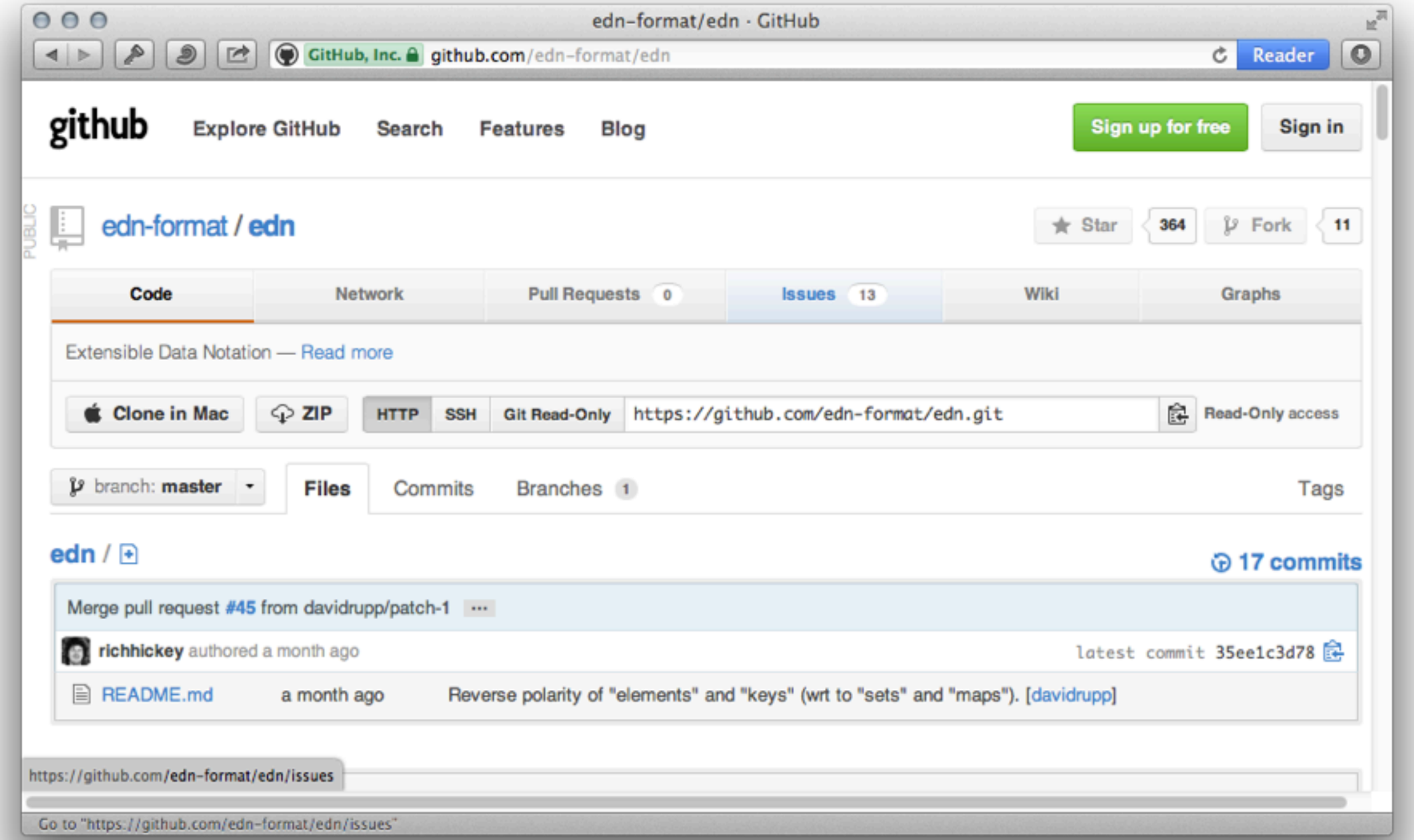
Clojure
in 10 big ideas

@stuarthalloway
stu@cognitect.com

1. edn


edn-format/edn · GitHub
GitHub, Inc.
github.com/edn-format/edn
Reader
github
Explore GitHub
Search
Features
Blog
Sign up for free
Sign in
PUBLIC
edn-format / edn
Star
364
Fork
11
Code
Network
Pull Requests 0
Issues 13
Wiki
Graphs
Extensible Data Notation — Read more
Clone in Mac
ZIP
HTTP
SSH
Git Read-Only
https://github.com/edn-format/edn.git
Read-Only access
branch: master
Files
Commits
Branches 1
Tags
edn /
17 commits
Merge pull request #45 from davidrupp/patch-1
richhickey authored a month ago
latest commit 35ee1c3d78
README.md
a month ago
Reverse polarity of "elements" and "keys" (wrt to "sets" and "maps"). [davidrupp]
https://github.com/edn-format/edn/issues
Go to "https://github.com/edn-format/edn/issues"

edn example

```
{ :firstName "John"
  :lastName "Smith"
  :age 25
  :address {
    :streetAddress "21 2nd Street"
    :city "New York"
    :state "NY"
    :postalCode "10021" }
 :phoneNumber
    [ {:type "name" :number "212 555-1234"}
      {:type "fax" :number "646 555-4567" } ] }
```

type	examples
string	`"foo"`
character	`\f`
integer	`42, 42N`
floating point	`3.14, 3.14M`
boolean	`true`
nil	`nil`
symbol	`foo, +`
keyword	`:foo, ::foo`

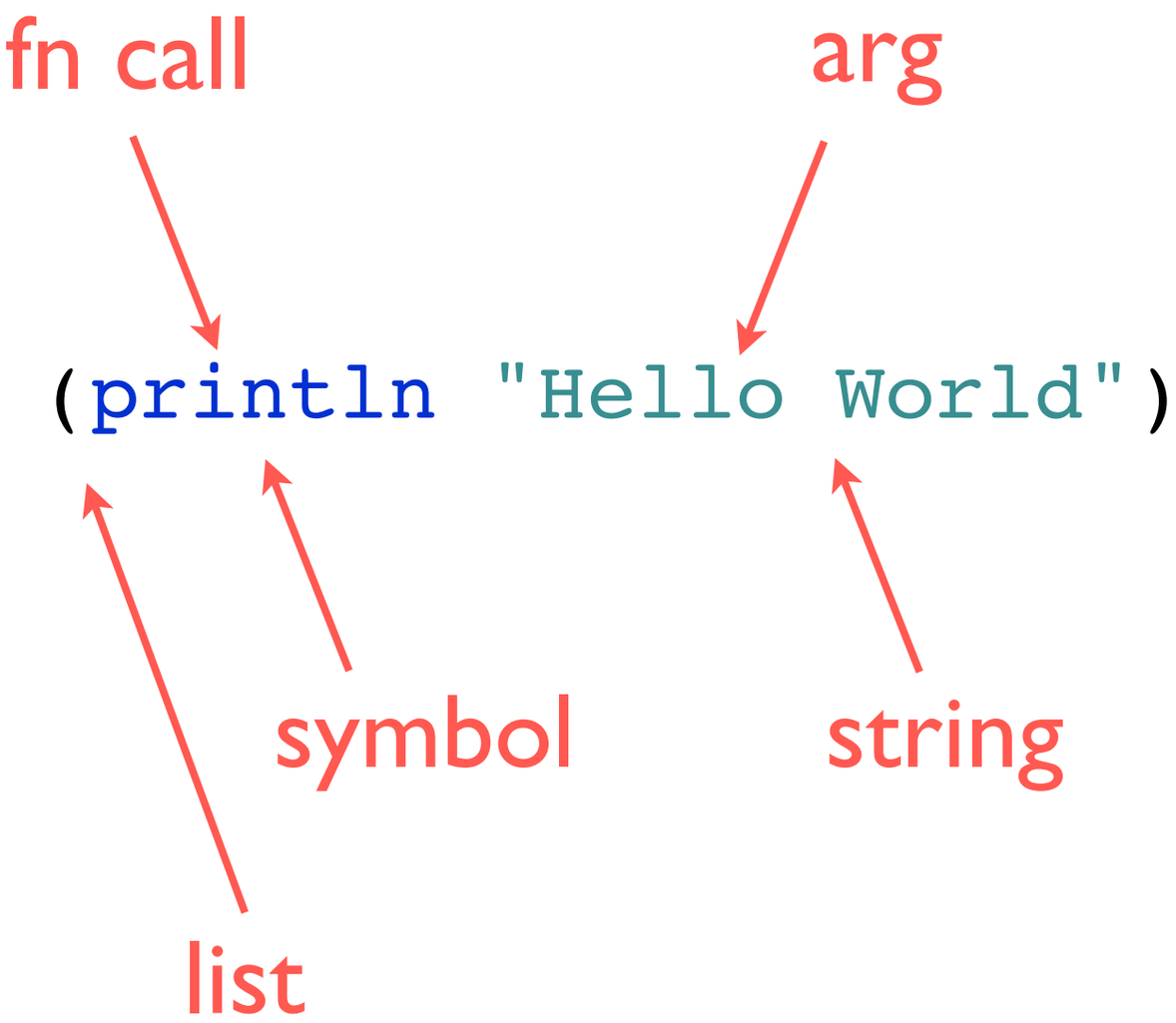
type	properties	examples
list	sequential	`(1 2 3)`
vector	sequential and random access	`[1 2 3]`
map	associative	`{:a 100` `:b 90}`
set	membership	`#{:a :b}`

program in data, not text

function call

semantics:

structure:



function def

define a fn

fn name

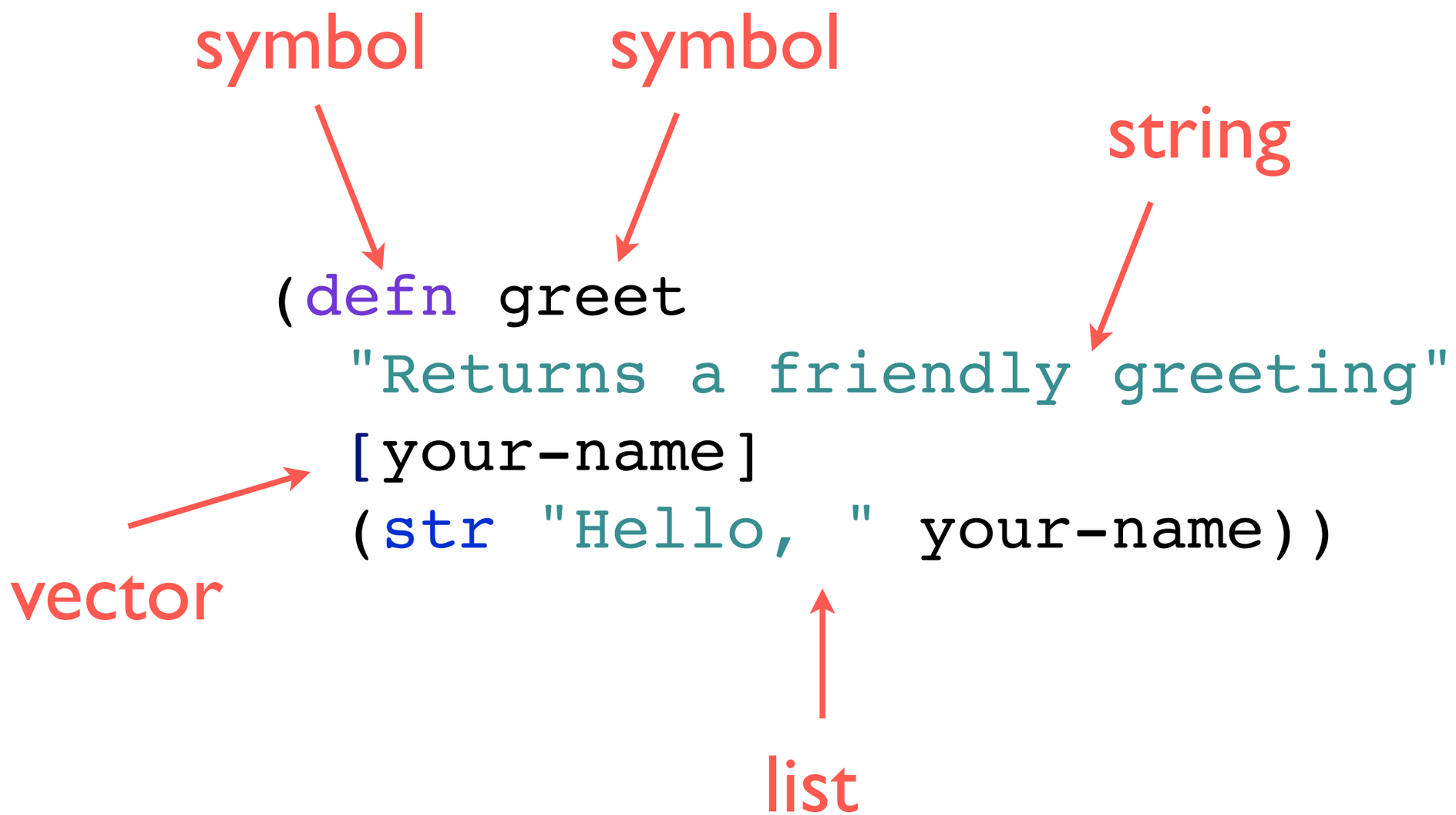
docstring

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

still just data



generic extensbility

#name edn-form

name describes interpretation of following element

recursively defined

all data can be literal

built-in tags

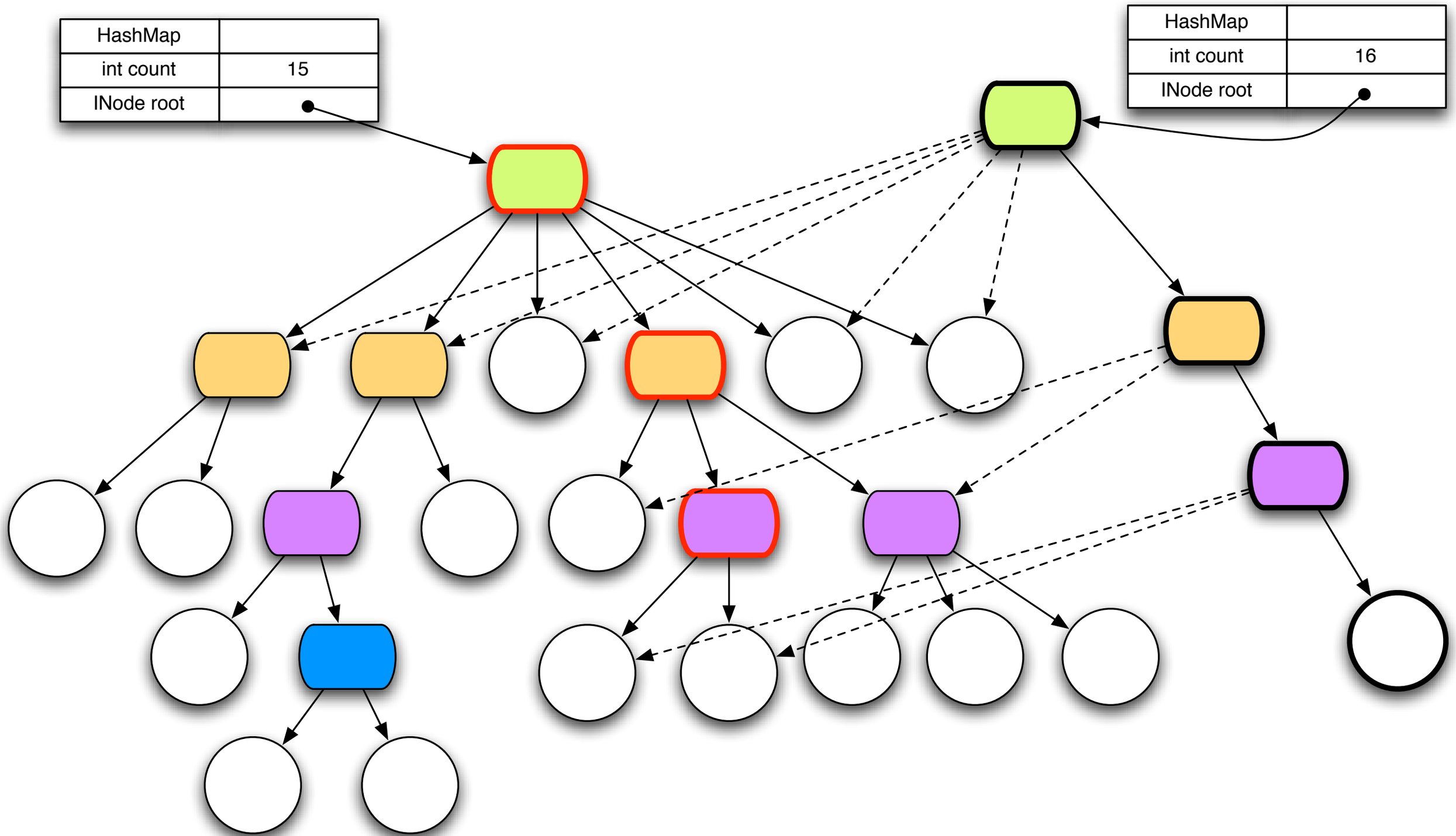
`#inst "rfc-3339-format"`

tagged element is a string in RFC-3339 format

`#uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"`

tagged element is a canonical UUID string

2. persistent data structures



persistent data structures

immutable

“change” by function application

maintain performance guarantees

full-fidelity old versions

transience vs. persistence

characteristic	transient	persistent
sharing	difficult	trivial
distribution	difficult	easy
concurrent access	difficult	trivial
access pattern	eager	eager or lazy
caching	difficult	easy
examples	Java, .NET collections relational databases NoSQL databases	Clojure, F# collections Datomic database

vectors

```
(def v [42 :rabbit [1 2 3]])

(v 1) -> :rabbit

(peek v) -> [1 2 3]

(pop v) -> [42 :rabbit]

(subvec v 1) -> [:rabbit [1 2 3]]
```

maps

```
(def m {:a 1 :b 2 :c 3})
```

```
(m :b) -> 2
```

```
(:b m) -> 2
```

```
(keys m) -> (:a :b :c)
```

```
(assoc m :d 4 :c 42) -> {:d 4, :a 1, :b 2, :c 42}
```

```
(dissoc m :d) -> {:a 1, :b 2, :c 3}
```

```
(merge-with + m {:a 2 :b 3}) -> {:a 3, :b 5, :c 3}
```

nested structure

```
(def jdoe {:name "John Doe",
           :address {:zip 27705, ...}})

(get-in jdoe [:address :zip])
-> 27705
```

```
(assoc-in jdoe [:address :zip] 27514)
-> {:name "John Doe", :address {:zip 27514}}

(update-in jdoe [:address :zip] inc)
-> {:name "John Doe", :address {:zip 27706}}
```

sets

```
(use clojure.set)
(def colors #{"red" "green" "blue"})
(def moods #{"happy" "blue"})

(disj colors "red")
-> #{"green" "blue"}

(difference colors moods)
-> #{"green" "red"}

(intersection colors moods)
-> #{"blue"}

(union colors moods)
-> #{"happy" "green" "red" "blue"}
```

3. unified succession model

in-place effects

subprograms are machines

programming: sticking together a bunch of moving parts

reasonable if memory is *very* (1970s) expensive

a better way: refs

new memories use new places

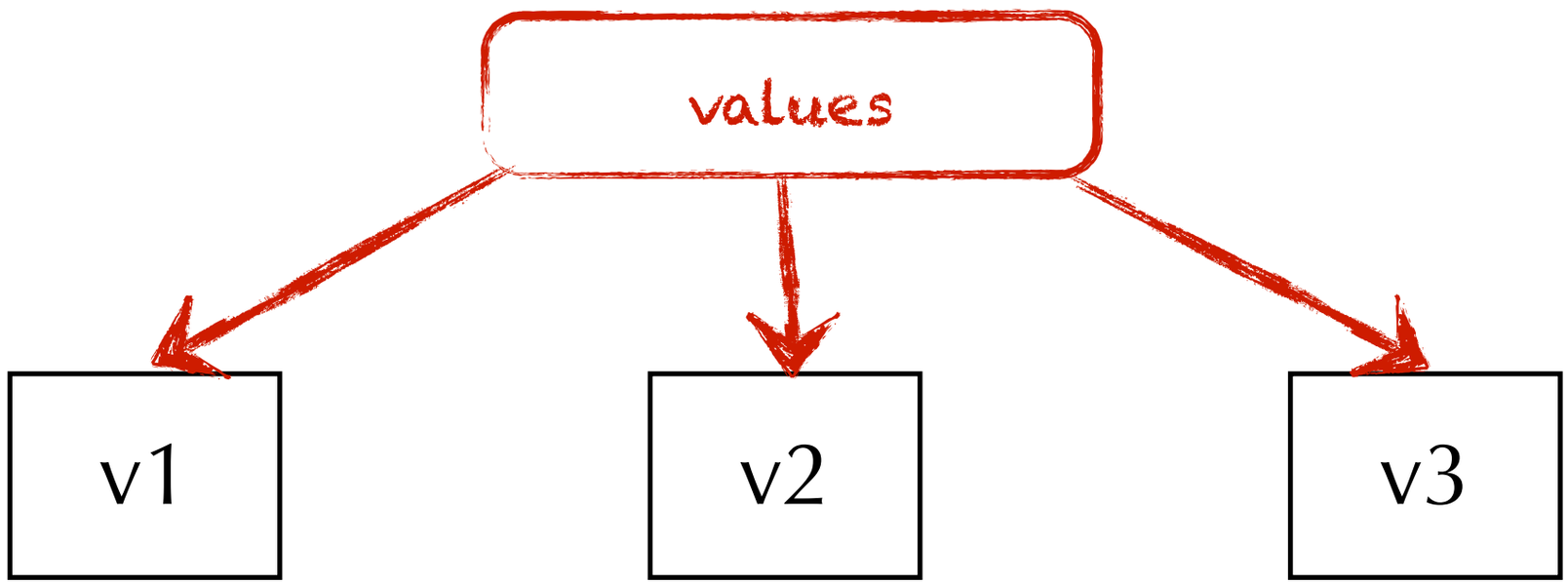
change encapsulated by constructors

references refer to point-in-time value

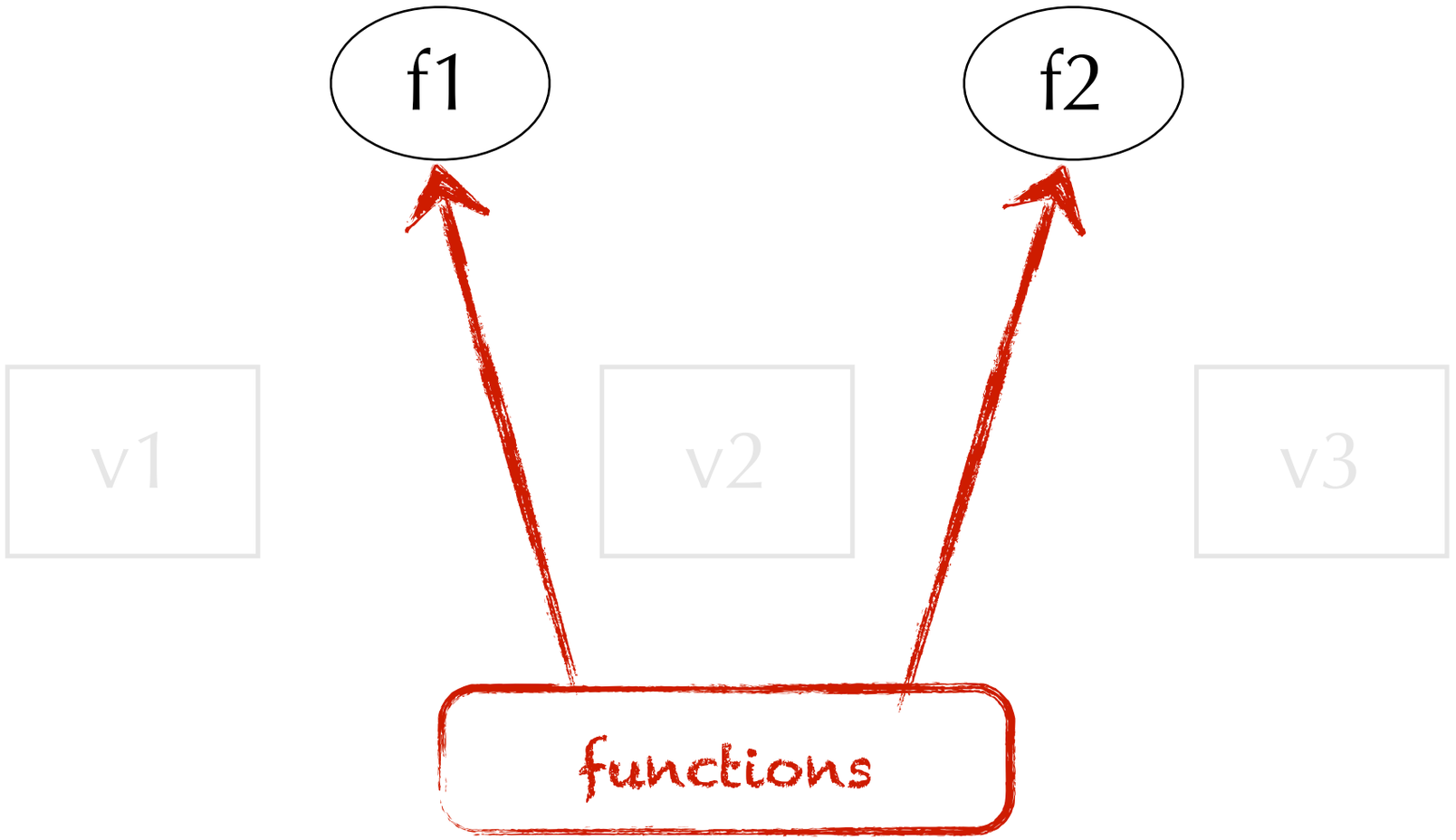
references see a *succession of values*

compatible with many update semantics

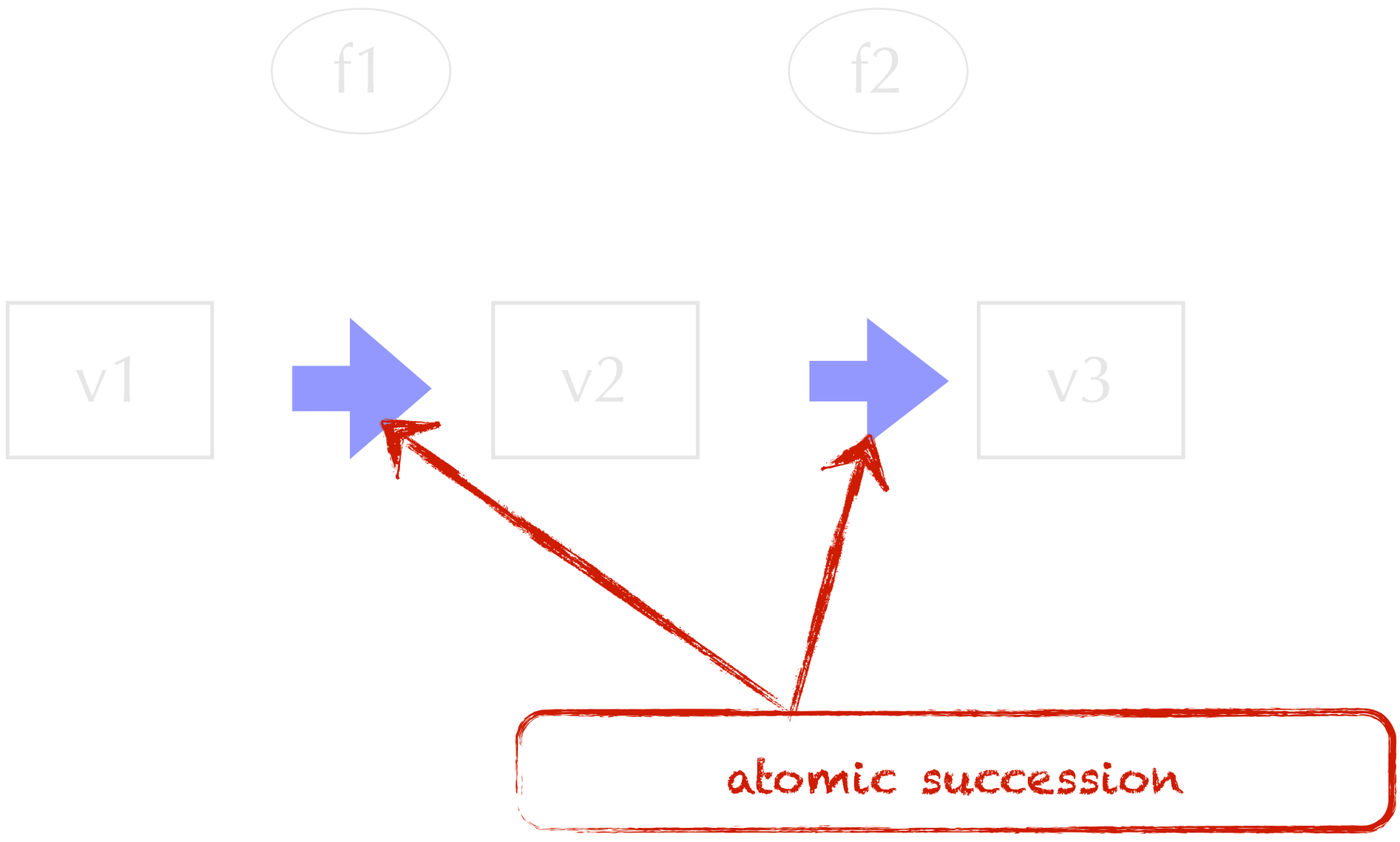
value succession



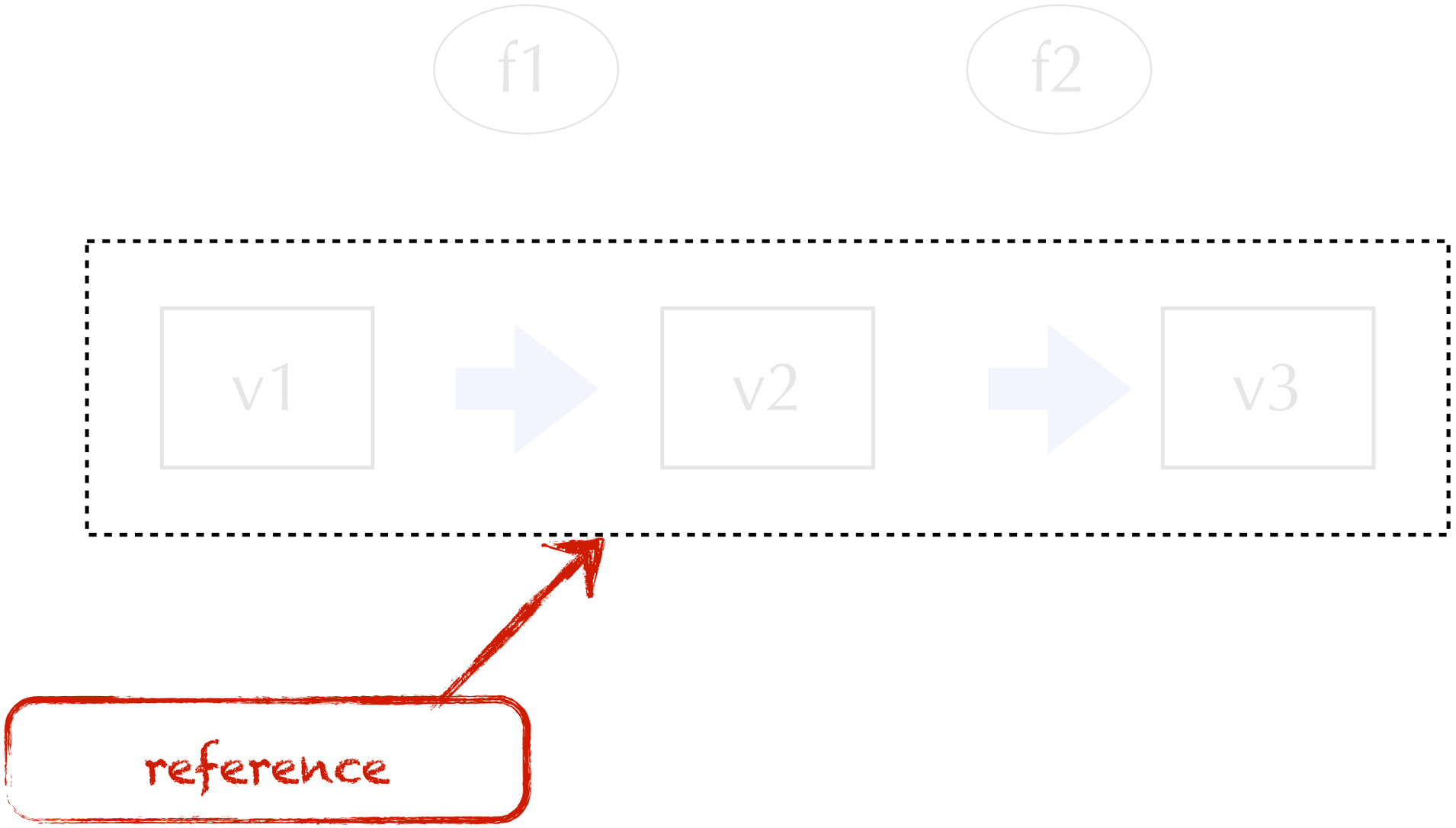
value succession



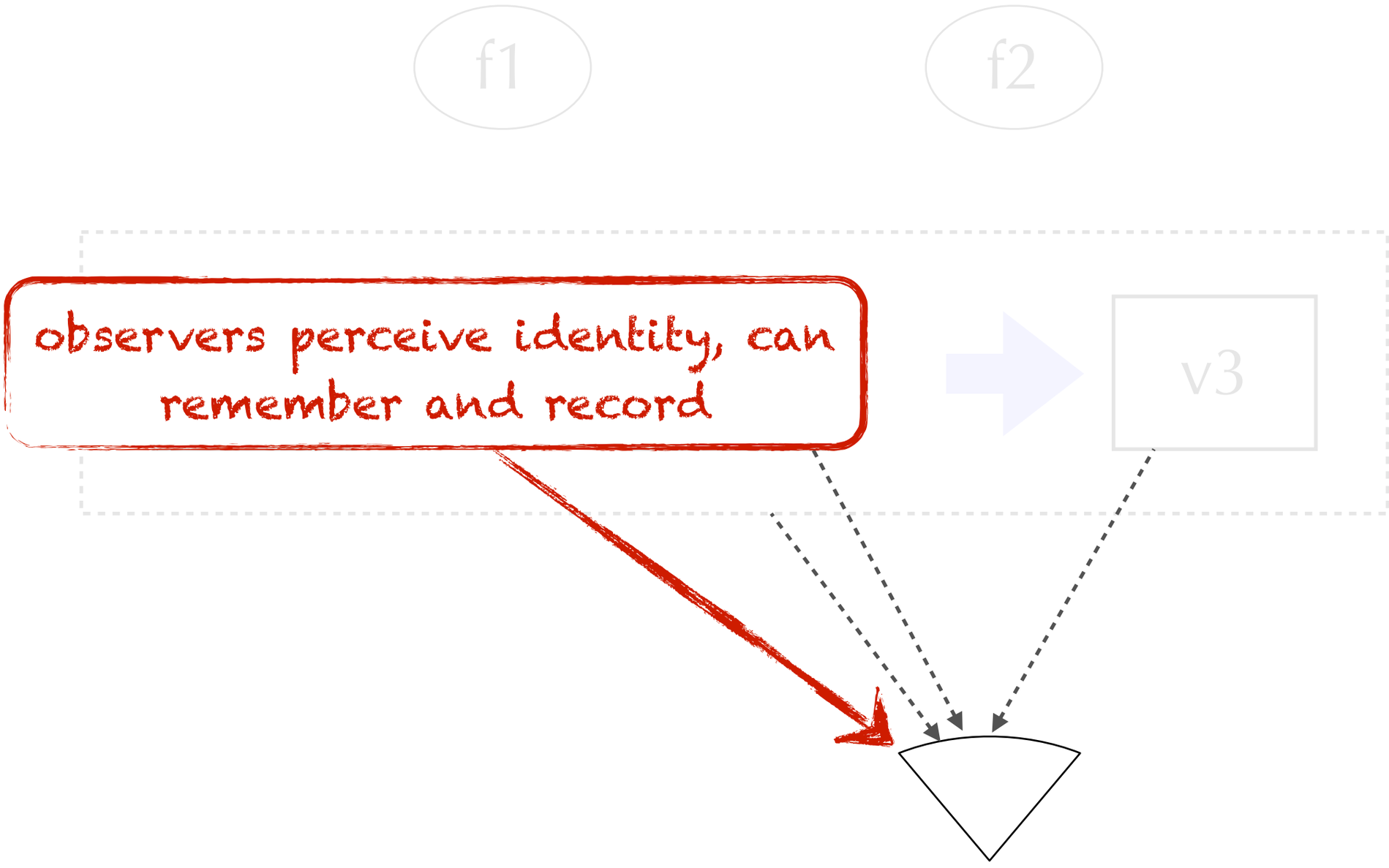
value succession



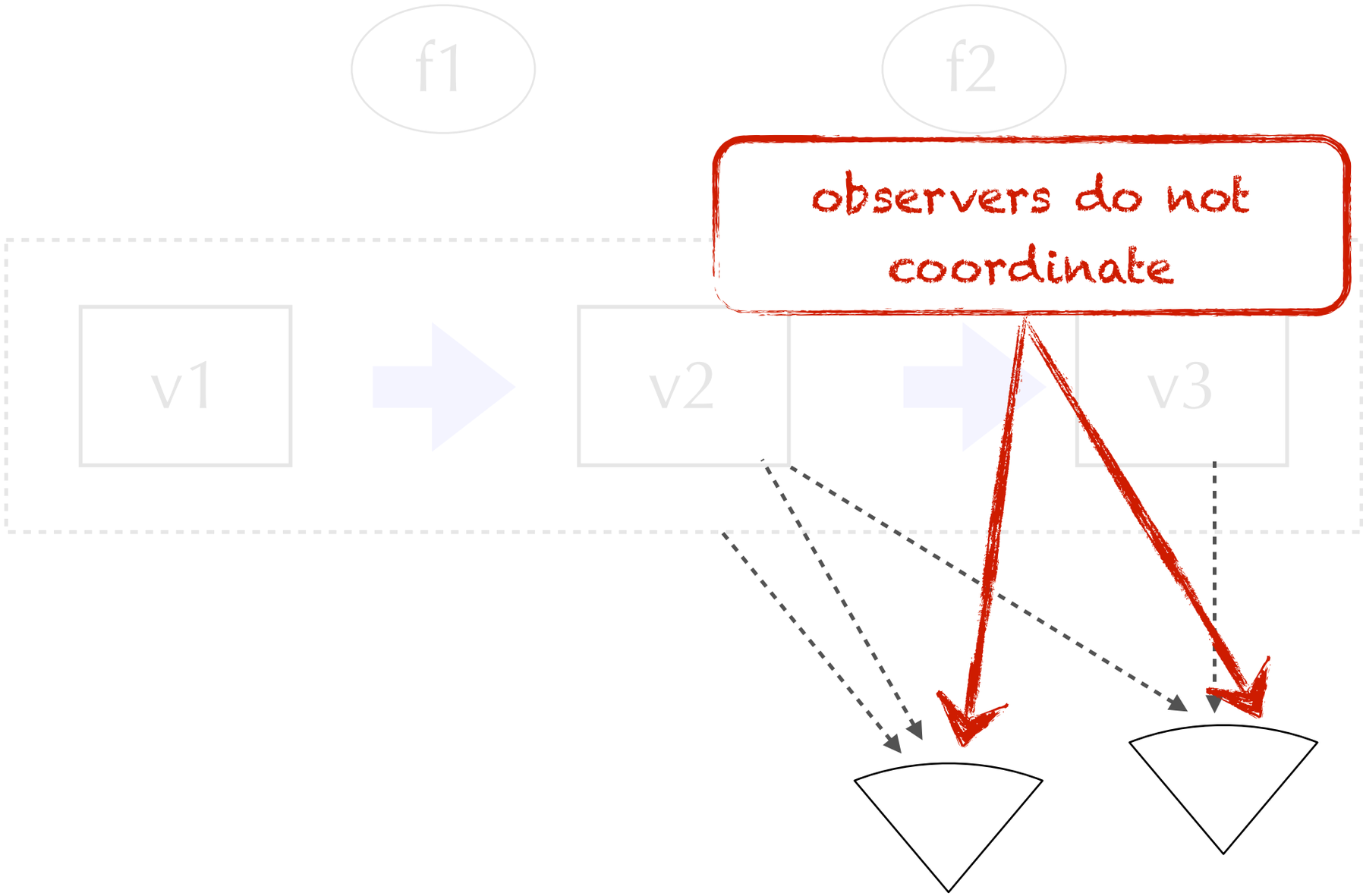
reference



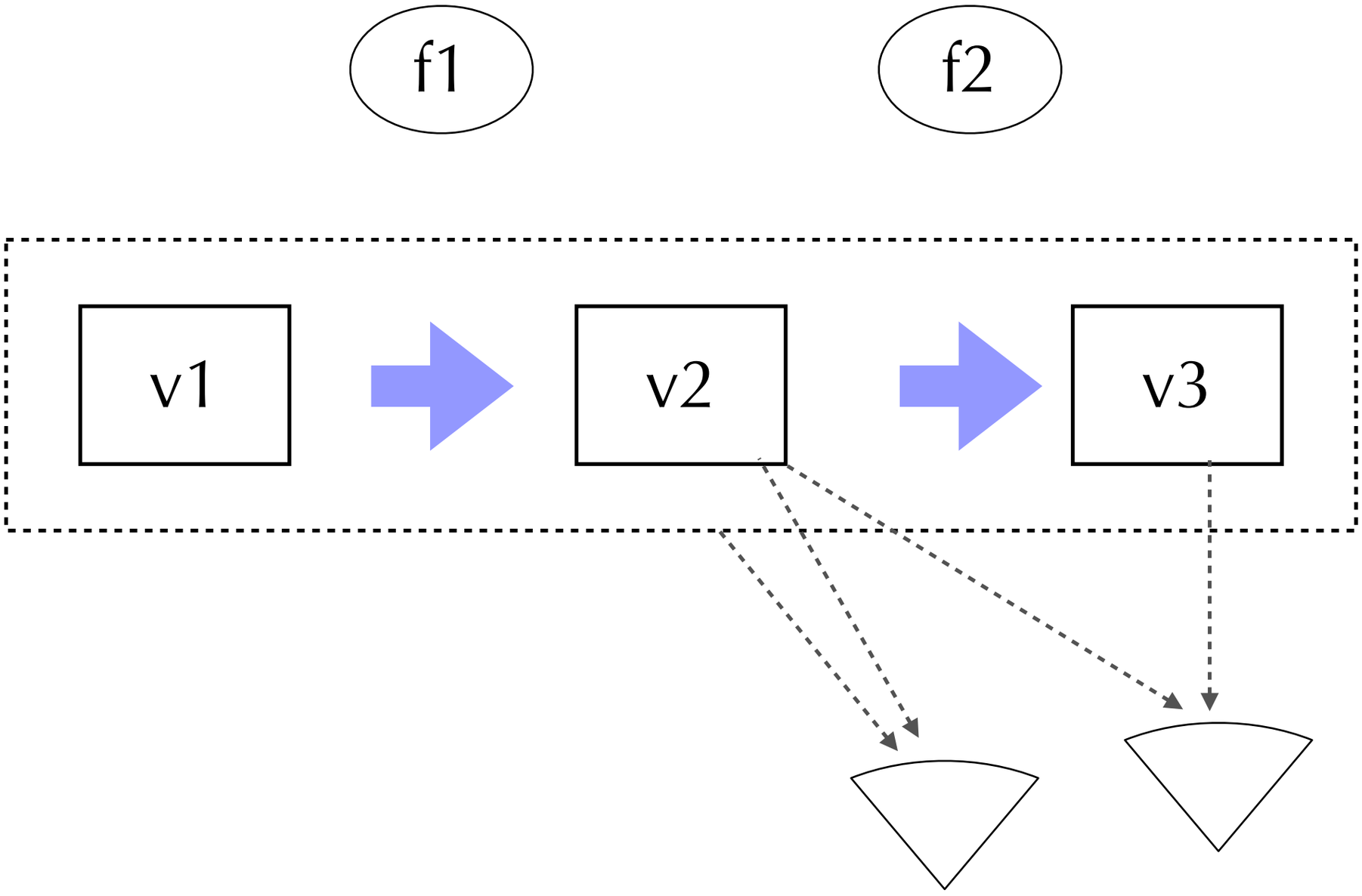
observers


f1
f2
observers perceive identity, can remember and record
v3

no coordination



unified succession model



atoms

```
(def counter  (atom 0))
(swap! counter + 10)
```

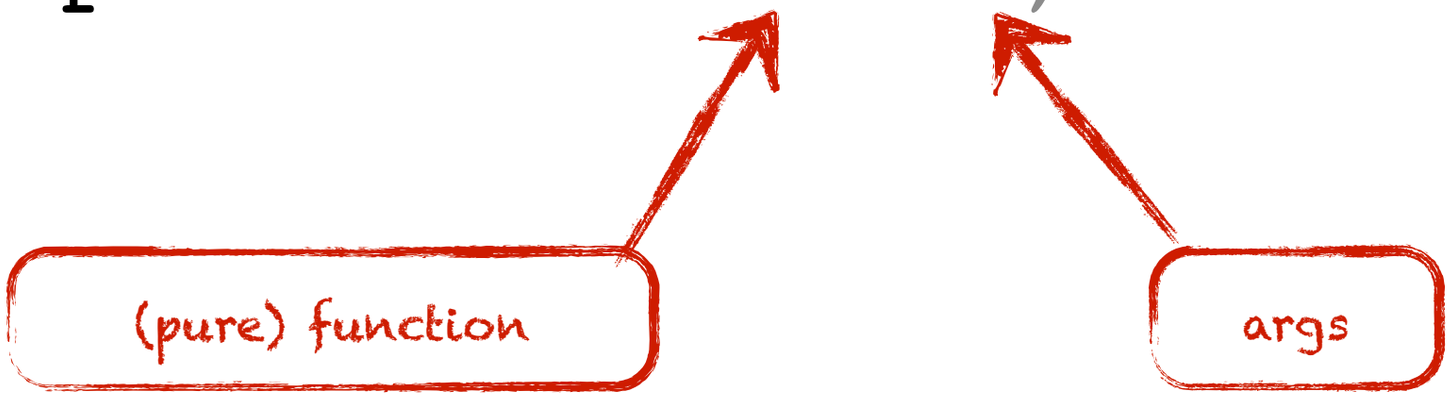
atoms

reference constructor

```
(def counter (atom 0))
(swap! counter + 10)
```

atoms

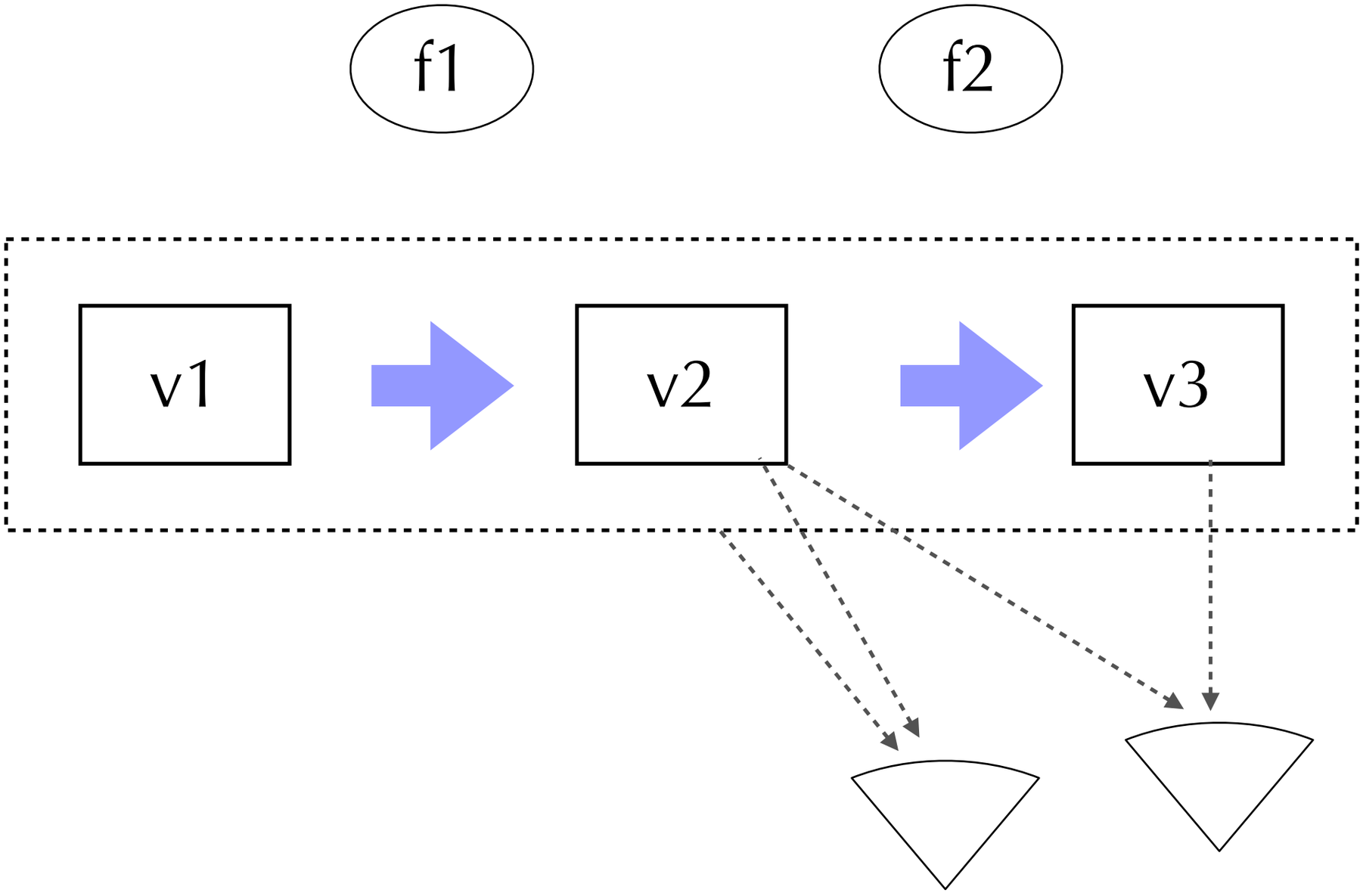
```
(def counter  (atom 0))
(swap! counter + 10)
```



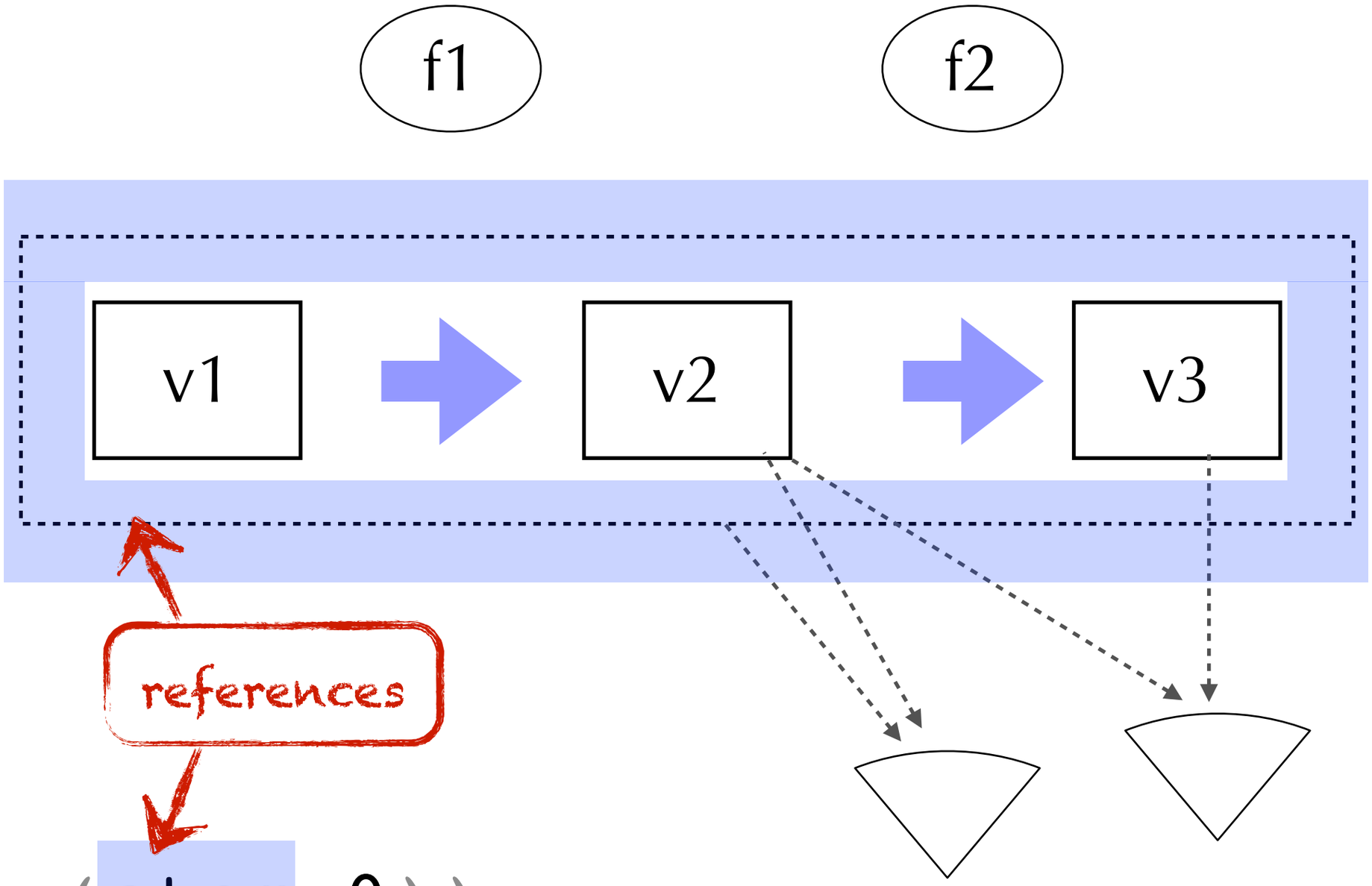
atoms

```
(def counter  (atom 0))
(swap! counter + 10)
```

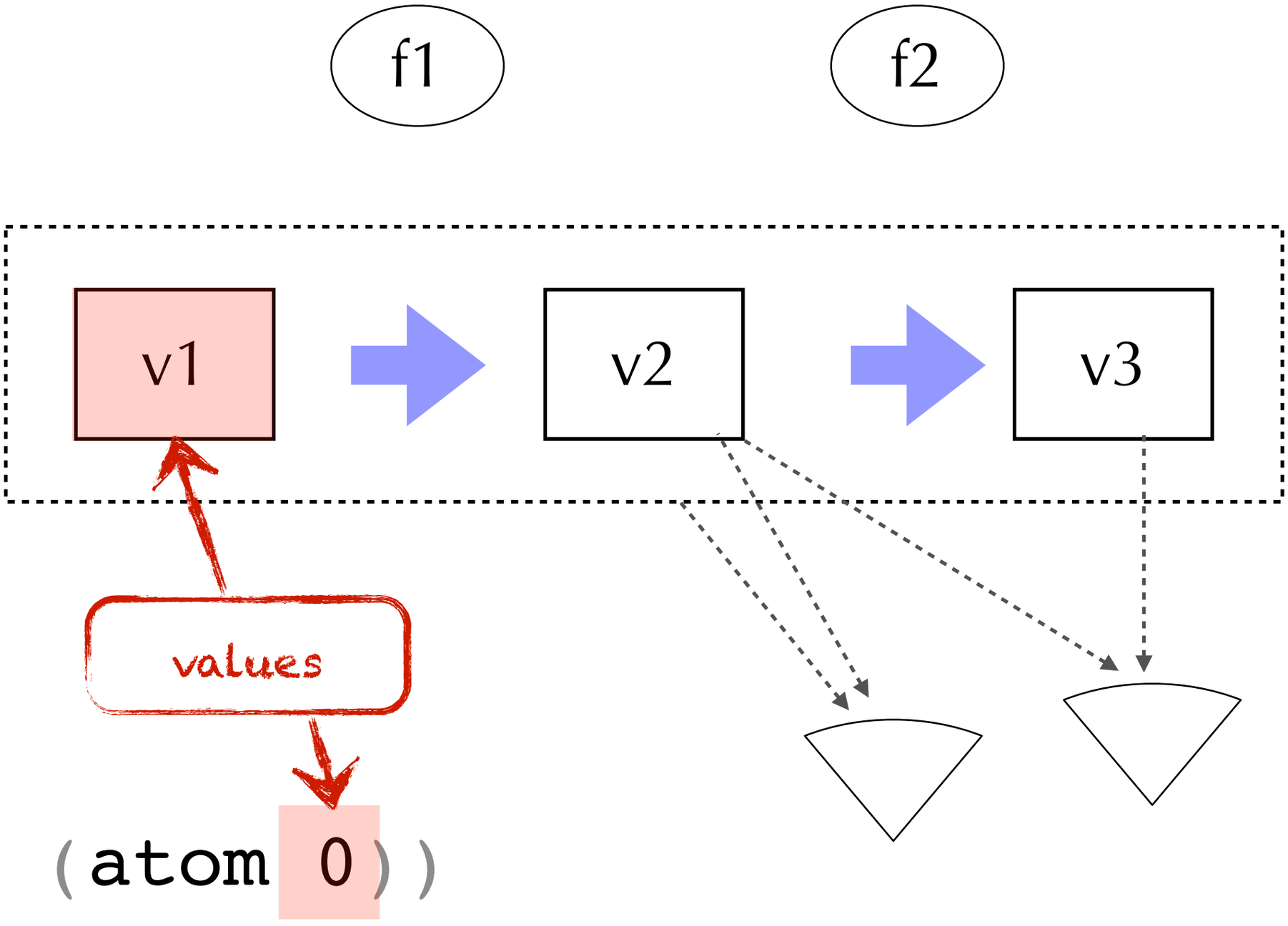
atoms



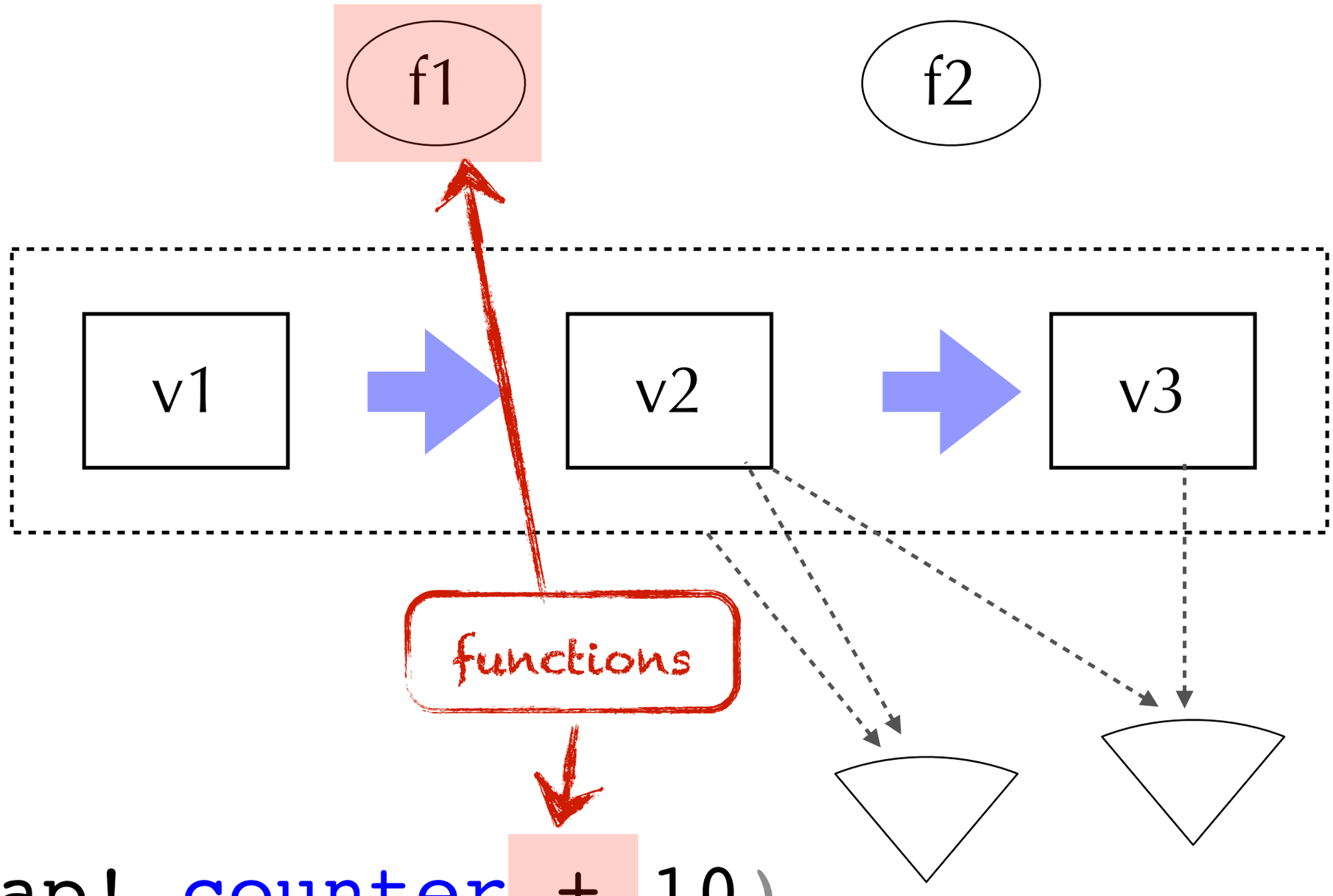
atoms



atoms

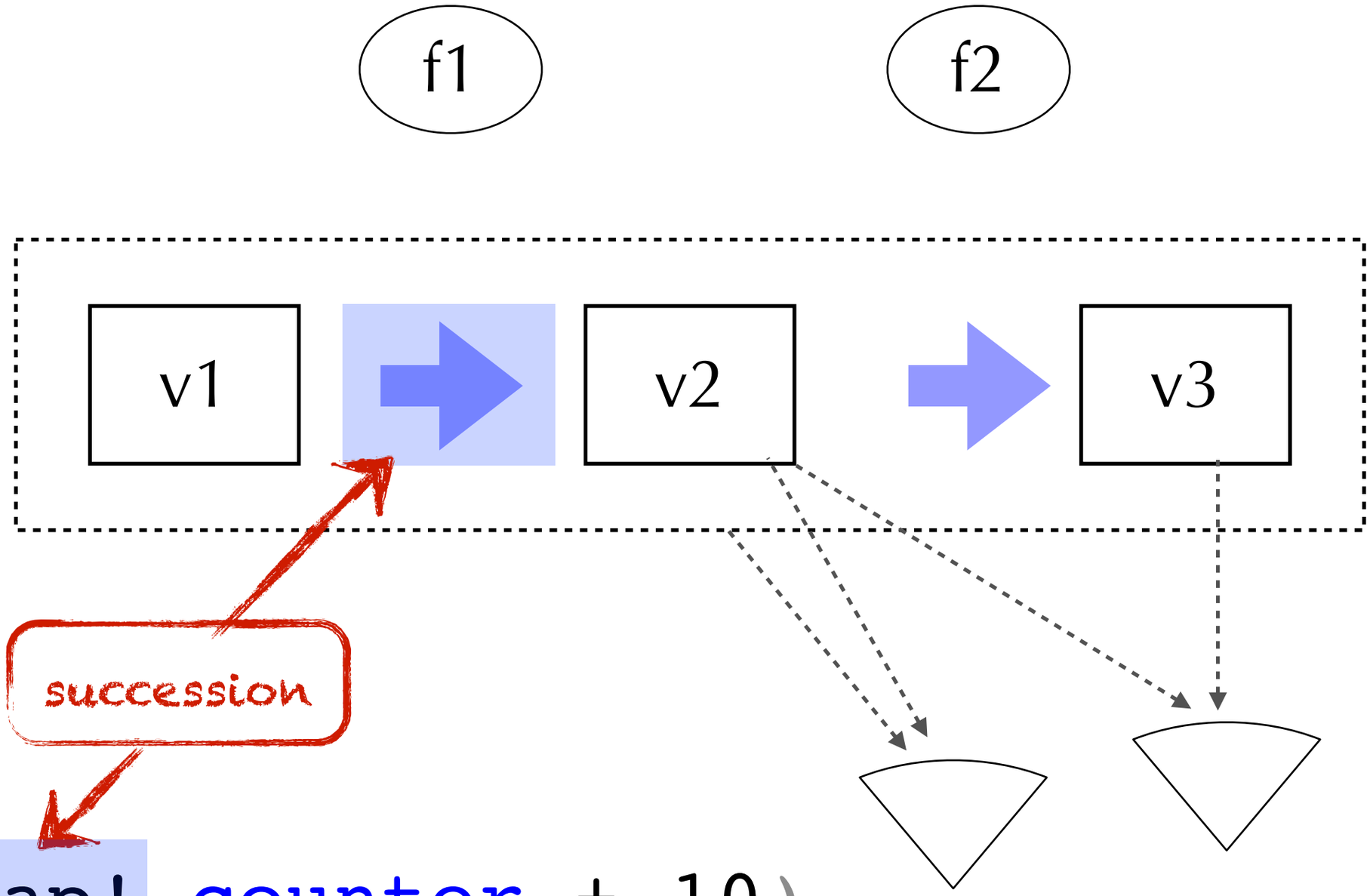


atoms



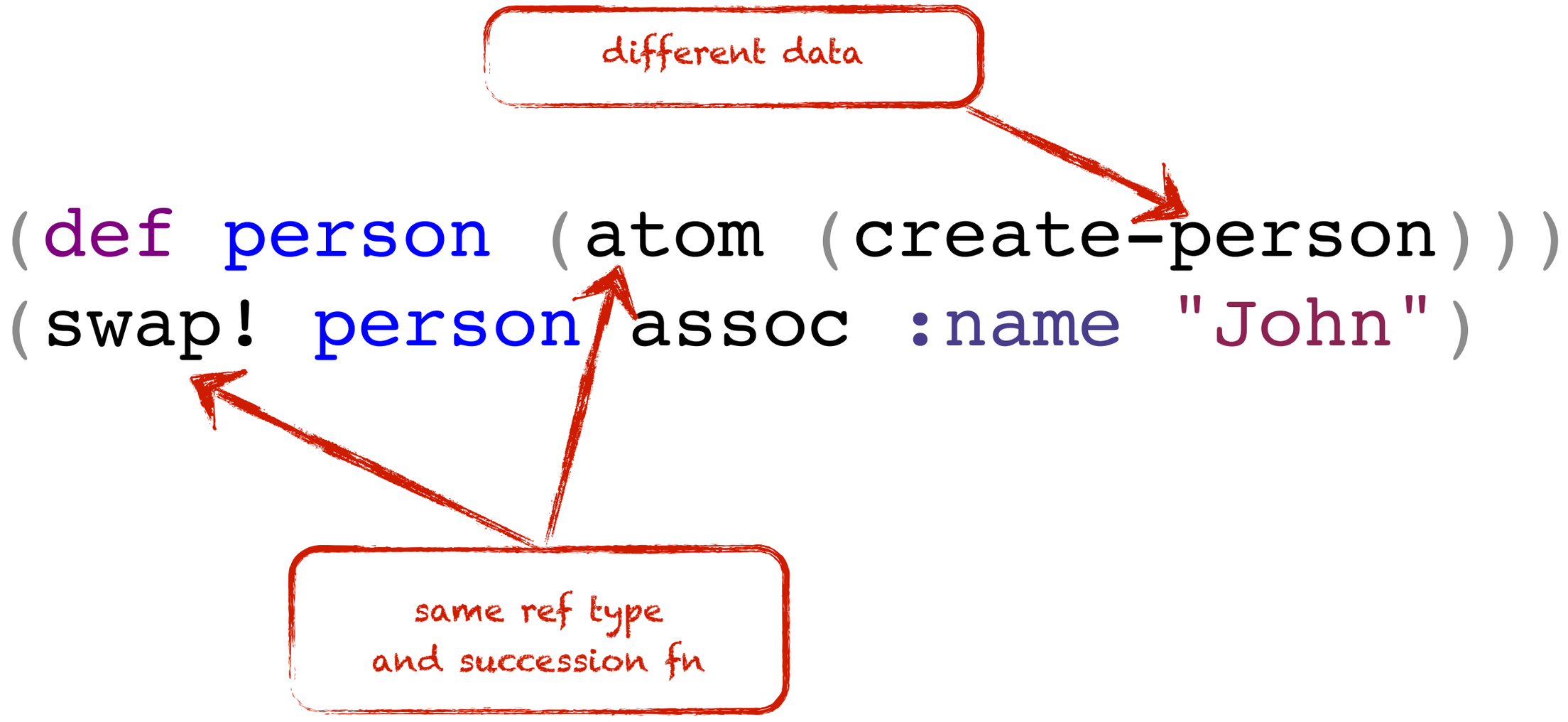

```
(swap! counter + 10)
```

atoms




```
(swap! counter + 10)
```

bigger structure



varying semantics

different kind of ref

```
(def number-later (promise))
(deliver number-later 42)
```

different succession

entire database

```
(def conn (d/connect uri)
(transact conn data)
```

entire database

```
(def conn (d/connect uri)
(transact conn data)
```

agent →

send	processor-derived pool
send-off	IO-derived pool
send-via	user-specified pool

atom ⇄

compare-and-set!	conditional
reset!	boring
swap!	functional transformation

connection

transact	⇄	ACID
transact-async	→	ACID

ref ⇄

alter	functional transformation
commute	commutative

var ⇄

alter-var-root	application config

var binding ⇄

binding, set!	dynamic, binding-local

4. sequences

first / rest /cons

```
(first [1 2 3])
-> 1

(rest [1 2 3])
-> (2 3)

(cons "hello" [1 2 3])
-> ("hello" 1 2 3)
```

take / drop

```
(take 2 [1 2 3 4 5])
-> (1 2)
```

```
(drop 2 [1 2 3 4 5])
-> (3 4 5)
```

predicates

```
(every? odd? [1 3 5])
-> true

(not-every? even? [2 3 4])
-> true

(not-any? zero? [1 2 3])
-> true

(some nil? [1 nil 2])
-> true
```

lazy and infinite

```
(set! *print-length* 5)
-> 5

(iterate inc 0)
-> (0 1 2 3 4 ...)

(cycle [1 2])
-> (1 2 1 2 1 ...)

(repeat :d)
-> (:d :d :d :d :d ...)
```

map / filter / reduce

```
(range 10)
-> (0 1 2 3 4 5 6 7 8 9)

(filter odd? (range 10))
-> (1 3 5 7 9)

(map odd? (range 10))
-> (false true false true false true
false true false true)

(reduce + (range 10))
-> 45
```

seqs work everywhere

collections

directories

files

XML

JSON

result sets

consuming JSON

What actors are in more than one movie currently topping the box office charts?

http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies

consuming JSON

```
find the JSON input
download it
parse json
walk the movies
accumulating cast
extract actor name
get frequencies
sort by highest frequency
```

http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies

consuming JSON

```
(->> box-office-uri
     slurp
     json/read-json
     :movies
     (mapcat :abridged_cast)
     (map :name)
     frequencies
     (sort-by (comp - second)))
```

http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies

consuming JSON

```
["Shiloh Fernandez" 2]
["Ray Liotta" 2]
["Isla Fisher" 2]
["Bradley Cooper" 2]
["Dwayne \"The Rock\" Johnson" 2]
["Morgan Freeman" 2]
["Michael Shannon" 2]
["Joel Edgerton" 2]
["Susan Sarandon" 2]
["Leonardo DiCaprio" 2]
```

http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies

5. protocols

protocols

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

polymorphic on type of first argument

no implementation

define fns in same namespace as protocol

55

implement protocols in-line

```
(deftype Bar [a b c]
  AProtocol
  (bar [this b] "Bar bar")
  (baz [this] (str "Bar baz " c)))
```

```
(def b (Bar. 5 6 7))

(baz b)

=> "Bar baz 7"
```


56

extending a protocol

```
(baz "a")

java.lang.IllegalArgumentException:
No implementation of method: :baz
of protocol: #'user/AProtocol
found for class: java.lang.String
```

```
(extend-type String
  AProtocol
  (bar [s s2] (str s s2))
  (baz [s] (str "baz " s)))

(baz "a")

=> "baz a"
```


57

extension options

extend to classes/interfaces: extend-type

extend to nil

extend multiple protocols: extend-type

extend to multiple types: extend-protocol

at bottom, arbitrary fn maps: extend

58

reify

instantiate an unnamed type

implement 0 or more protocols or interfaces

```
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

closes over environment like fn

59

interlude:

defrecord

defrecord

```
(defrecord Foo [a b c])
-> user.Foo

(def f (Foo. 1 2 3))
-> #'user/f

(:b f)
-> 2

(class f)
-> user.Foo

(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
 clojure.lang.IPersistentMap clojure.lang.IMeta java.lang.Object
 java.lang.Iterable clojure.lang.ILookup clojure.lang.Seqable
 clojure.lang.Counted clojure.lang.IPersistentCollection
 clojure.lang.Associative}
```

named type with slots

positional constructor

keyword access

plain ol' class

casydht*

***Clojure abstracts so you don't have to**

61

from maps...

```
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

```
(:lname stu)
=> "Halloway"
```

keyword access

```
(-> stu :address :city)
=> "Durham"
```

nested access

```
(assoc stu :fname "Stuart")
=> {:fname "Stuart", :lname "Halloway",
    :address ...}
```

update

```
(update-in stu [:address :zip] inc)
=> {:address {:street "200 N Mangum",
              :zip 27702 ...} ...}
```

nested update

62

...to records!

```
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                  (Address. "200 N Mangum"
                            "Durham"
                            "NC"
                            27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
```

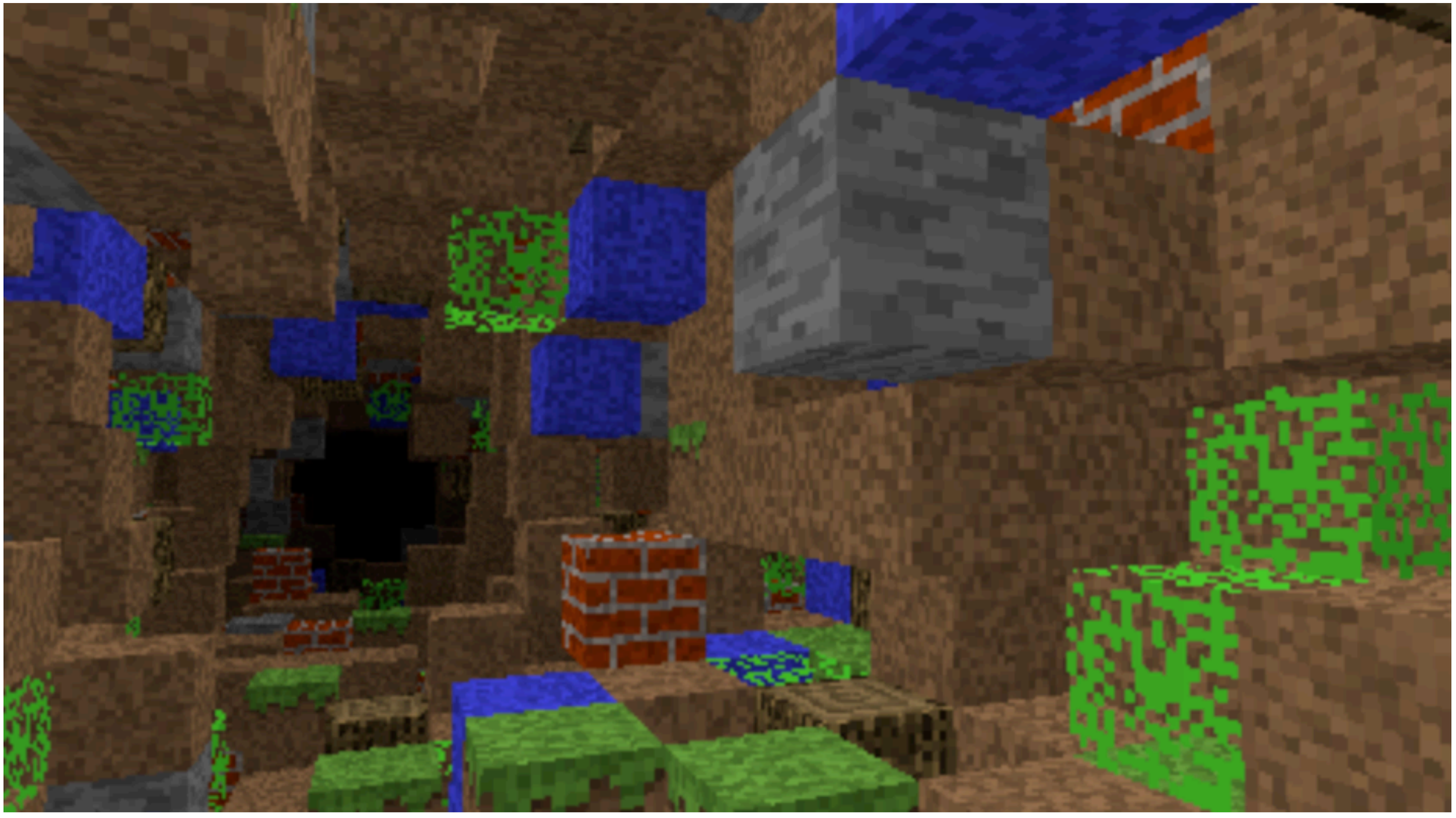
object-oriented

still data-oriented:
everything works as before

type is there when you care

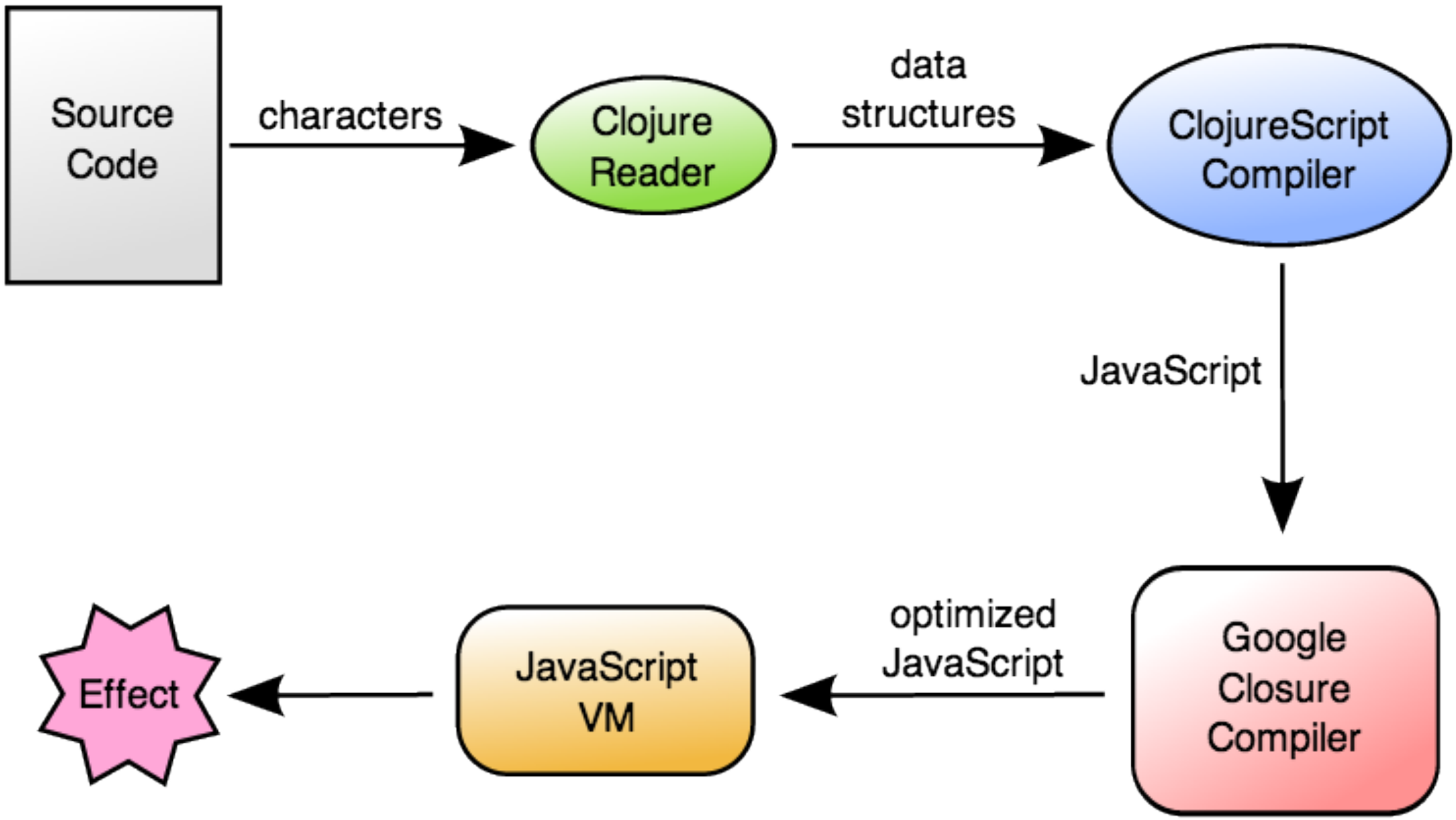
63

6. ClojureScript

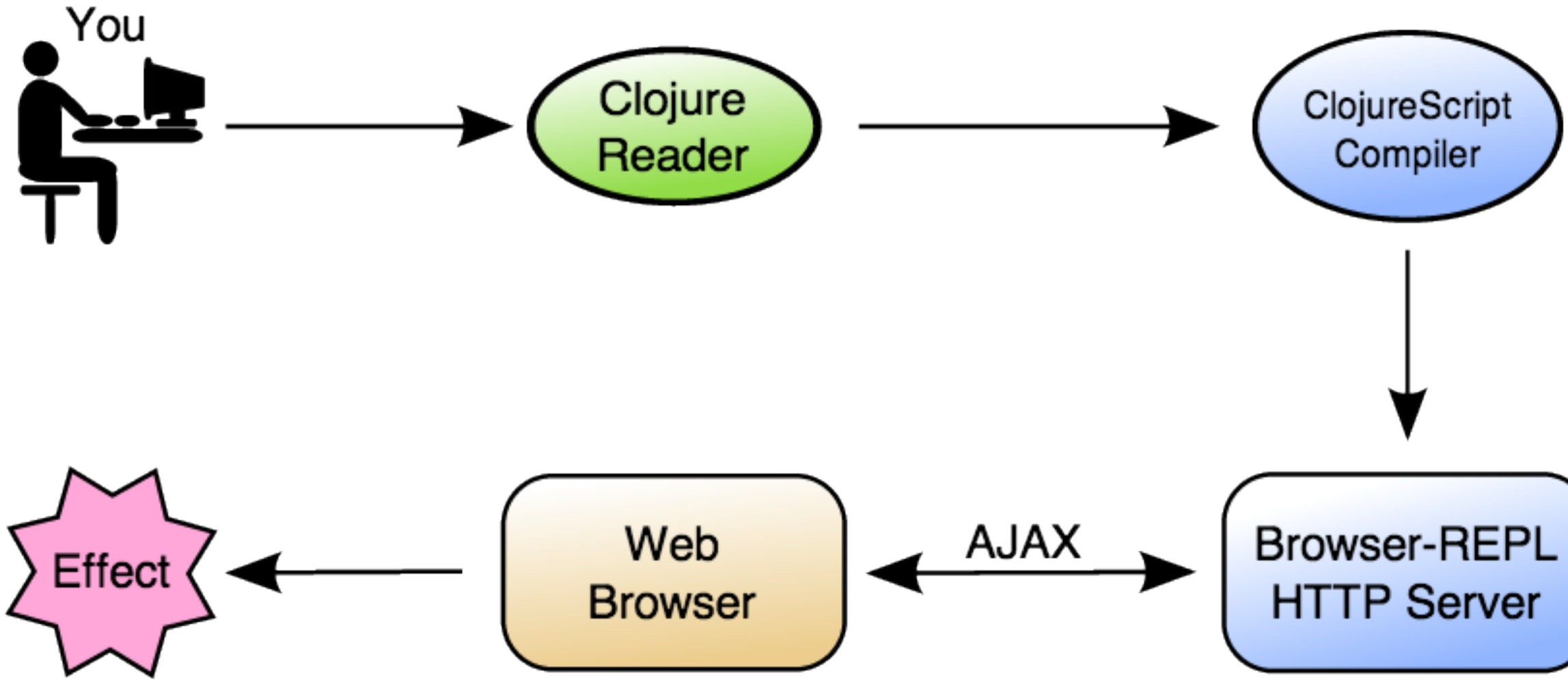


http://swannodette.github.io/2013/06/10/porting-notchs-minecraft-demo-to-clojurescript/

compilation pipeline



browser connected REPL



7. reducers

composing sequences

```
(->> apples
     (filter :edible?)
     (map #(dissoc % :sticker?))
     count)
```


68

reducing

```
(ns ...
  (:require
   [clojure.core.reducers :as r]))

(->> apples
     (r/filter :edible?)
     (r/map #(dissoc % :sticker?))
     (r/reduce counter))
```


69

folding

```
(ns ...
  (:require
   [clojure.core.reducers :as r]))

(->> apples
     (r/filter :edible?)
     (r/map #(dissoc % :sticker?))
     (r/fold counter))
```


70

8. core.logic


Scissors
smashes
cuts
Spock
Paper
disproves
decapitates
crushes
poisons
vaporizes
eats
covers
Lizard
Rock
crushes

logical approach

```
(defrel rps winner defeats loser)

(fact rps :scissors :cut :paper)
(fact rps :paper :covers :rock)
...
(fact rps :rock :breaks :scissors)

(run* [verb]
      (fresh [winner]
             (rps winner verb :paper)))
```

generic search

relation slots can be inputs or outputs

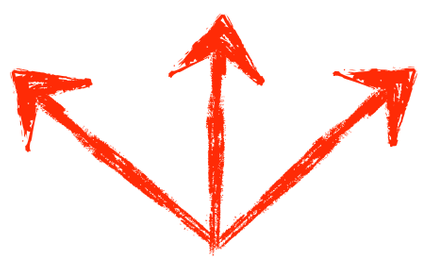
logical approach

```
(defrel rps winner defeats loser)

(fact rps :scissors :cut :paper)
(fact rps :paper :covers :rock)
...
(fact rps :rock :breaks :scissors)

(run* [winner]
      (fresh [verb loser]
             (rps winner verb loser)))
```

generic search



different bindings,
different query!

9. datalog

Datomic

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```

functional, lazy peers

pluggable storage protocol

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```

database is a lazily realized value, available to all peers equally

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```

query databases,
not connections

functional, lazy peers

```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```

join across databases,
systems, in-memory collections

functional, lazy peers

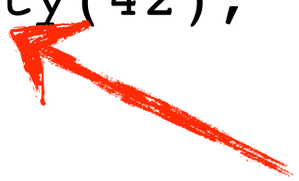
```
Connection conn =
connect("datomic:ddb://us-east-1/mb/mbrainz");

Database db = conn.db();

Set results = q(..., db);

Set crossDbResults = q(..., db1, db2);

Entity e = db.entity(42);
```



lazy, associative
navigable value

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);
```

information in generic data structures

```
allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

contains old db, new db, change

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

one possible future

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

ACID, serialized, time aware

```
List newData = ...;
Future<Map> f = conn.transactAsync(list);

dbBefore = conn.db.asOf(time);

possibleFuture = db.with(...);

allTime = db.history();

BlockingQueue<Map> queue = conn.txReportQueue();

Log log = conn.log();
Iterable<Map> it = log.txRange(startOfMonth, null);
```

review any time range

example database

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

data pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

data pattern

Constrains the results returned,
binds variables

```
[?customer :email ?email]
```

entity attribute value

data pattern

Constrains the results returned,
binds variables

constant

```
[?customer :email ?email]
```

data pattern

Constrains the results returned,
binds variables

variable variable

```
[?customer :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[?customer :email ?email]
```

constants anywhere

"Find a particular customer's email"

```
[42 :email ?email]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 :email ?email]
```

variables anywhere

"What attributes does
customer 42 have?

```
[42 ?attribute]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute]
```

variables anywhere

"What attributes and values does customer 42 have?

```
[42 ?attribute ?value]
```

entity	attribute	value
42	:email	jdoe@example.com
43	:email	jane@example.com
42	:orders	107
42	:orders	141

```
[42 ?attribute ?value]
```

where clause

data pattern

```
[:find ?customer
 :where [?customer :email]]
```

find clause

variable to return

```
[:find ?customer
 :where [?customer :email]]
```

implicit join

"Find all the customers who
have placed orders."

```
[:find ?customer
 :where [?customer :email]
        [?customer :orders]]
```

api

```
import static datomic.Peer.q;

q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

q

```
import static datomic.Peer.q;

q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

query

```
import static datomic.Peer.q;


q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

inputs

```
import static datomic.Peer.q;


q("[:find ?customer
  :where [?customer :id]
         [?customer :orders]]",
db);
```

in clause

Names inputs so you can refer to them elsewhere in the query

```
:in $database ?email
```

parameterized query

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

first input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

second input

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

verbose?

"Find a customer by email."

```
q([:find ?customer
   :in $database ?email
   :where [$database ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

shortest name possible

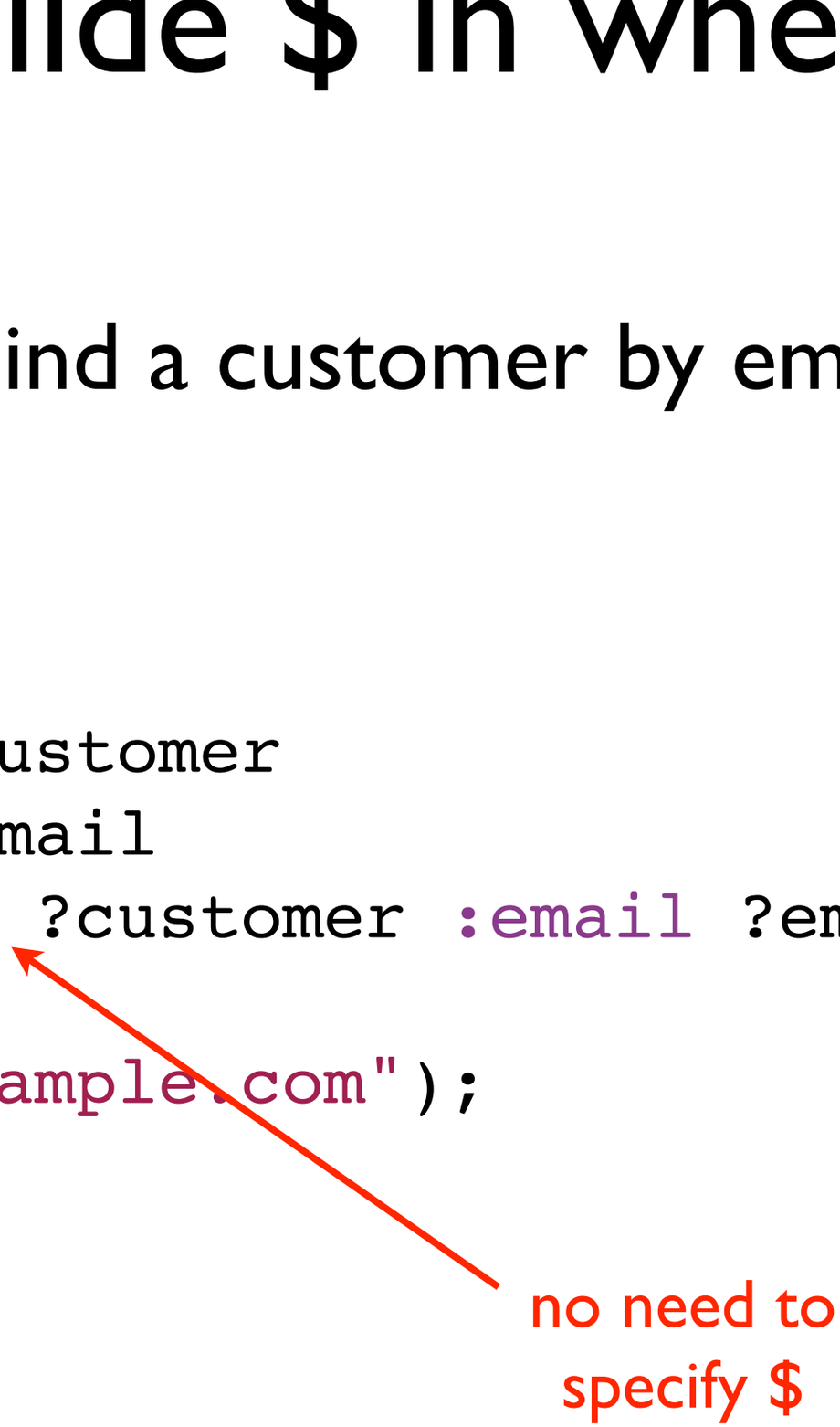
"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [$ ?customer :email ?email]],
  db,
  "jdoe@example.com");
```

elide $ in where

"Find a customer by email."

```
q([:find ?customer
   :in $ ?email
   :where [ ?customer :email ?email]],
  db,
  "jdoe@example.com");
```



predicates

Functional constraints that can appear in a :where clause

```
[(< 50 ?price)]
```

adding a predicate

"Find the expensive items"

```
[:find ?item
 :where [?item :item/price ?price]
        [(< 50 ?price)]]
```

functions

Take bound variables as inputs
and bind variables with output

```
[(shipping ?zip ?weight) ?cost]
```

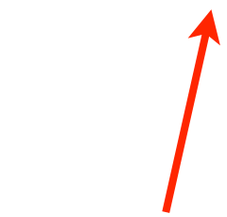
function args

```
[(shipping ?zip ?weight) ?cost]
```

bound inputs

function returns

```
[(shipping ?zip ?weight) ?cost]
```



calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

navigate from customer to zip

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

get product facts needed *during query*

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

call web service to bind shipCost

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

byo functions

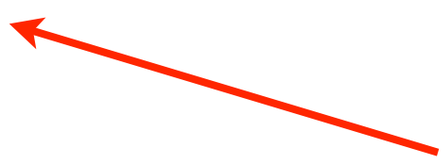
Functions can be plain JVM code.

```
public class Shipping {
  public static BigDecimal
  estimate(String zip1, int pounds);
}
```

calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```



constrain price

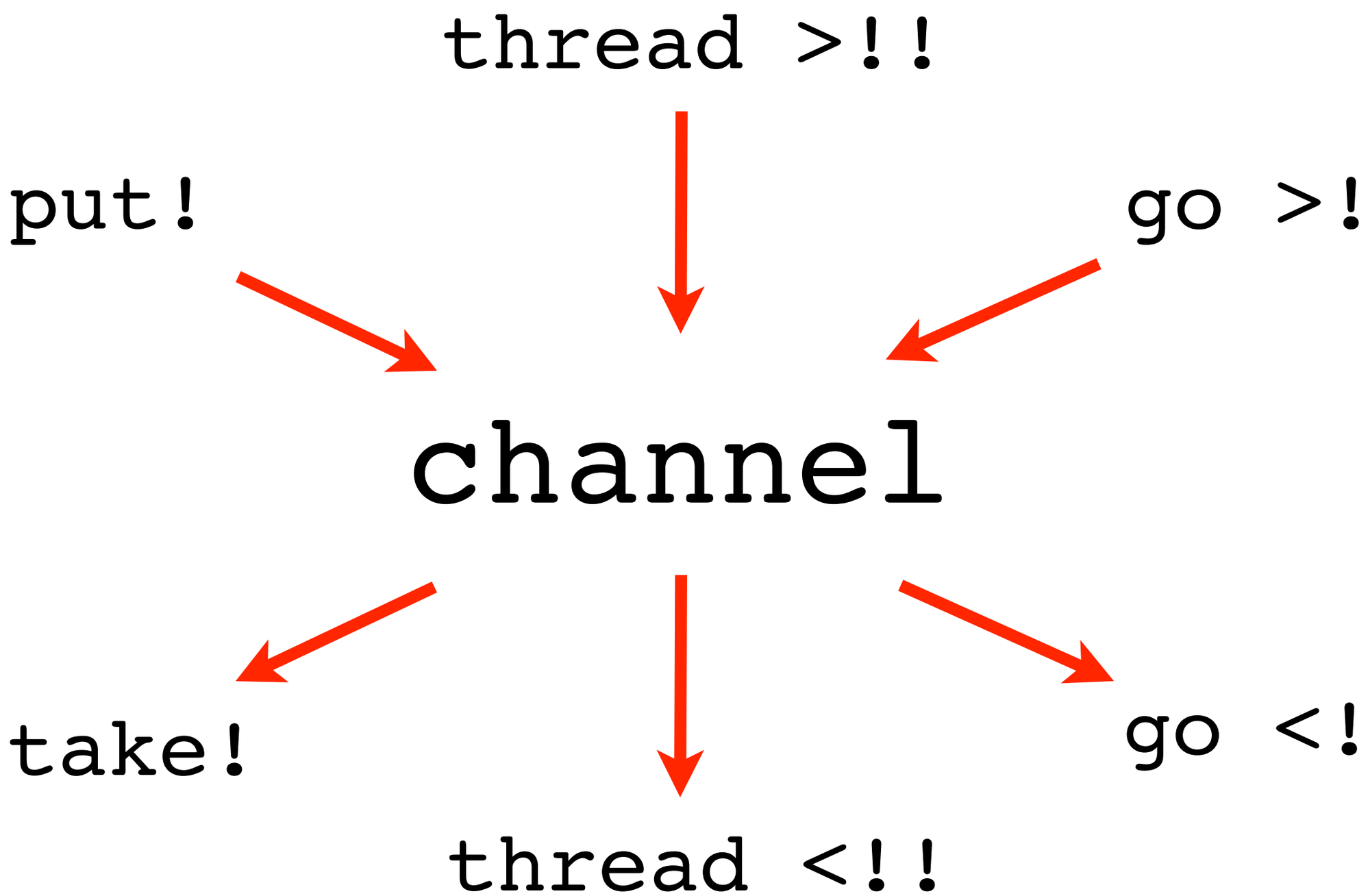
calling a function

"Find me the customer/product combinations where the shipping cost dominates the product cost."

return customer, product pairs

```
[:find ?customer ?product
 :where [?customer :shipAddress ?addr]
        [?addr :zip ?zip]
        [?product :product/weight ?weight]
        [?product :product/price ?price]
        [(Shipping/estimate ?zip ?weight) ?shipCost]
        [(<= ?price ?shipCost)]]
```

10. core.async



running in the browser

```
(go (while true (<! (timeout 250)) (>! c 1)))
(go (while true (<! (timeout 1000)) (>! c 2)))
(go (while true (<! (timeout 1500)) (>! c 3)))
```

channel put

IOC 'thread"

```
(let [out (by-id "ex0-out")]
  (go (loop [results []]
        (set-html out (render results))
        (recur (-> (conj results (<! c)) (peekn 10))))))
```

channel get

alt(*)

wait on multiple channel operations

puts, takes, timeouts

compare unix select

works with threads *or go blocks*

alt!, alt!!

choice of takes

choice of puts

```
(alt!
  [c t]          ([val ch] (foo ch val))
  x              ([v] v)
  [[out val]] :wrote
  :default       42)
```

single "put" op

[value, channel] taken

130

search with SLA

```
(defn search [query]
  (let [c (chan)
        t (timeout 80)]
    (go (>! c (<! (fastest query web1 web2))))
    (go (>! c (<! (fastest query image1 image2))))
    (go (>! c (<! (fastest query video1 video2))))
    (go (loop [i 0
               ret []]
          (if (= i 3)
            ret
            (recur (inc i)
                   (conj ret (alt! [c t] ([v] v)))))))))
```

coordinates all searches and shared timeout

http://talks.golang.org/2012/concurrency.slide#50

131

protocols

targeting
platforms

immutability

seqs

refs

reducers

core.async

edn

datalog

core.logic

resources

Clojure
http://clojure.com. The Clojure language.
http://tryclj.com/. Try Clojure.
http://himera.herokuapp.com. Try ClojureScript.
http://thinkrelevance.com/blog/tags/podcast. The Cognicast.
http://www.datomic.com/. Datomic.
http://clojure.in/. Planet Clojure.
http://pragprog.com/book/shcloj2/programming-clojure. *Programming Clojure*.

@stuarthalloway
https://github.com/stuarthalloway/presentations/wiki. Presentations
http://www.linkedin.com/pub/stu-halloway/0/110/543/
https://twitter.com/stuarthalloway
mailto:stu@cognitect.com

cognitect

relevance

Datomic