Clojure

core.async

@stuarthalloway
stu@cognitect.com

the problems

objects make terrible machines

function chains make poor machines

direct-connect relationships

callback hell

j.u.c queues block real threads

threads are expensive and/or nonexistent

the opportunity

first class conveyance (queue-like)

processes

multi reader/writer

platform integration (JVM, browser)

robust subsystems

Communicating Sequential Processes (CSP)

first class processes

first class channels

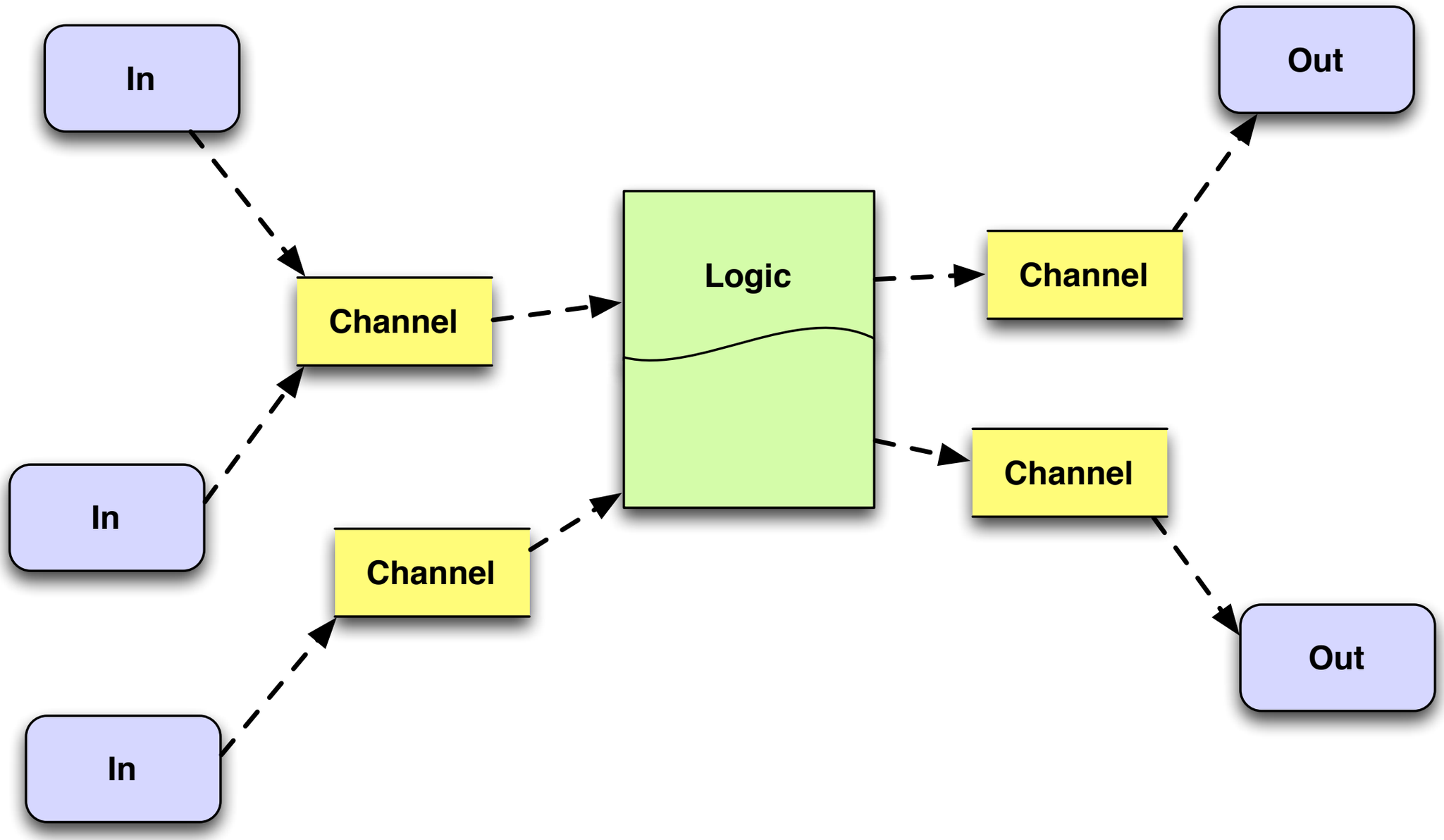
concurrency primitive (coordination)

coherent sequential logic

multi reader/writer

buffering

core.async


In
In
In
Channel
Channel
Channel
Logic
Channel
Channel
Out
Out

first-class processes

```
(go
```

IOC 'thread',
state machine,
parking

```
)
```

```
(thread
```

real thread,
blocking

```
)
```

first-class channels

op	go	thread	*(external)*
put	`(>! ch val)`	`(>!! ch val)`	`(put! ch val)`
take	`(<! ch)`	`(<!! ch)`	`(take! ch)`

single-operation examples

http://go.cognitect.com/core_async_webinar_recording

https://github.com/cognitect/async-webinar examples 1-5

"Hello World"

"Hello World"

Everything is Data

```
{ :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" } ] }
```

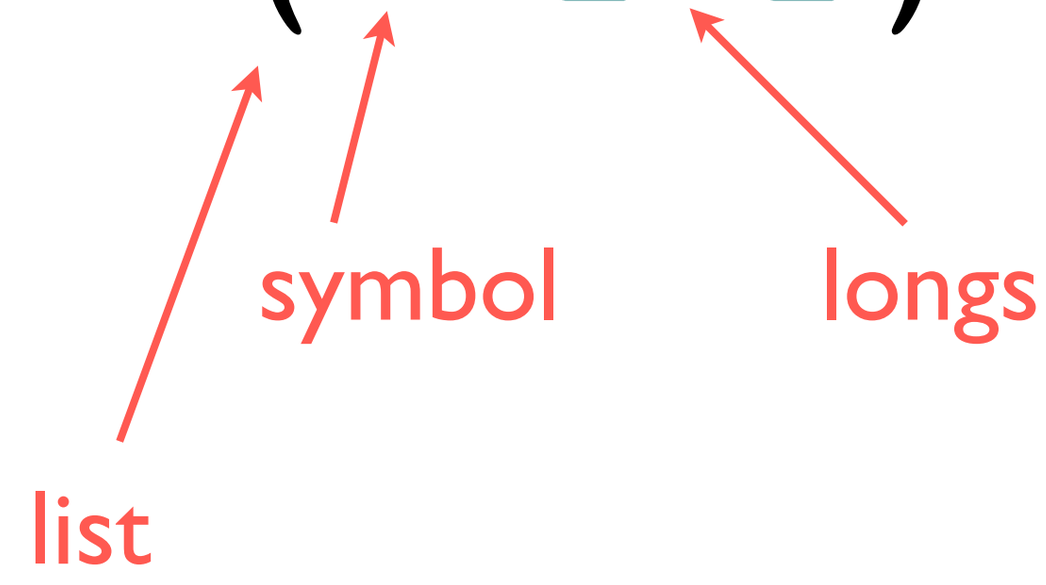
type	examples
string	`"foo"`
character	`\f`
integer	`42, 42N`
floating point	`3.14, 3.14M`
boolean	`true`
nil	`nil`
symbol	`foo, +`
keyword	`:foo, ::foo`

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}`

Function Call

semantics:

structure:



Function Definition

define a fn

fn name

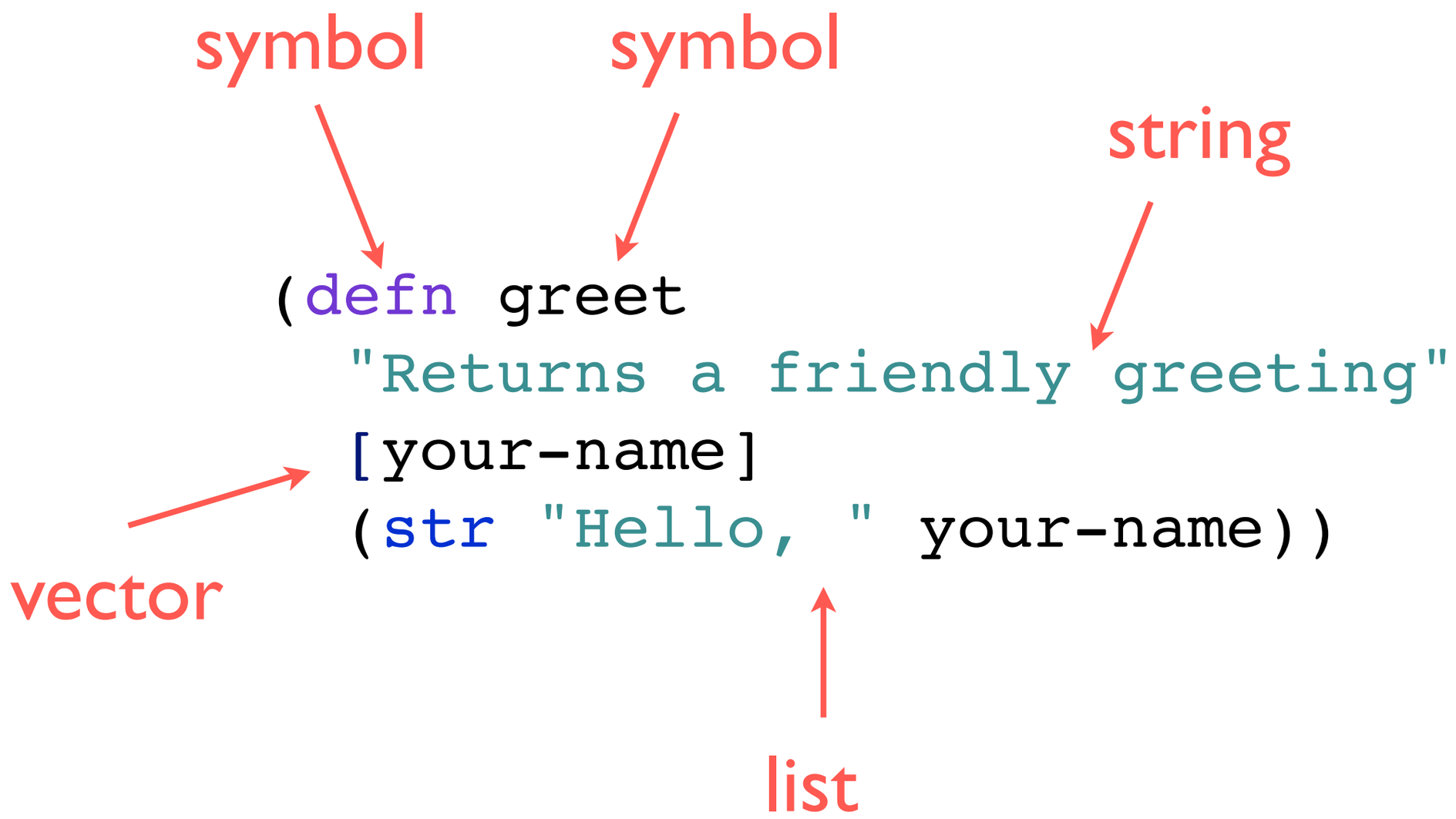
docstring

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

arguments

fn body

...Still Just Data



n separate processes

```
(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"

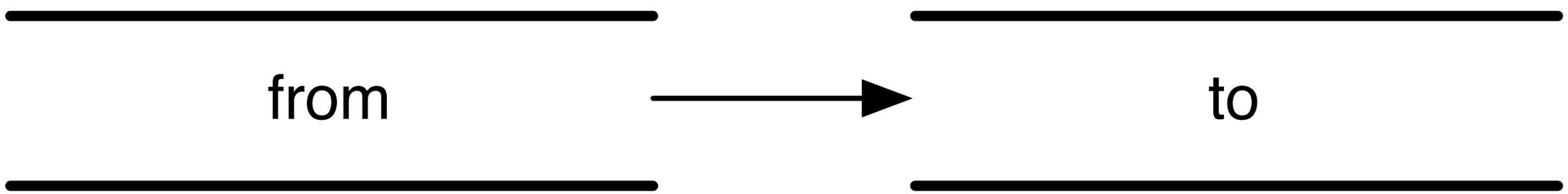
```
(let [out (by-id "ex0-out")]
  (go (loop [results []]
        (set-html out (render results))
        (recur (-> (conj results (<! c)) (peekn 10))))))
```

channel get

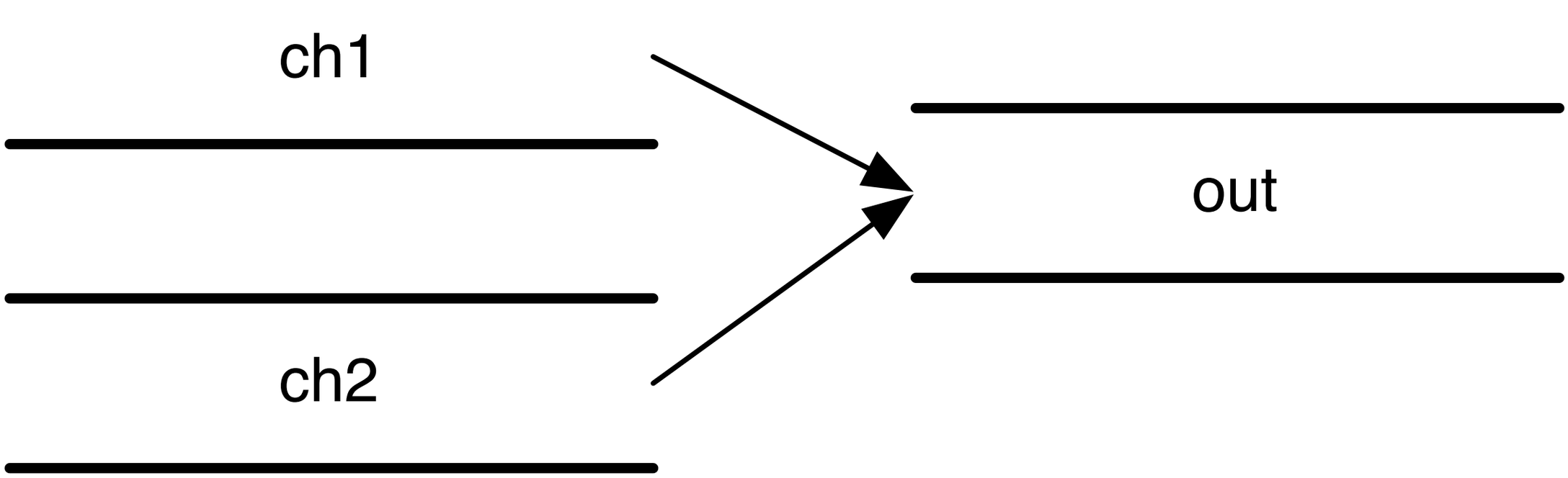
https://github.com/stuarthalloway/exploring-clojure/blob/master/src/cljs/scriptbowl2013/core.cljs

joint fittings

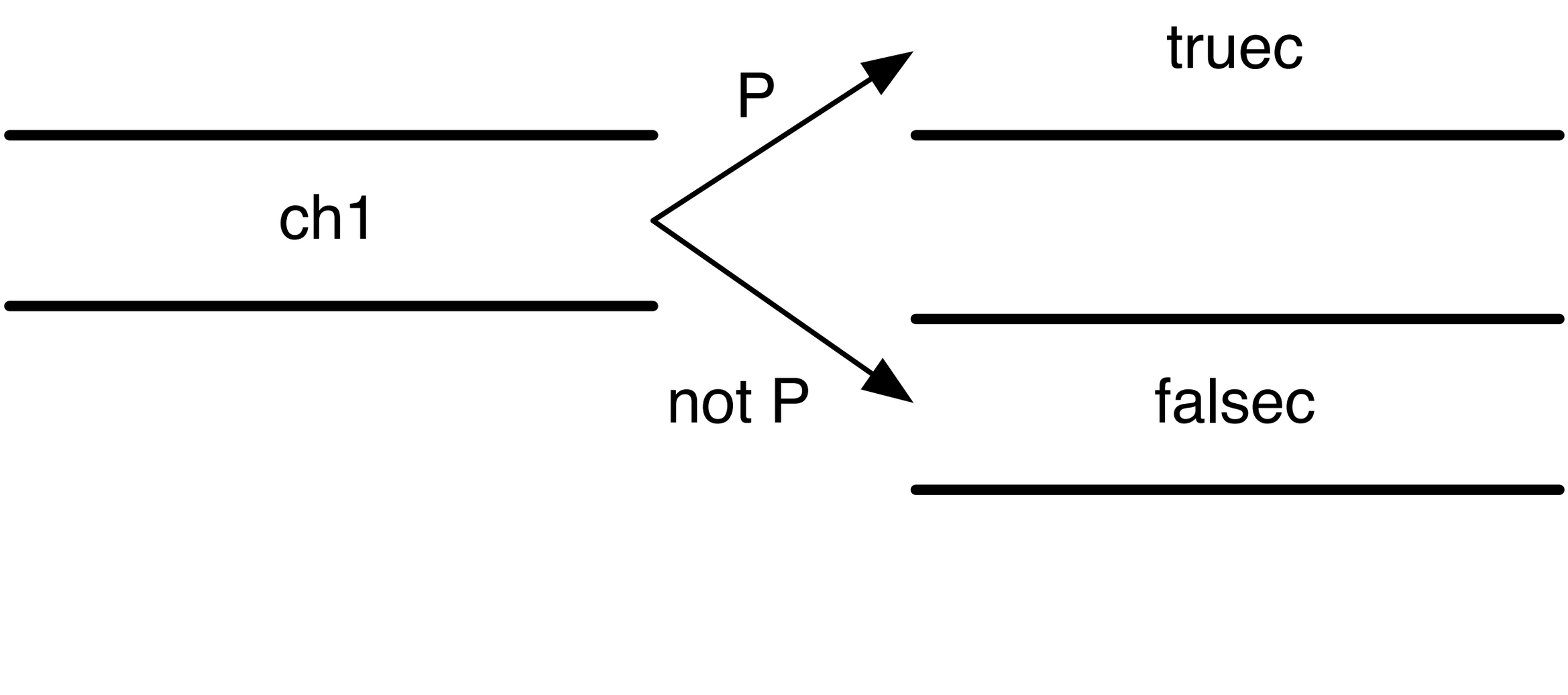
```
(pipe from to)
```



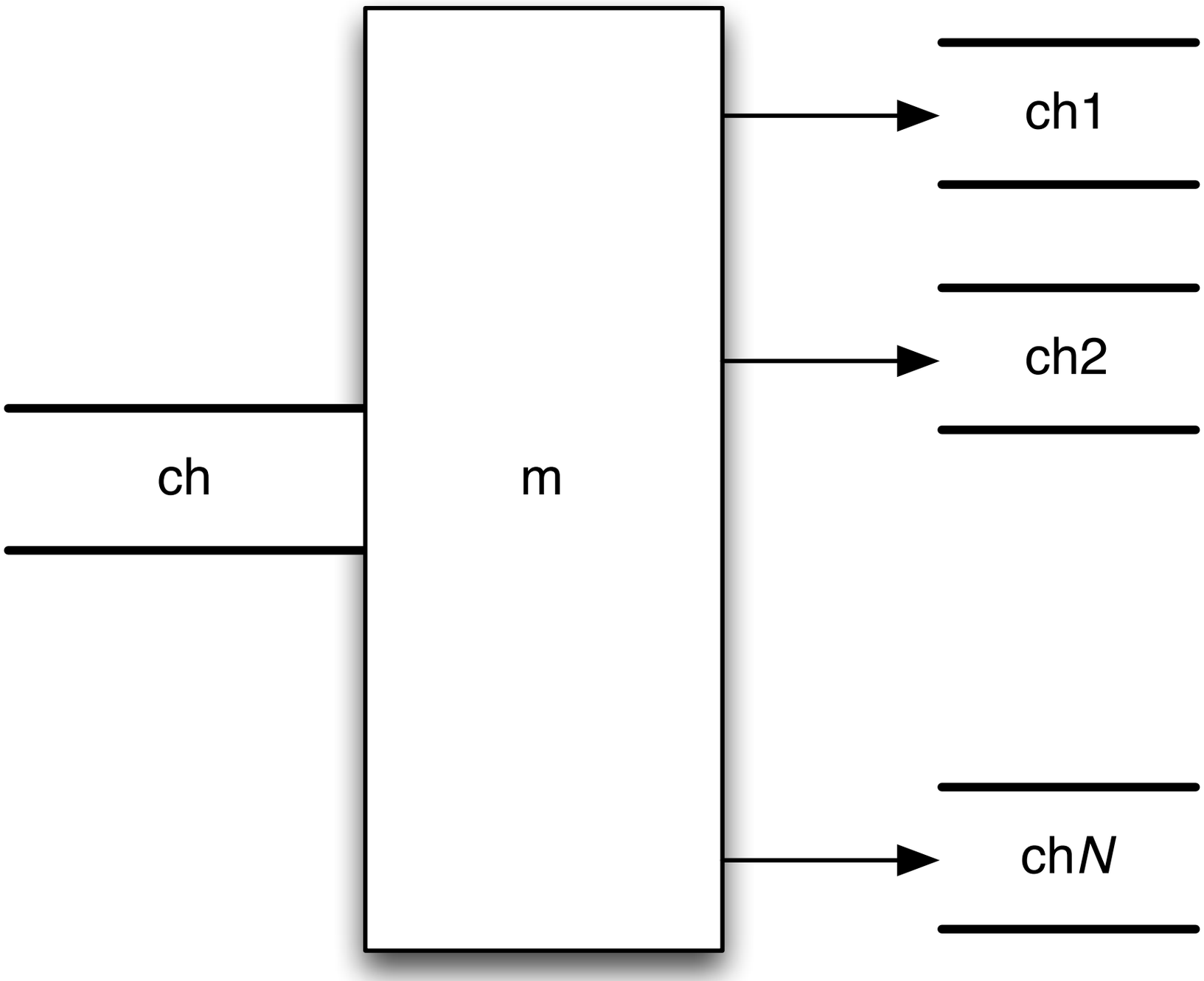
```
(merge ch1 ch2 out)
```



```
(split p ch1 truec falsec)
```



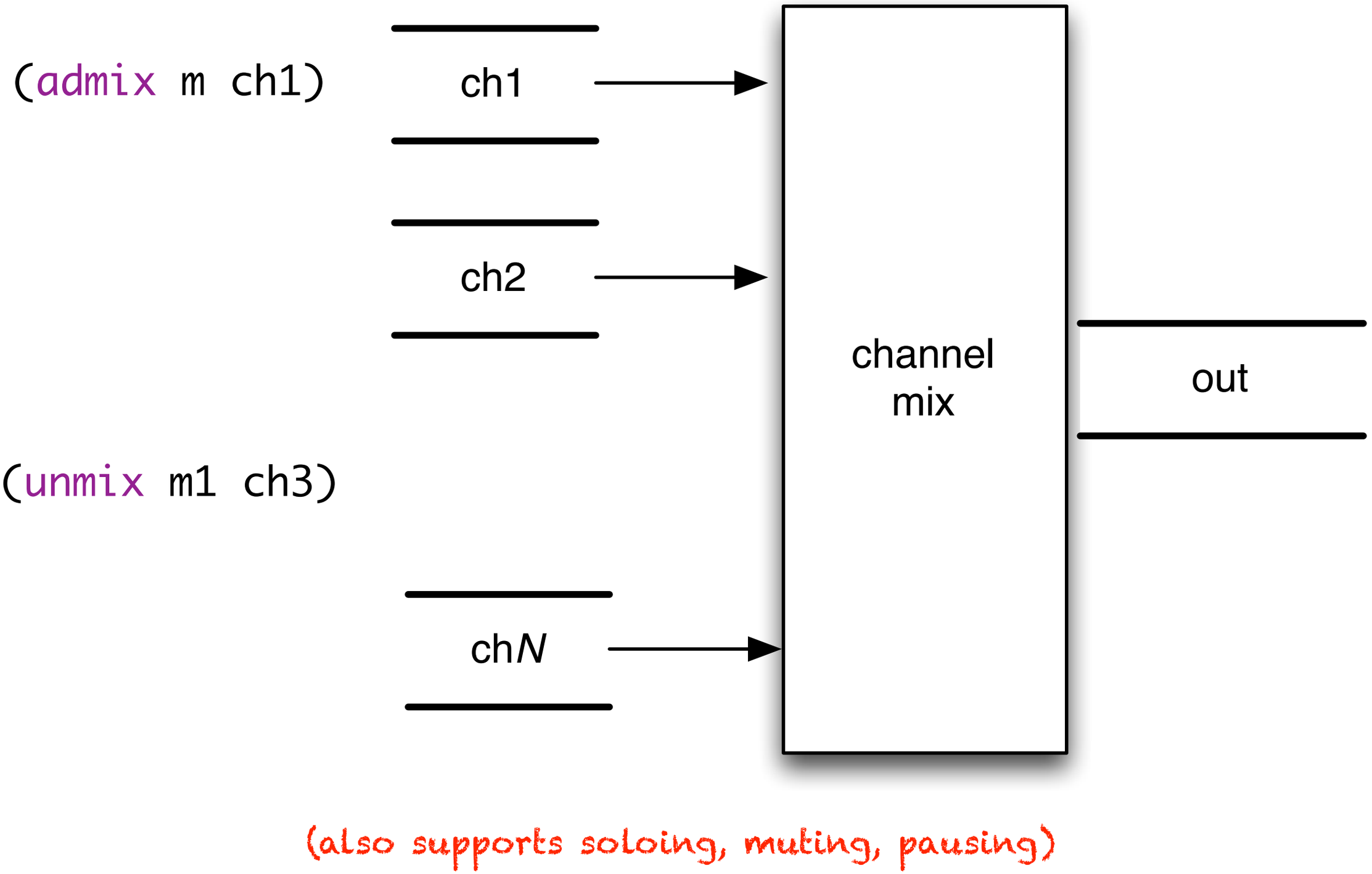
(mult ch)



(tap m1 ch1)

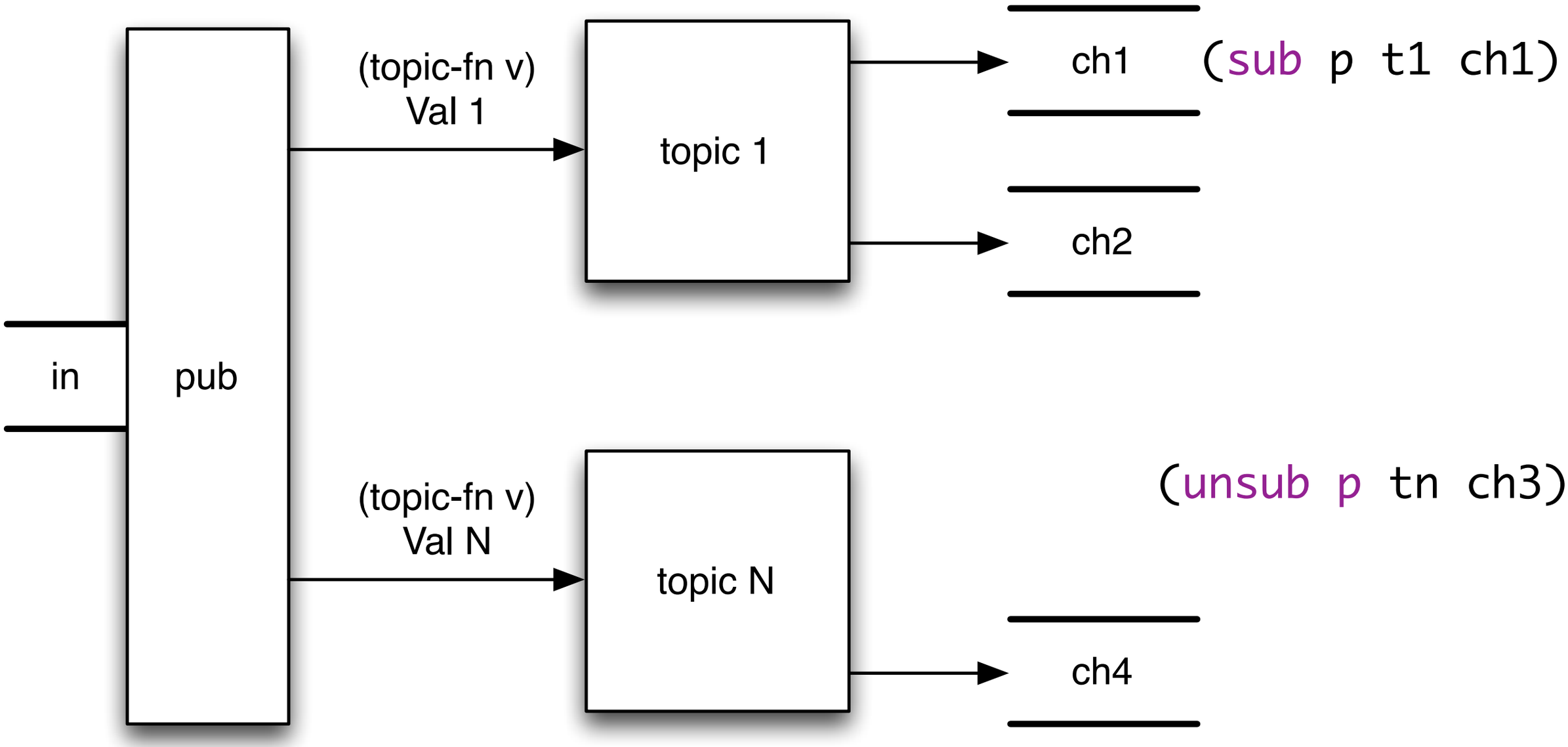
(untap m1 ch3)

(mix ch)



(also supports soloing, muting, pausing)

(pub ch topic-fn)



multi-ops: alt*

one process, N concurrent inputs/outputs

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 "take" op

[value, channel] taken

multi-operation examples

http://go.cognitect.com/core_async_webinar_recording

https://github.com/cognitect/async-webinar examples 6-10

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

differences from go

operations are expressions (not statements)

core.async is a library, not a language feature

alts! is a *function*

alt supports priority

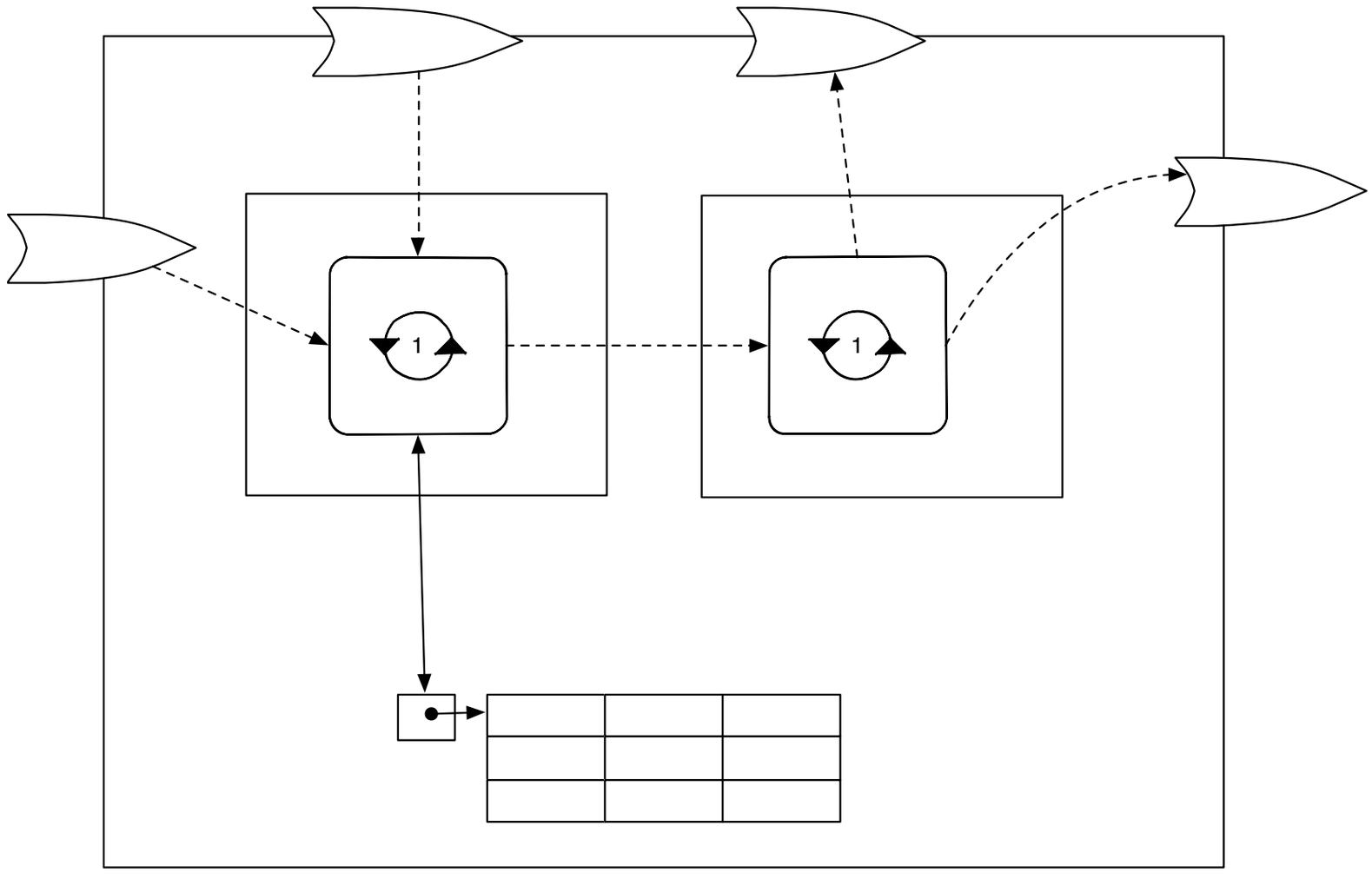
designing subsystems with core.async

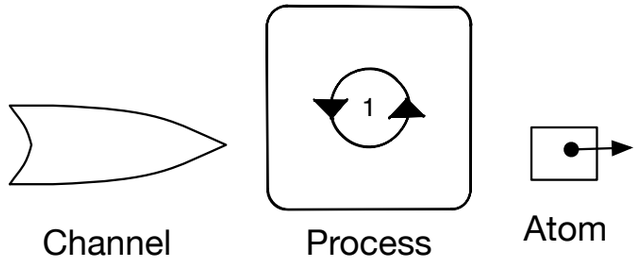
	without async	with async
arguments	active objects	maps
abstraction	protocols	maps
invocation	direct	channel put
queues	occasionally for pipelining	primary subsystem boundary
orchestration	pass objects to constructors	pass channels to constructors
errors	exceptions	don't know, don't care
state	functional *or* unified succession model	functional *or* unified succession model

control coupling with buffers

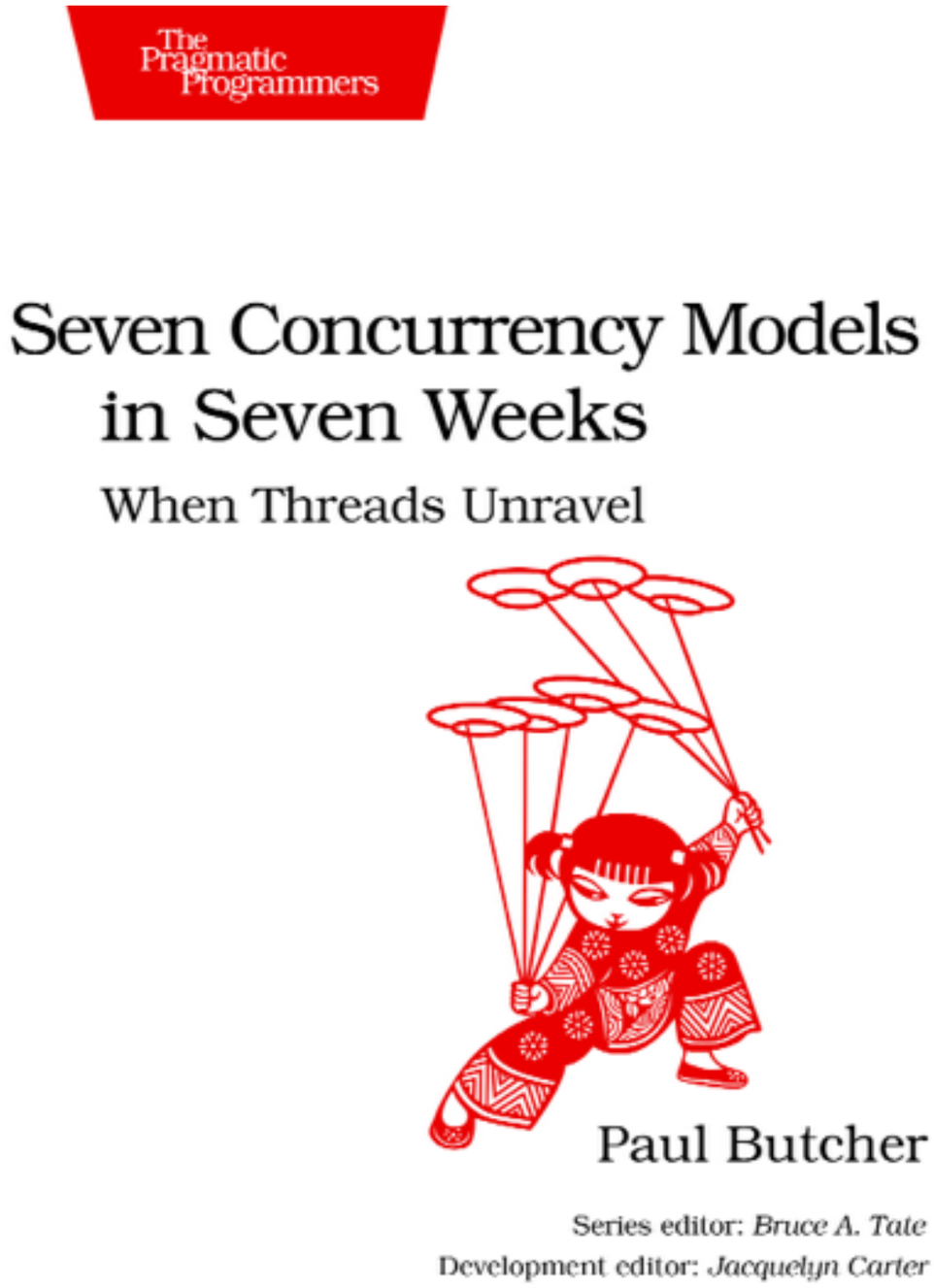
strategy	semantics	example
unbuffered	rendezvous	(chan)
fixed	block when full	(chan 10)
sliding	drop oldest when full	(chan (sliding-buffer 10))
dropping	drop newest when full	(chan (dropping-buffer 10))

visual vocabulary



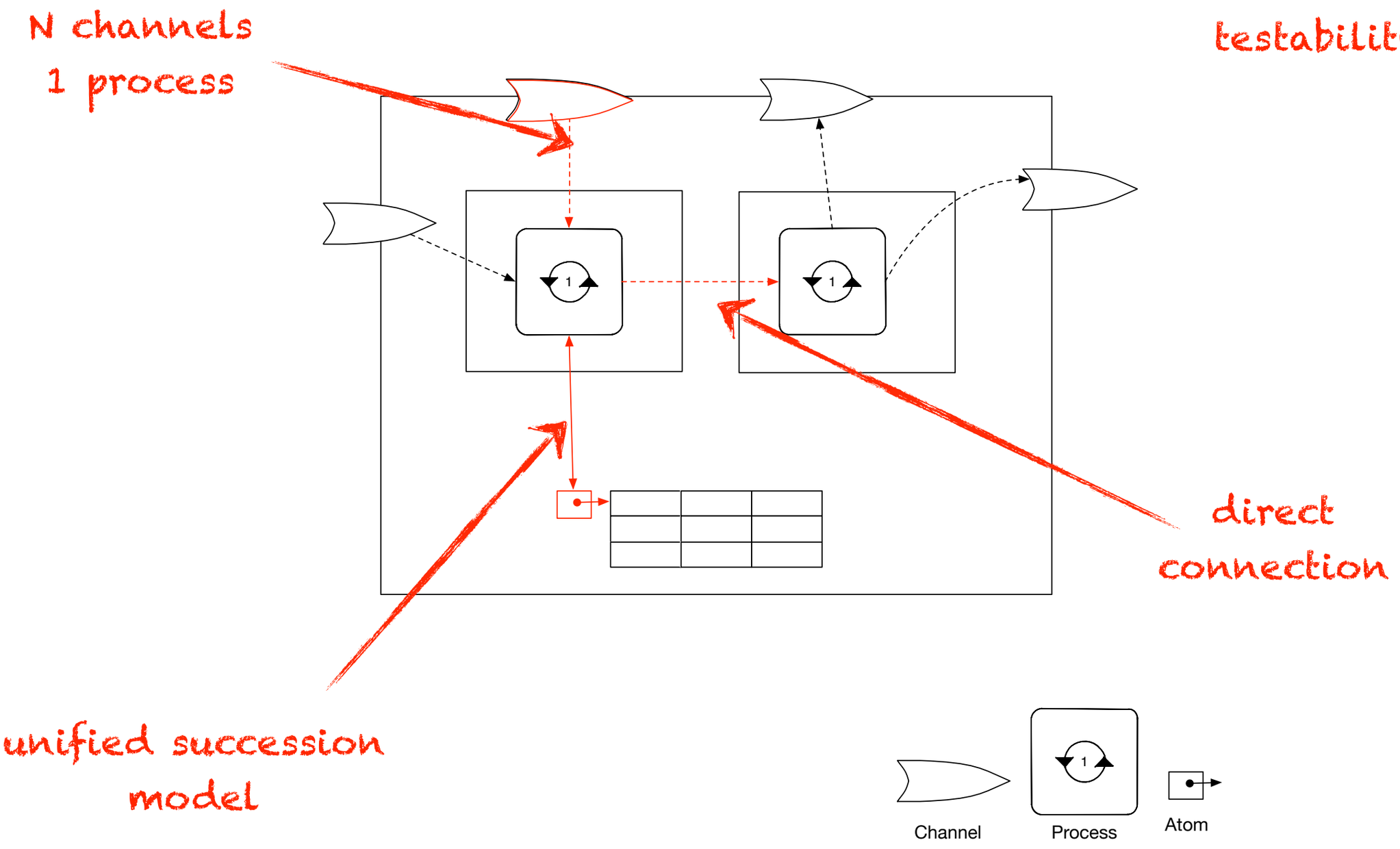


what about actors?



http://pragprog.com/book/pb7con/seven-concurrency-models-in-seven-weeks

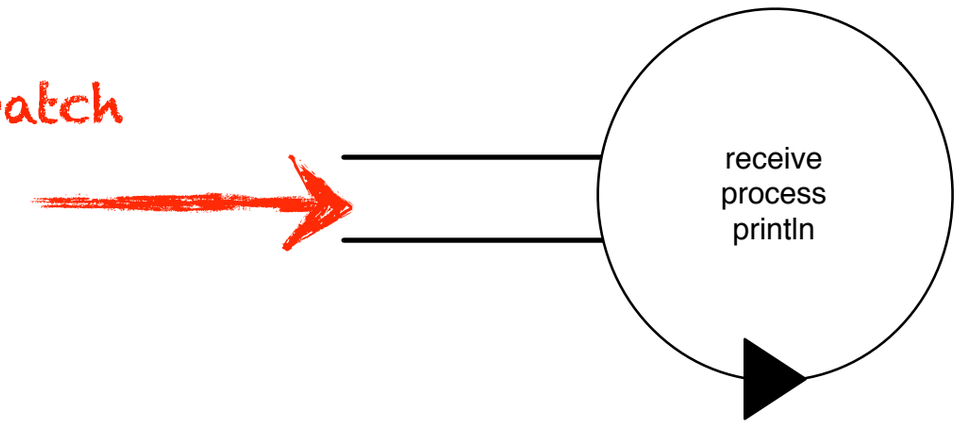
challenges for actors



hello actors

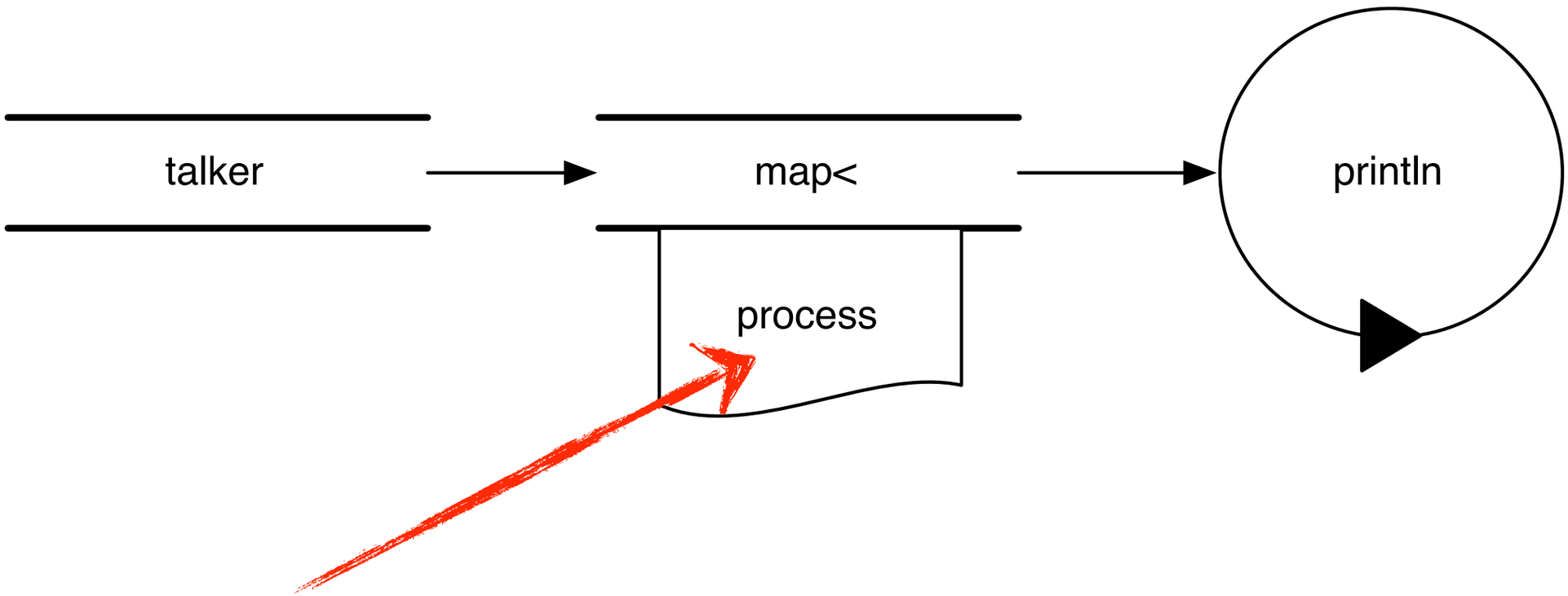
```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
    end
    loop
  end
end
```

channel, process, & dispatch
fused together



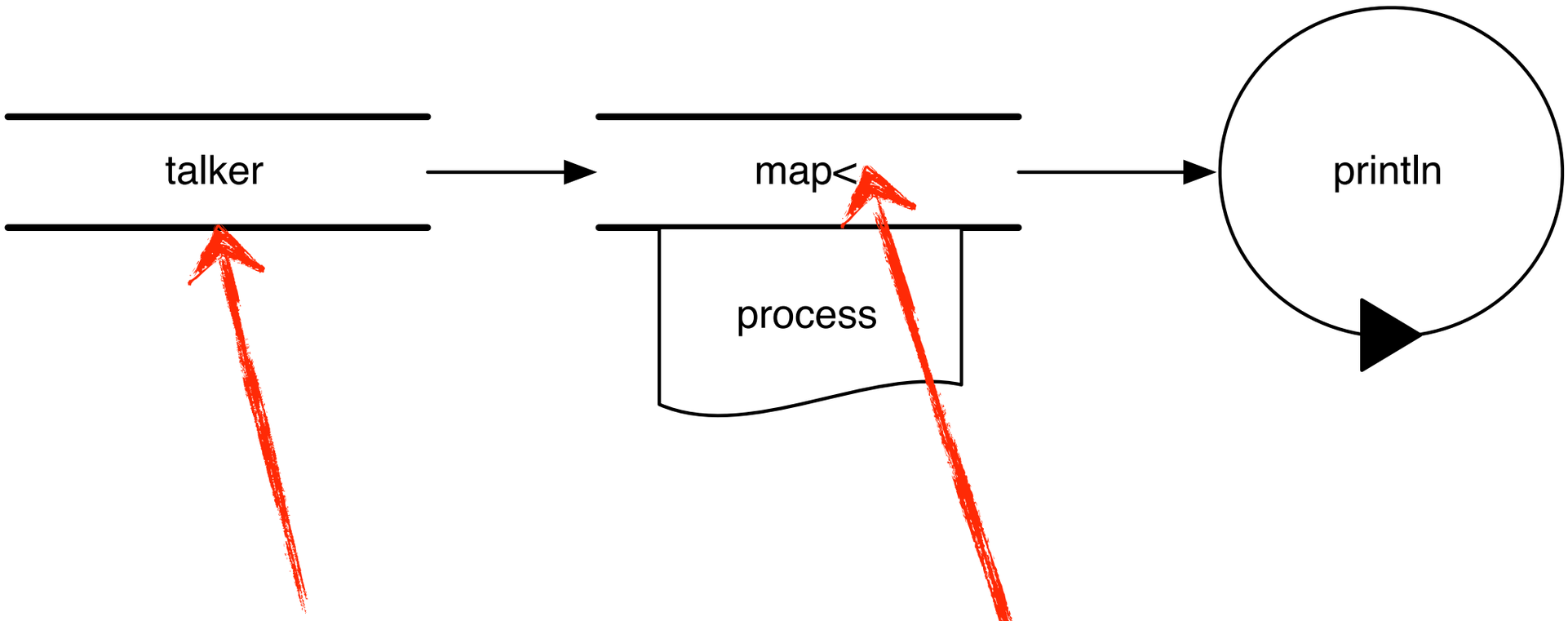

```
pid <- {:greet, "Huey"}
pid <- {:praise, "Dewey"}
pid <- {:celebrate, "Louie", 16}
```

closed (pattern) dispatch



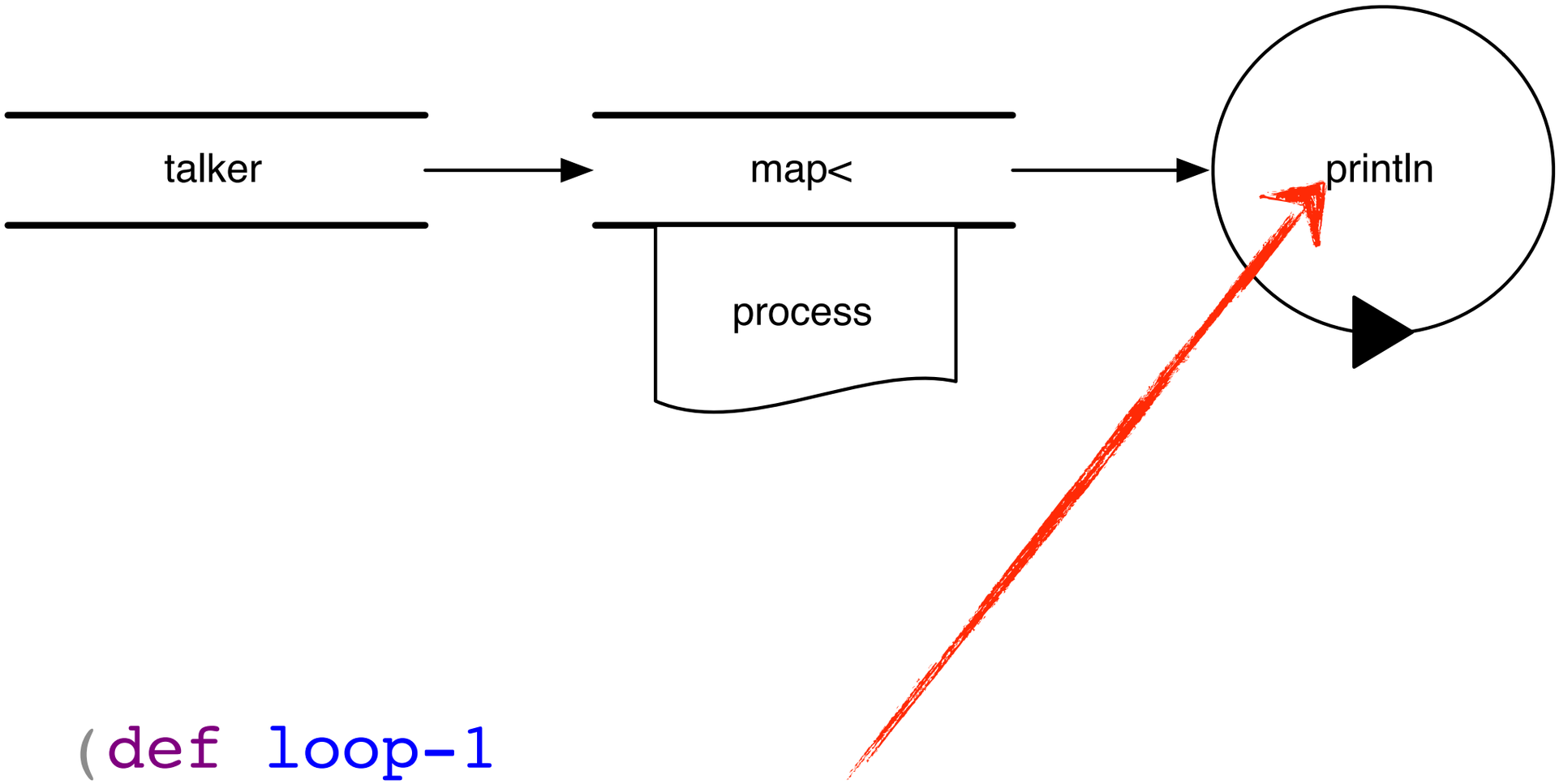

```
(defn process-1
  [item]
  (match
   [item]
   [[:greet & [name]]] (str "Hello " name)
   [[:praise & [name]]] (str name ", you're amazing")
   [[:celebrate & [name age]]] (str "Here's to another " age " years, " name)))
```

+channels



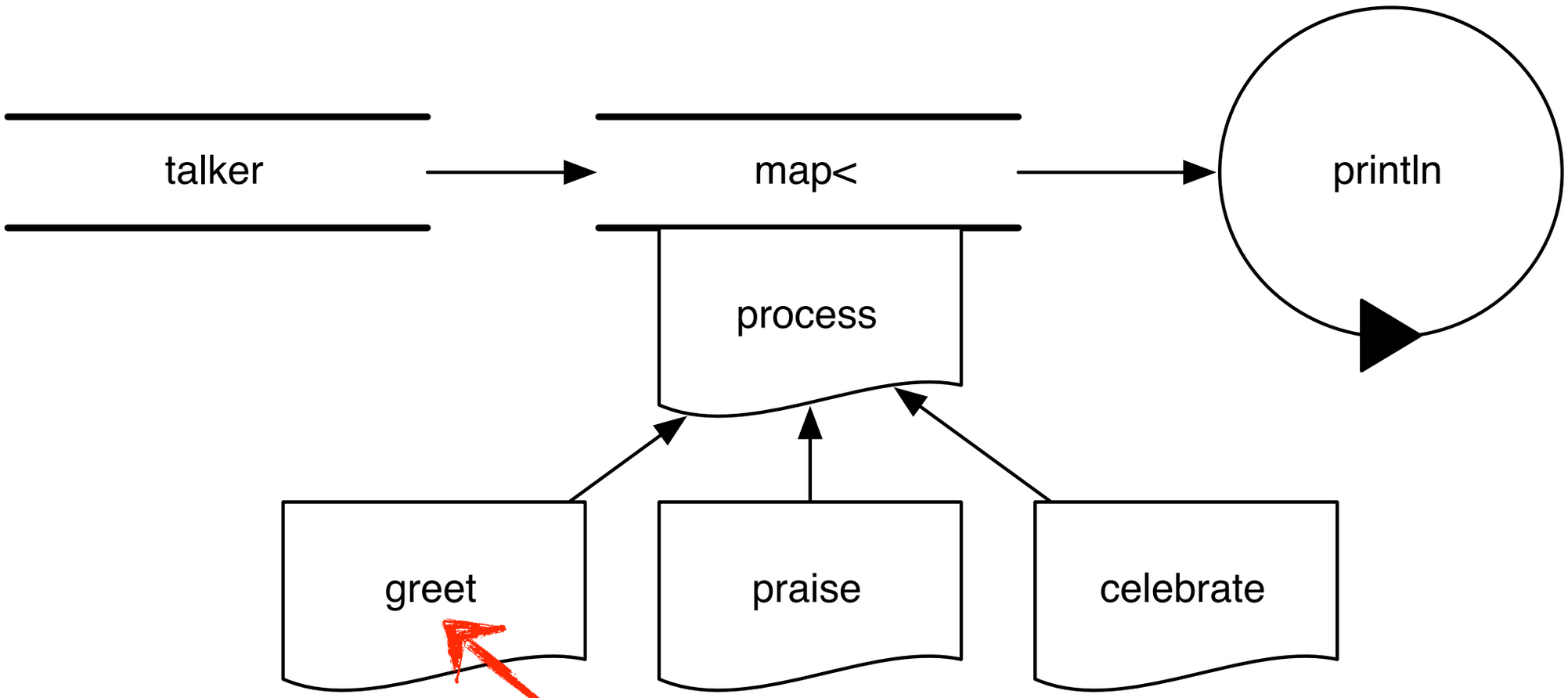

```
(def talker-ch-1 (chan))


   (def composed-ch-1
     (->> talker-ch-1 (map< process-1)))
```

+processes



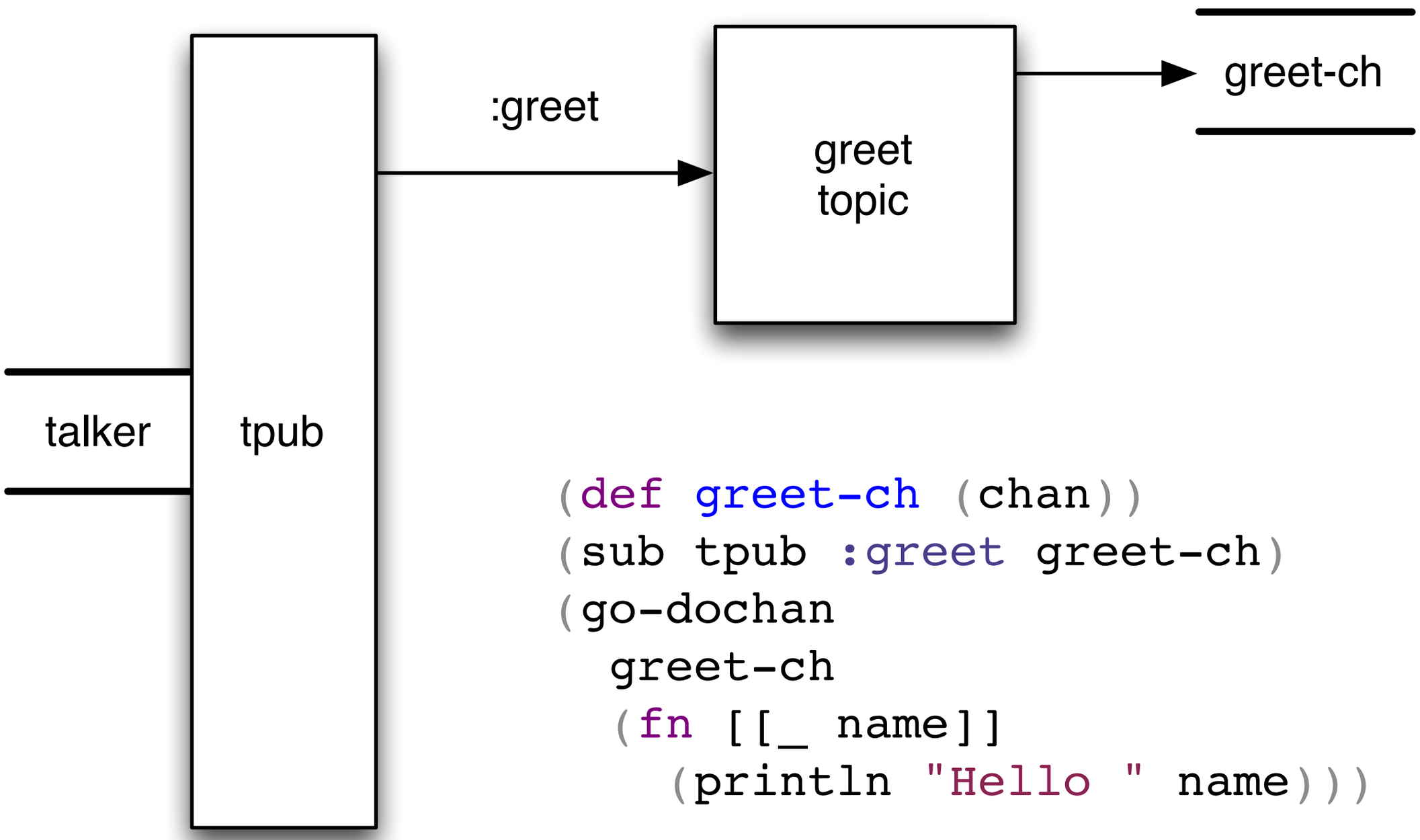

```
(def loop-1
  (go-loop [msg (<! composed-ch-1)]
           (when msg
             (println msg)
             (recur (<! composed-ch-1)))))
```

substitute open dispatch




```
(defmulti process-2 (fn [[k & more]] k))

(defmethod process-2 :greet
  [[_ & [name]]]
  (str "Hello " name))
```

substitute pub/sub




```
(def greet-ch (chan))
(sub tpub :greet greet-ch)
(go-dochan
  greet-ch
  (fn [[_ name]]
    (println "Hello " name)))
```

om: ClojureScript + React

app	live server	code
Netrunner	http://www.jinteki.net/	https://github.com/mtgred/netrunner
Goya (pixel editor)	http://jackschaedler.github.io/goya/	https://github.com/jackschaedler/goya
Wordsmith (markdown editor)	http://wordsmith.variadic.me/	https://github.com/eakron/wordsmith

https://github.com/omcljs/om

resources

core.async and CSP
http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html
https://github.com/clojure/core.async
http://www.cs.kent.ac.uk/projects/ofa/jcsp/
http://www.usingcsp.com/

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

cognitect