Clojure

A (not-so-pure) functional approach to concurrency

Paolo Baldan

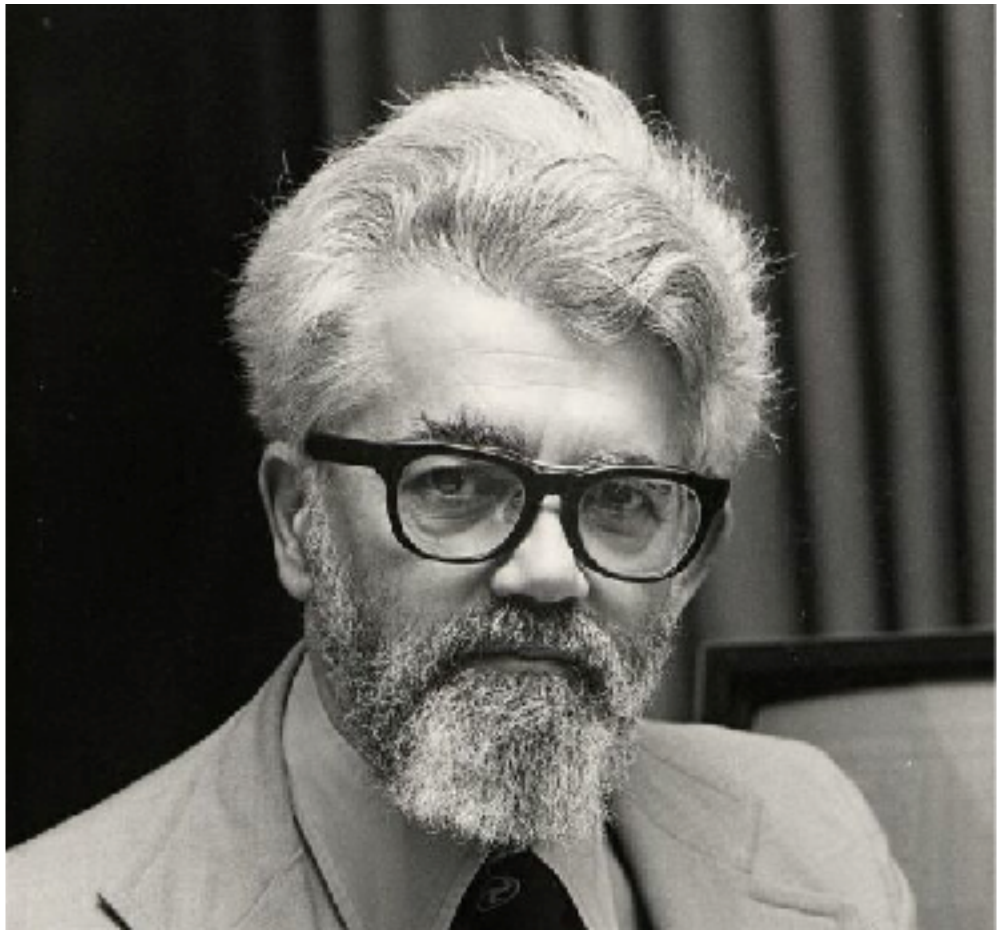
Languages for Concurrency and Distribution

In the words of the inventor

- Functional programming language (rooted in Lisp, from 60s … old but beautifully compact language)
- symbiotic with an established platform (JVM)
- designed for concurrency
- with focus on immutability

Rich Hickey, 2010

Where all started



“uncle”
John McCarthy

Turing award
1971

“Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I” MIT 1960

Pragmatic

- Focus on pure functions (immutable data types), but allows for mutability
- Functional, but allows for control (e.g., iteration)
- Direct compilation into JVM (strings are java strings, …, access to Java's libraries)
- Used in industry (CapitalOne, Amazon, Facebook, Oracle, Boeing)

Outline

- **Basics**: expressions and evaluation
- **Functional parallelism** (on data, for free)
- **Programmable concurrency**: futures, promises and all that (STM?)

Basics

Program as data

- **Code** as a **data structure** (**everything** is a **list**)
- Tools offered for manipulating data as code (eval, quote, apply, etc.)
- Make the language easily **extensible** (see macros)

"A programmable programming language."

Starting …

- Numbers

```
user> 9
9
```

- Strings

```
user> "Here I am"
"Here I am"
```

- Keywords (atoms)

```
user> :name
:name
```

Evaluate to themselves

Lists

- … it's essentially all about that

```
(mylist 1 2)
```

```
nil
```

- In the evaluation the first element is interpreted as a function

```
user> (+ 1 2)
3
```

```
user> (* 2 (+ 1 2 3))
12
```

Defining things: vars

- **Vars**: names for pieces of Clojure data

```
user> (def word "cap")
#'user/word
```

```
user> (def len (count word))
#'user/len
```

- Evaluate to the corresponding value

```
user> len
3
```

- Vars can be changed …

```
user> (def len 2)
#'user/len
```

Vectors

- Int indexed vectors

```
user> (def myvec [12 "some string" :name])
#'user/myvec


user=> (myvec 2)
:name


user=> (myvec 1)
"some string"


user=>(myvec 3)
IndexOutOfBoundsException …
```

Vectors

- Copy, not modify

```
user> (assoc myvec 2 :newname)
[12 "some string" :newname]

user=> myvec
[12 "some string" :name]
```

Maps

- Key/val maps

```
user=> (def mymap {"one" 1, "two" 2, "three" 3})
#'user/mymap

user=> (mymap "one")
1

user=> (mymap "on")
nil
```

- Comma ok style

```
user=> (mymap "on" :error)
:error
```

Functions

- **Unnamed** functions

```clojure
(fn [x y]
    "Take the x percentage of y"
    (* x (/ y 100.0)))      ; evaluate in sequence
                            ; value is last
```

- **Naming** functions with **def**

```clojure
(def percentage
    "Take the x percentage of y"
    (fn [x y]
        (* x (/ y 100.0)))
)
```

Functions

- Defining functions with **defn**

```
(defn percentage
    "Take the x percentage of y"
    [x y]
    (* x (/ y 100.0))
)
```

- Different **arities**

```
(defn percentage
    "Take the x percentage of y (50% by default)"
    ([x y] (* x  (/  y 100.0)))
    ([y]   (* 50 (/ y 100.0)))
)
```

Control & Recursion

- **If**

```
(defn sum-down-from [x]
  (if (pos? x)
    (+ x (sum-down-from (dec x)))
    0 )
)
```

- **Case-like**

```
(defn fib [n]
  (cond
    (= n 0) 1
    (= n 1) 1
    :else (+ (fib (- n 1))
             (fib (- n 2)))))
```

Careful with recursion

- Very natural, but can have horrible performances (try with 9 or 100)

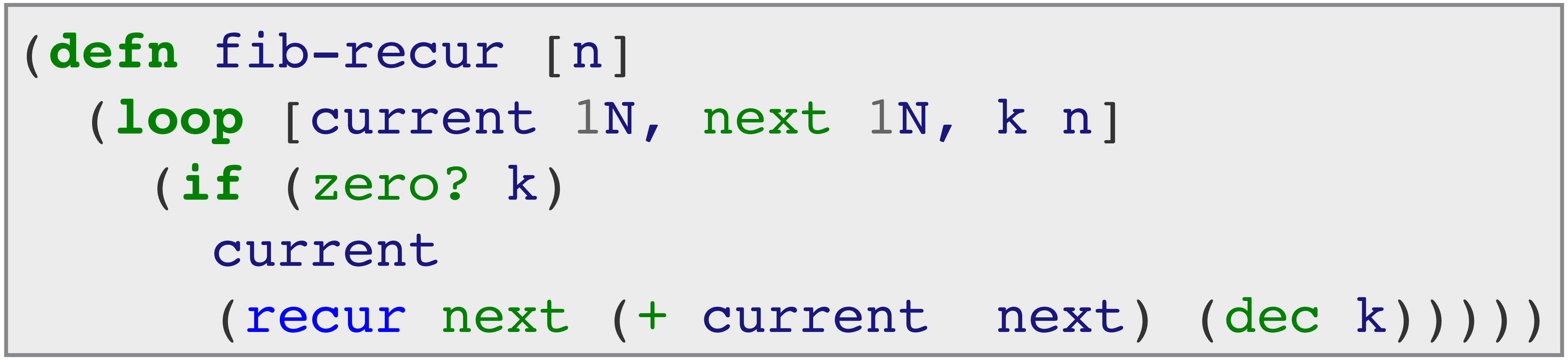
- Tail recursive

```
(defn fib-tail [n]
   (letfn [(fib [current next k]
            ; idea: fib [fib(n-k) fib(n+1-k) k]
            (if (zero? k)
               current
               (fib next (+ current  next) (dec k))))]
    (fib 1N 1N n)))
```

- Better, but no tail call optimisation (try with 100000)

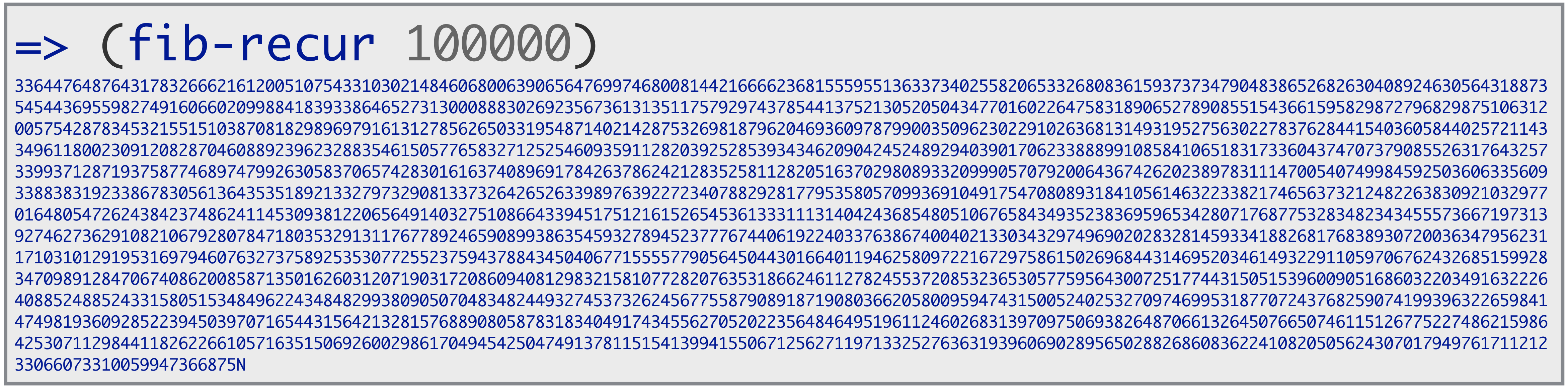
Tail elimination,
do it yourself with recur

- Tail recursion, explicitly with loop-recur



```
(defn fib-recur [n]
  (loop [current 1N, next 1N, k n]
    (if (zero? k)
      current
      (recur next (+ current  next) (dec k)))))
```

- Works better!



```
=> (fib-recur 100000)
3364476487643178326662161200510754331030214846068006390656476997468008144216666236815559551363373402558206533268083615937373479048386526826304089246305643188 73
5454436955982749160660209988418393386465273130008883026923567361313511757929743785441375213052050434770160226475831890652789085515436615958298727968298751 06312
0057542878345321551510387081829896979161312785626503319548714021428753269818796204693609787990035096230229102636813149319527563022783762844154036058440257211 43
3496118002309120828704608892396232883546150577658327125254609359112820392528539343462090424524892940390170623388899108584106518317336043747073790855263176432 57
3399371287193758774689747992630583706574283016163740896917842637862421283525811282051637029808933209990570792006436742620238978311147005407499845925036063356 09
3388383192338678305613643535189213327973290813373264265263398976392272340788292817795358057099369104917547080893184105614632233821746563732124822638309210329 77
0164805472624384237486241145309381220656491403275108664339451751216152654536133311131404243685480510676584349352383695965342807176877532834823434555736671973 13
9274627362910821067928078471803532913117677892465908993863545932789452377767440619224033763867400402133034329749690202832814593341882681768389307200363479562 31
1710310129195316979460763273758925353077255237594378843450406771555577905645044301664011946258097221672975861502696844314695203461493229110597067624326851599 28
3470989128470674086200858713501626031207190317208609408129832158107728207635318662461127824553720853236530577595643007251774431505153960090516860322034916322 26
4088524885243315805153484962243484829938090507048348244932745373262456775587908918719080366205800959474315005240253270974699531877072437682590741993963226598 41
4749819360928522394503970716544315642132815768890805878318340491743455627052022356484649519611246026831397097506938264870661326450766507461151267752274862159 86
4253071129844118262266105716351506926002986170494542504749137811515413994155067125627119713325276363193960690289565028826860836224108205056243070179497617112 12
33066073310059947366875N
```

Quoting and unquoting

- **Quote**: prevent evaluation

```
=> (+ 1 2)
3
=> '(+ 1 2)              ; also (quote (+ 1 2))
(+ 1 2)
=> (first '(+ 1 2))
+
```

- **Eval**: force evaluation

```
=> (def mysum (quote (+ 1 1)) )
=> (first  mysum)
+
=> (eval  mysum)
2
```

Interfacing with Java

- Accessing and using Java classes

```
user> (new java.util.HashMap {"answer" 42,
                              "question" "who knows"})
```

```
user> (java.util.HashMap. {"answer" 42,
                           "question" "who knows"})
```

```
user> (def myhash
          (java.util.HashMap. {"answer" 42,
                               "question" "who knows"}))
#'user/myhash
user> (.get myhash "answer")
42
```

Functional Parallelism (and laziness)

Functional parallelism

- Program are pure functions, copying not modifying
  - No mutable state:
  - No side effects
- Parallelization for map, reduce and all that

- Some form of laziness
  - Evaluate (realize) it when (if) you need it
  - Clojure is not lazy, in general, but sequences are

Summing numbers

- Sum of a sequence of numbers, recursively

```
(defn recursive-sum [numbers]
  (if (empty? numbers)
    0
    (+ (first numbers) (recursive-sum (rest numbers)))))
```

- Get rid of the tail

```
(defn recur-sum [numbers]
  (loop [ acc 0, list numbers ]
    (if (empty? list)
      acc
      (recur (+ acc (first list)) (rest list)))))
```

Using reduce

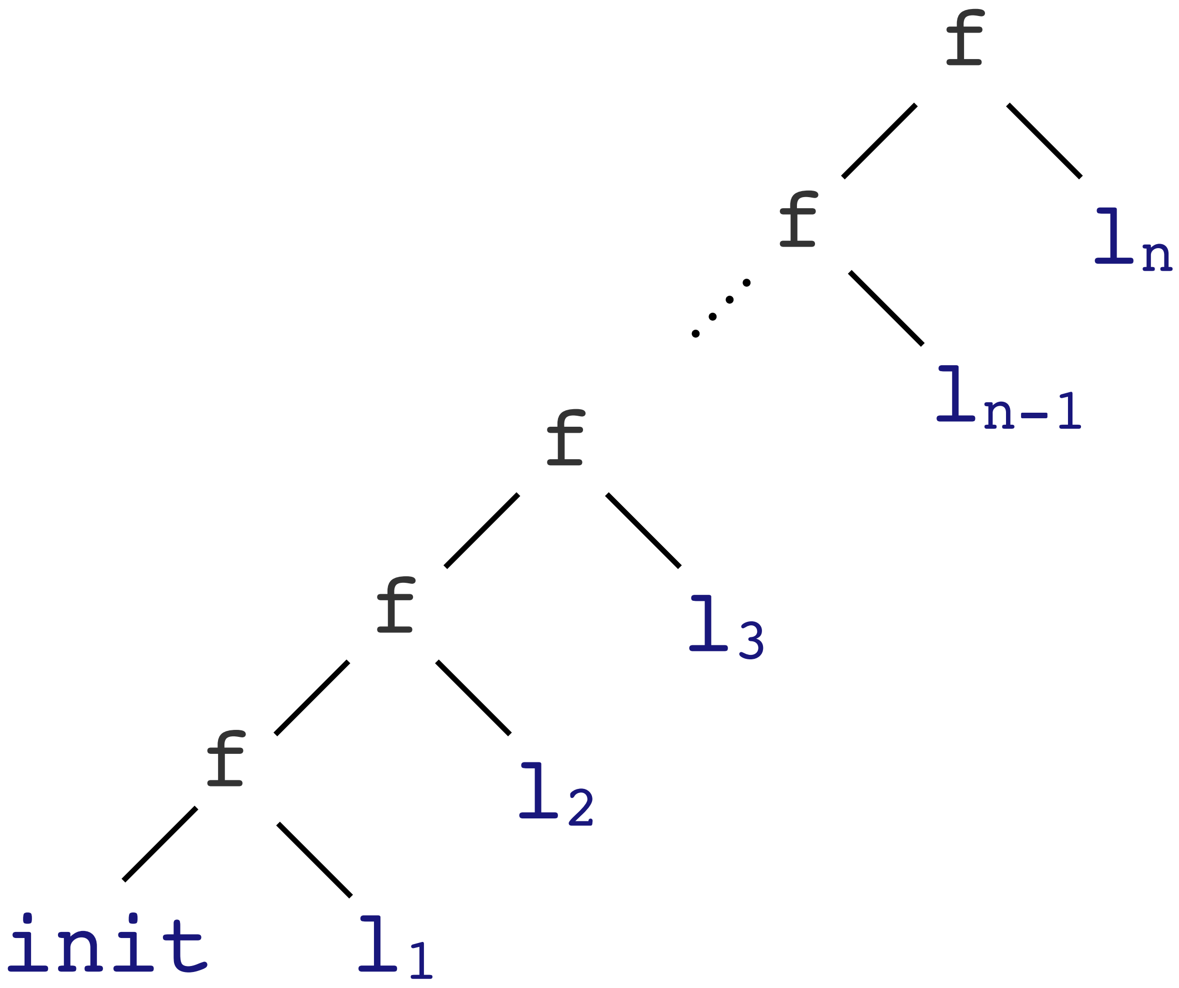
- Much simpler

```
(defn reduce-sum [numbers]
  (reduce (fn [acc x] (+ acc x))  0  numbers))
```

- **Reduce**: applies the function once for each item of the collection
  - Initial result 0
  - Then apply the function on (acc, 1st element)
  - then on (acc, 2nd),
  - then on (acc, 3rd) etc.

Reduce

(**reduce** f init [l_1 ... l_n])



Reduce, reprise

- Even simpler: since + is a function summing two (or more) elements

```
(defn sum [numbers]
  (reduce + 0 numbers))
```

- reduce takes as standard initializer the zero of the type ...

```
(defn sum [numbers]
  (reduce + numbers))
```

Map

- Apply f to all elements of a sequence

```
(map f [l_1 ... l_n])
```

- **Example**: module of a vector $\vec{v} = [x_1, \ldots, x_n]$ -> $\sqrt{x_1^2 + \ldots + x_n^2}$

```
(defn module v
  (Math/sqrt (reduce + (map square v)))
```

```
(defn square [x]
  (* x x))
```

- Looks fine, but still sequential. Can’t we parallelize?

Computing frequencies

- Compute the number of occurrences of each word in a (large) text
- **Idea**: use a map

```
"the cat is on the table" --→
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

```
=>(def counts {"the" 2, "cat" 1})
#user/counts
=>(get counts "the" 0)
2
=>(get counts "tho" 0)
0
=>(assoc counts "is" 1)
{"the" 2, "cat" 1, "is" 1 } ; returns new map
```

Maps recap

Word frequencies

- Word frequencies, sequentially, with reduce

```clojure
(defn counting [counts word]
  (assoc counts
         word (inc (get counts word 0))))
```

```clojure
(defn word-frequencies [words]
  (reduce counting {} words))
```

```
=> (word-frequencies ["the" "cat" "is" "on" "the" "table"])
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

- Compute frequencies, from the string

```
=> (word-frequencies (get-words "the cat is on the table"))
{"the" 2, "cat" 1, "is" 1, "on" 1, "table" 1}
```

Counting words, from several sources

- Imagine we want to count words from several strings
- **Idea**: List of lists of words of all the strings using **`map`**

```
=> (map get-words ["the cat is on the table"
                   "the dog is not on the table"])

(("the" "cat" "is" "on" "the" "table")
 ("the" "dog" "is" "not" "on" "the" "table"))
```

- and same for the frequencies

```
=> (map word-frequencies
        (map get-words ["the cat is on the table"
                        "the dog is not on the table"]))
```

Merging

- Then we need to merge the resulting maps
  - union of the keys
  - sum counts (for common keys)

```
(merge-with f map1 ... mapn)
```

- Proceeds left-to-right, using **f** for combining the values with common keys

```
(def merge-counts (partial merge-with +))

=>(merge-counts {:x 1 :y 2} {:y 1 :z 1})
{:z 1, :y 3, :x 1}
```

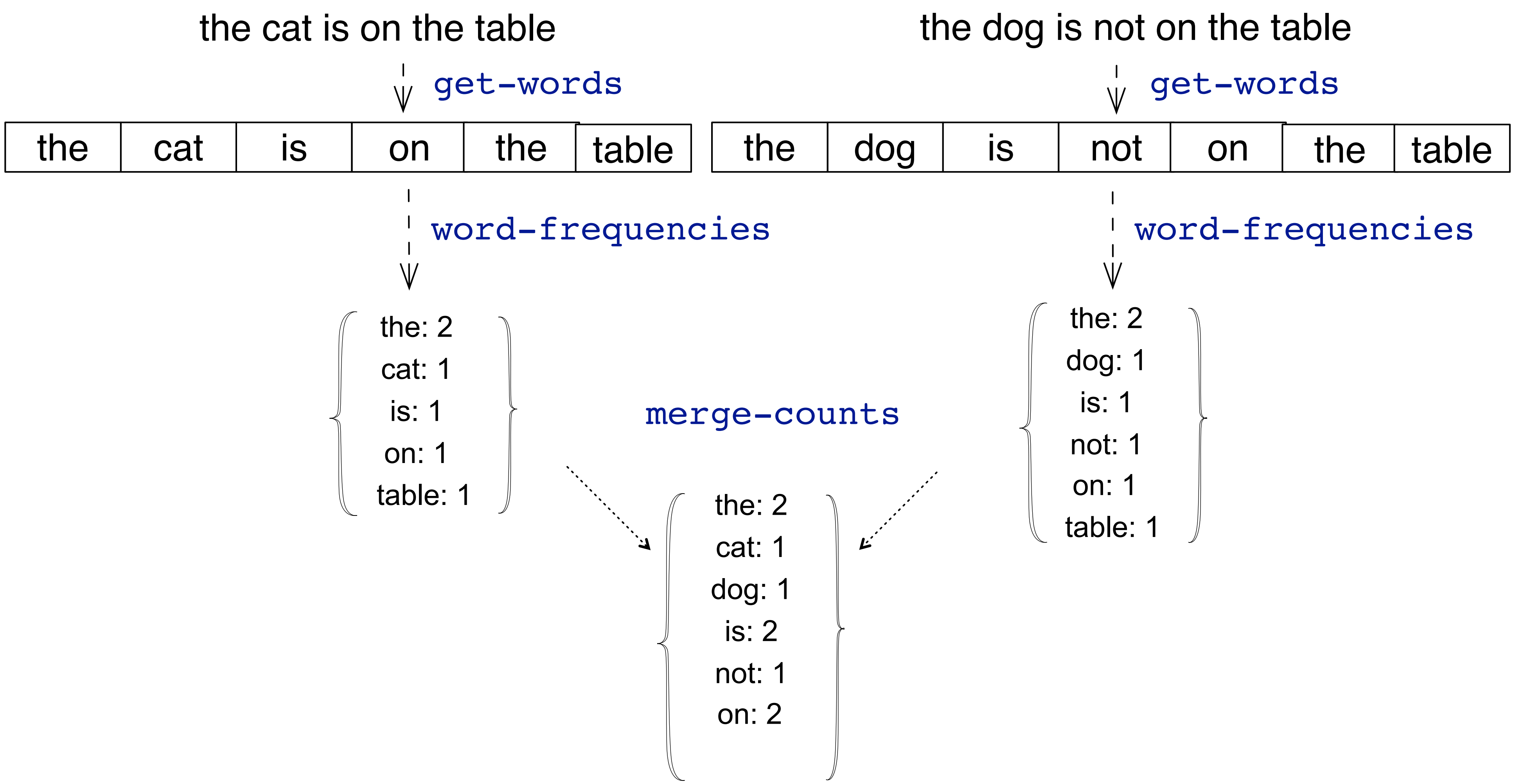
Counting words, from several sources

- Putting things together

```
(defn count-words-sequential [strings]
  (merge-counts
     (map word-frequencies
        (map get-words strings))))
```

Parallelizing!

- Idea:



the cat is on the table

`get-words`

the	cat	is	on	the	table

`word-frequencies`

the: 2
cat: 1
is: 1
on: 1
table: 1

the dog is not on the table

`get-words`

the	dog	is	not	on	the	table

`word-frequencies`

the: 2
dog: 1
is: 1
not: 1
on: 1
table: 1

`merge-counts`

the: 2
cat: 1
dog: 1
is: 2
not: 1
on: 2

- Process different strings and merge in parallel

Parallel Map

- Apply a given function to all elements of a sequence, in parallel

```
=> (pmap inc [1 2 3])
[2 3 4]
```

- Example: Slow inc

```
(defn slow-inc [x] (Thread/sleep 1000) (inc x))
```

```
(map slow-inc (range 10))

(pmap slow-inc (range 10))
```

- What’s happening?

Parallelising

- Use pmap to perform map in parallel

```
(defn count-words-parallel [strings]
  (reduce merge-counts
    (pmap word-frequencies
          (pmap get-words strings))))
```

Parallelising

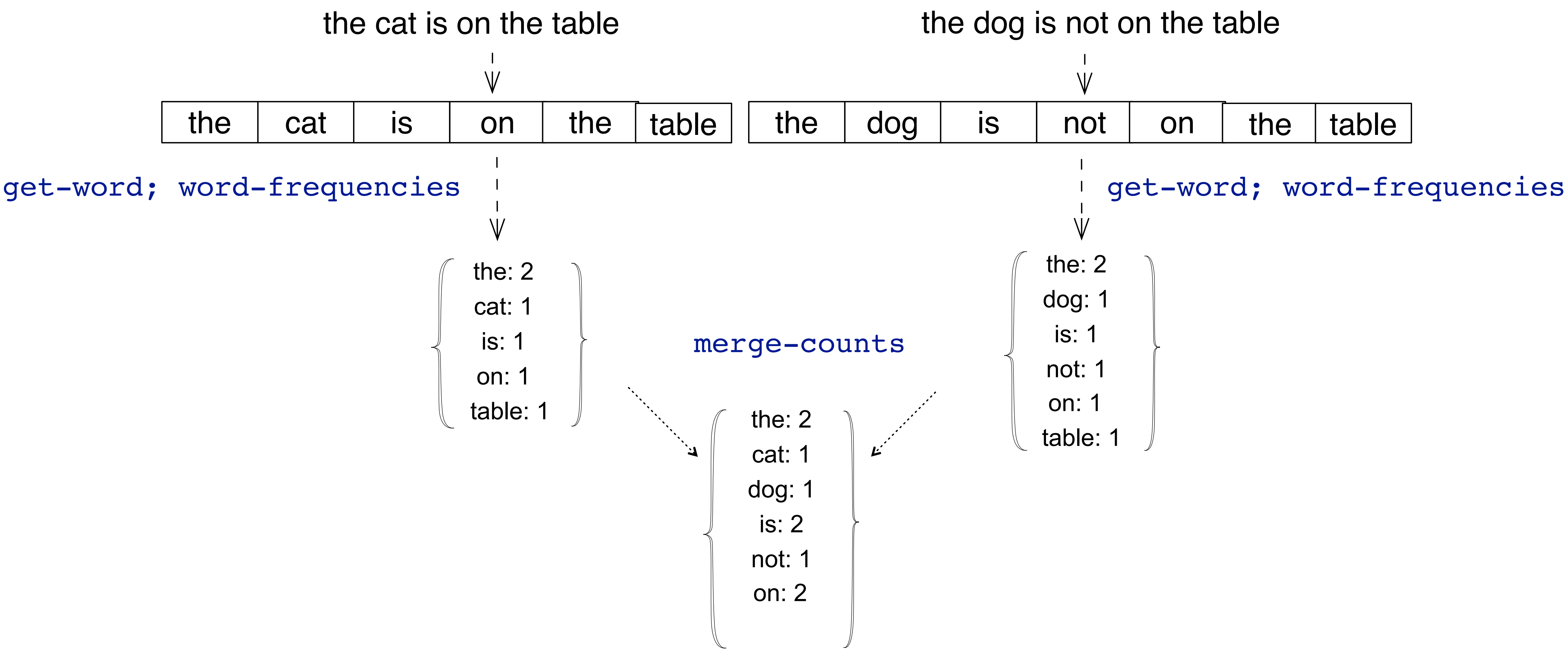
- Avoid going through the sequences twice

```
(defn count-words-parallel [strings]
  (reduce merge-counts
    (pmap #(word-frequencies (get-words %)) strings)))
```

- Macro #(...): Creates a function taking `%1, …,%n` as parameters (`%` stands for `%1`, if none constant fun)

Parallelizing!

- Idea:



the cat is on the table

the | cat | is | on | the | table

`get-word; word-frequencies`

the: 2
cat: 1
is: 1
on: 1
table: 1

the dog is not on the table

the | dog | is | not | on | the | table

`get-word; word-frequencies`

the: 2
dog: 1
is: 1
not: 1
on: 1
table: 1

`merge-counts`

the: 2
cat: 1
dog: 1
is: 2
not: 1
on: 2

- One process per string … too much?

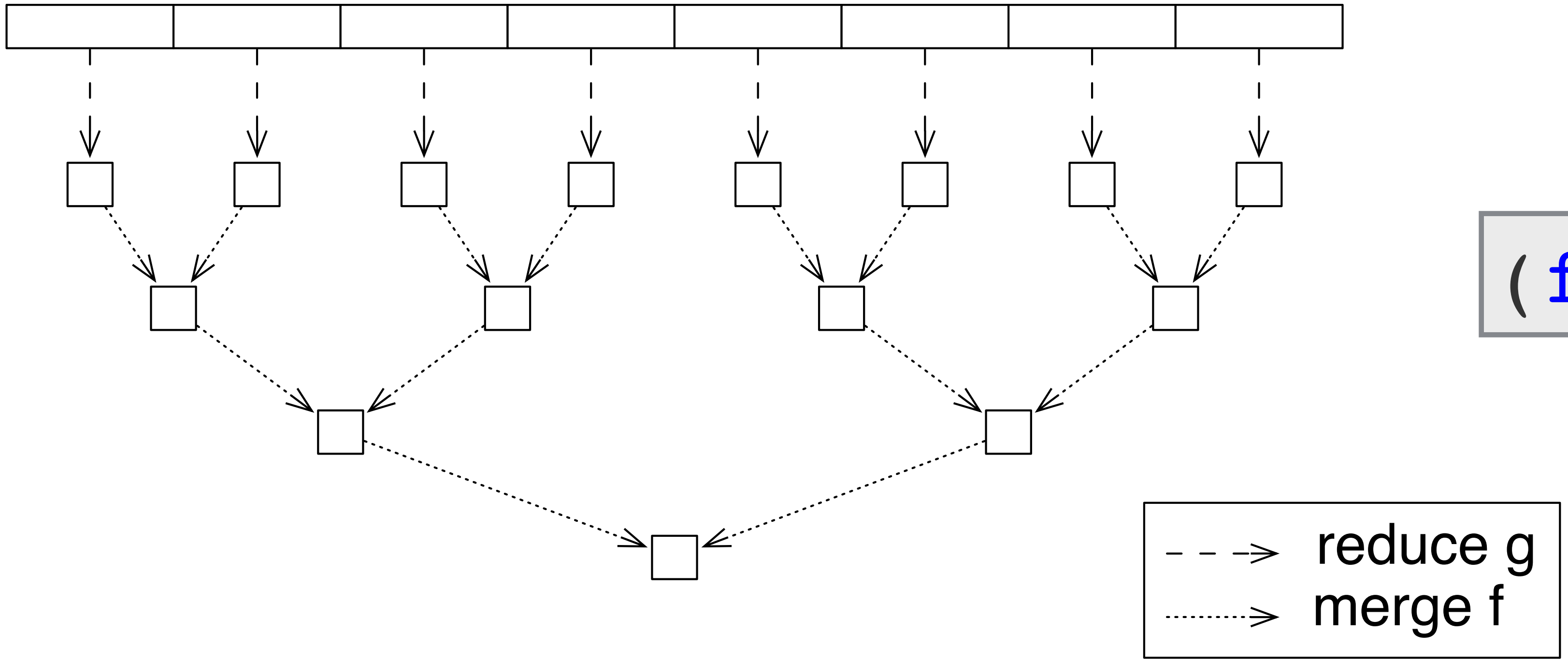
Batching

- The parallel version creates lot of processes for possibly too small jobs
- **Idea**: Create larger batches (size n)

```
(defn count-words-parallel-batch [strings n]
  (reduce merge-counts
          (pmap count-words-sequential
               (partition-all n strings))))
```

Using fold

- Fold works similarly:
    - Split in subproblems (divide & conquer)
    - Different functions for base and merge



`(fold f g coll)`

Fold

- The two functions can coincide …

```
(defn parallel-sum [numbers]
  (fold + numbers))
```

- Subproblems parallelized, more efficient!

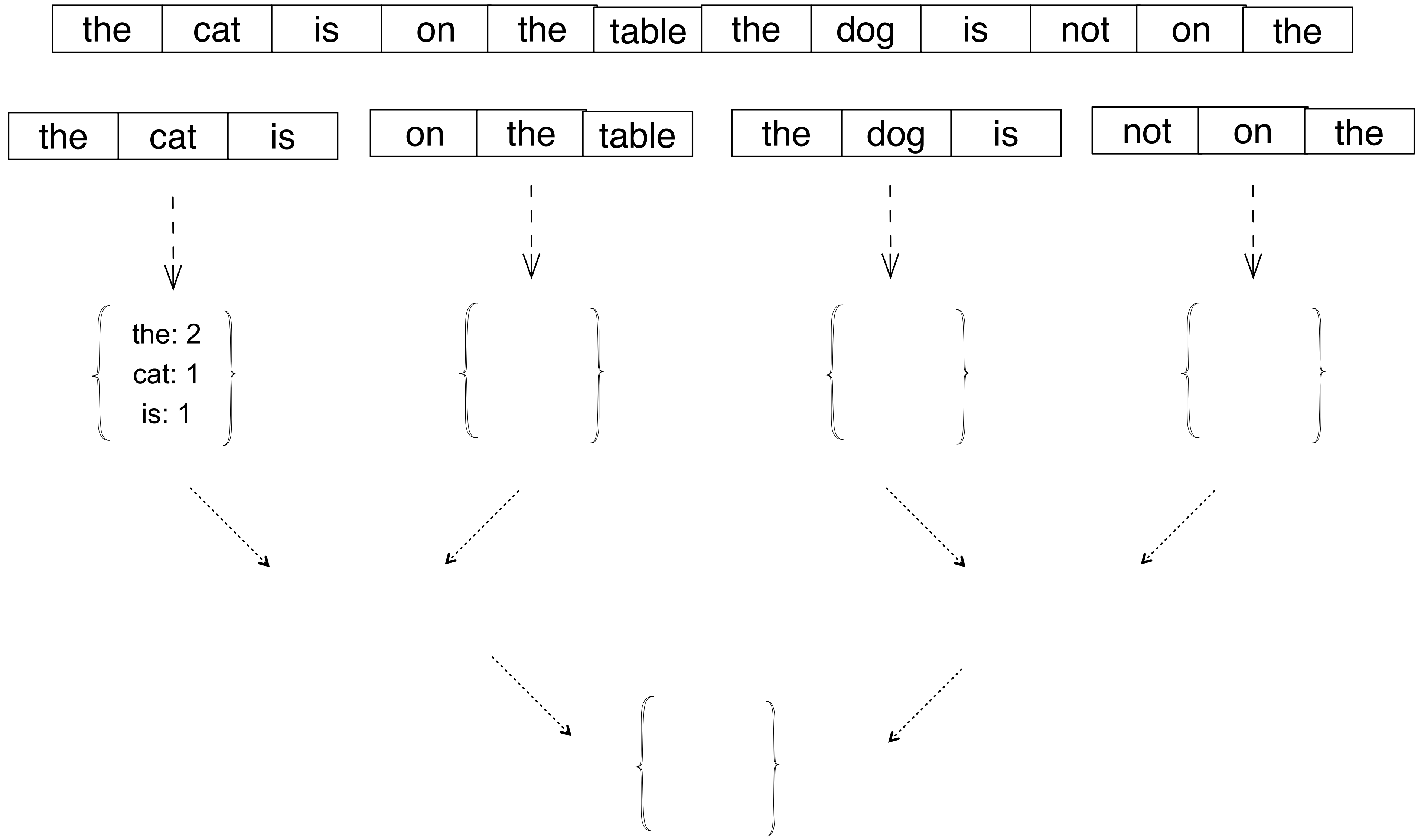
```
user> (def numbers (range 0 10000000))

user> (time (parallel-sum numbers))
"Elapsed time: 21.227375 msecs"

user> (time (recur-sum numbers))
"Elapsed time: 167.805875 msecs"
```

Counting words with fold

- Idea: From the sequence of strings construct a unique sequence of words and fold it with different functions:



Counting words with fold

- Idea: From sequence of strings construct a unique sequence of words and fold with different functions

```
(defn count-words-fold [strings]
  (fold merge-counts counting
          (mapcat get-words strings)))
```

```
(defn counting [counts word]
  (assoc counts
         word (inc (get counts word 0))))

(def merge-counts (partial merge-with +))
```

Laziness

Lazy sequences

- Sequences are lazy in Clojure, elements generated “on demand”
- Very long sequence of integers

```
=> (range 0 1000000)
```

- Generated on demand ...

```
=> (take 2 (range 0 1000000))
```

- Only two (actually a bit more) elements generated
- `Doall` for reifying a sequence

(Lazy) Streams

- Lazy sequences can be infinite
- Iterate: lazy sequence by iterating a function to an initial value

```
(def naturals (iterate inc 0))
```

```
(take 10 naturals)
(0 1 2 3 4 5 6 7 8 9)
```

- Repeatedly apply a function with no arguments

```
(def rand-seq (repeatedly #(rand-int 10)))
```

Delay as much as you can

- When transforming a sequence the actual transformation is only "recorded"

```
(def numbers (range 1000000))
```

```
(def shift (map inc numbers))
```

```
(def doubleshift (map inc shift))
```

- Some real computation only if sequence accessed

```
(take 2 doubleshift)
```

What happens

- Conceptually each lazy sequence **`seq`** is associated with a transforming function **`f`**.

```
<f,seq>
```

- Applying a function to the elements of the sequence just means composing with **`f.`**

```
(map g <f,seq>) -> < g∘f, seq>
```

- See also the concept of reducibles

Clojure

Summary

- A functional Lisp-based language compiled to the JVM
- Functional paradigm goes fine with parallel processing
- Map, Reduce, Fold naturally admit concurrent realisations

Different orders

- Functions are referential transparent: an expression can be replaced by its value without changing the overall behaviour
- Different evaluation orders produce the same results

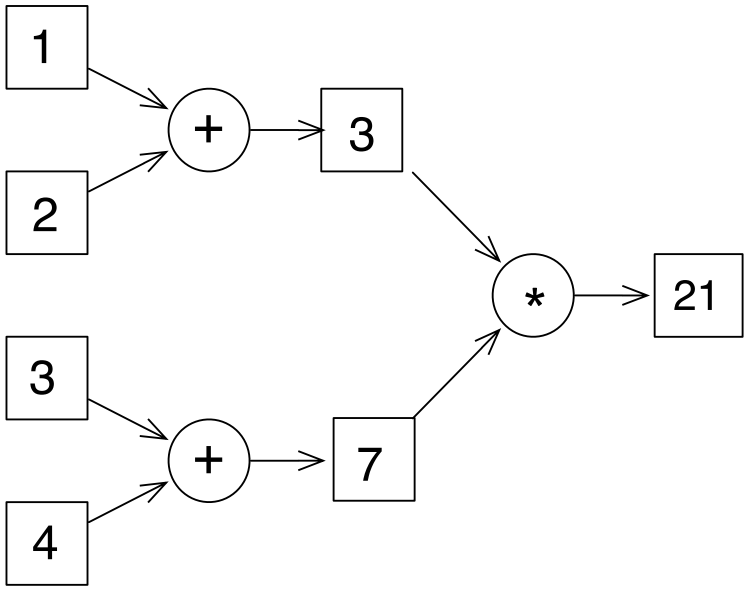
```
(reduce + (doall (map inc (range 1000))))
```

```
(reduce + (pmap inc (range 1000)))
```

```
(fold + (map inc (doall (range 1000))))
```

Self-made concurrency

- Independent expressions – in principle - can be evaluated concurrently



```
(* (+ 1 2) (+ 3 4))
```

- Can we do this?

Future and Promises

Futures

- Intuitively: expression evaluated in a different thread

```
(future Expr)
```

- Realised by an asynchronous concurrent thread
- Value does not immediately exist, (might be) available at some later point

Futures

- Example

```
user=> (def sum (future (+ 1 2 3 4 5)))
#'user/sum
user=> sum
#object[clojure.core$future_call …]
```

- Deref (or **@** for short): get the value

```
user=> (deref sum)
15
```

```
user=> @sum
15
```

Wait until the value is realised (available)

Timing out and checking

- Possibility of timing out when waiting for a value

```
(deref var tout-ms tout-val)
```

- Example

```
user=> (def sum (future (+ 1 2 3 4 5)))
user=> (deref sum 100 :timed_out)
```

- Checking if a future is realised

```
user=> (realized? sum)
```

Promises

- Placeholder for a value realised asynchronously

```
(promise)
```

- (Might be) later written (delivered) only once

- Again deref (@ for short) for getting the value, and realised? for checking availability

Promises

- Example

```
user=> (def answer (promise))
```

```
user=> (deref answer)
```

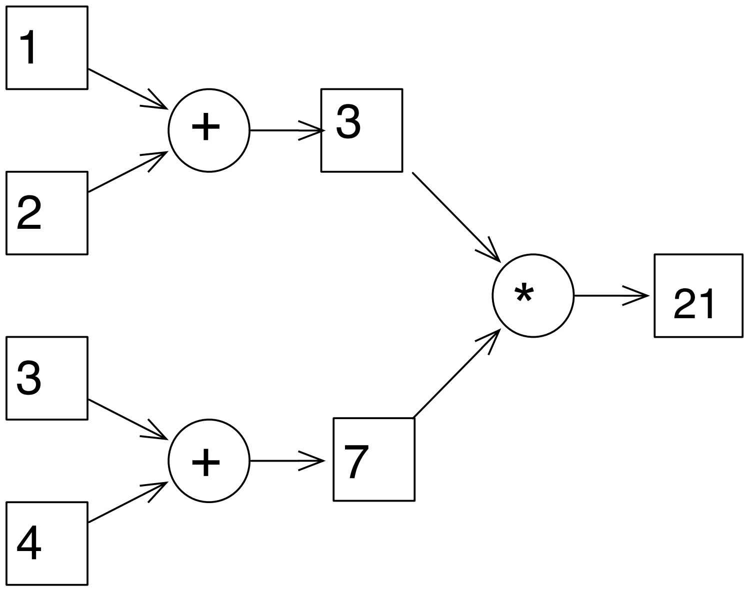
```
user=> (future
           (println "Waiting for answer ... ")
           (println "Here it is" @answer))
Waiting for answer ...

user=> (deliver answer 42)

Here it is 42
```

Self-made concurrency

- Getting back



```
(* (+ 1 2) (+ 3 4))
```

```
(let [ a (future (+ 1 2))
       b (future (+ 3 4)) ]
    (* @a @b))
```

- More generally they can be used to structure concurrent applications

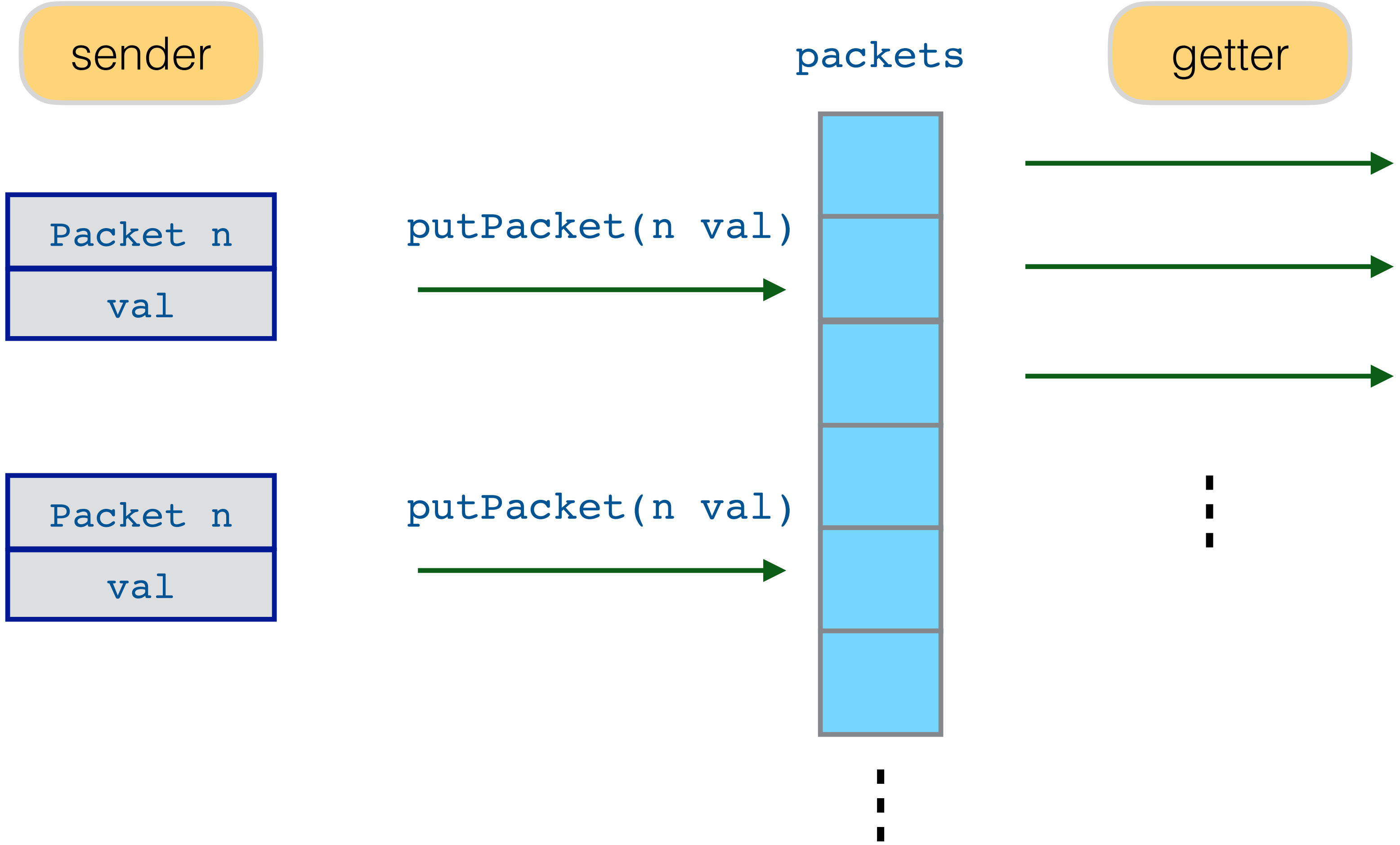
Example

- Call services and wait for the first result

```clojure
(def search-engines
    {:bing   "https://www.bing.com/"
     :google "https://www.google.com/"})


(defn search [search-term & engines]
  (let [result (promise)]
    (doseq [engine engines]
      (future (deliver result
                       (slurp (str (engine search-engines)
                                   "search?q%3D"
                                   search-term)))))
    @result))
```

Example



sender

packets

getter

Packet n

val

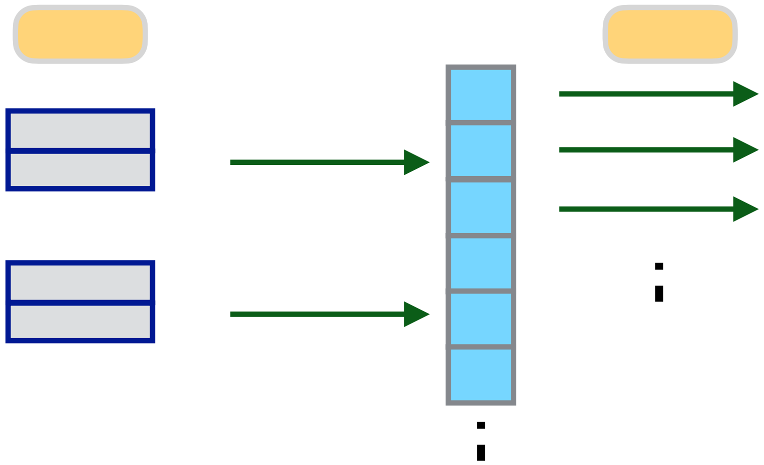
putPacket(n val)

Packet n

val

putPacket(n val)

Receiver



```
; packets are in a lazy sequence of promises
(def packets (repeatedly promise))

; when a packet arrives, the promise is realised
(defn put-packet [n content]
  (deliver (nth packets n) content))

; process taking each packet as long as it is available
; and all its predecessesors have been realised
(defn getter []
  (future
    (doseq [packet (map deref packets)]
      (println (str "*** GETTER: " packet)))))
```

Sender

```
; simulate the sender: split str in words and randomly
; send the words until finished (possibly with repetitions)
(defn send-str [str]
  (let [words (get-words str)
        len   (count words)]
    (send-words words len)))
```

Send words

```
; process that randomly sends the words
; until all have been successfully sent
(defn send-words [words len]
  (future
  (loop [words-to-send len]
    (if (> words-to-send 0)
      (let [n (rand-int len)
            word (nth words n)]

        (if (nil? (put-packet n word))
          (do (println (str "* SENDER:" word " already sent"))
              (recur words-to-send))
          (do (println (str "* SENDER:" word " successfully sent"))
              (recur (dec words-to-send)))
          ))))))
```

Mutable state

Processes

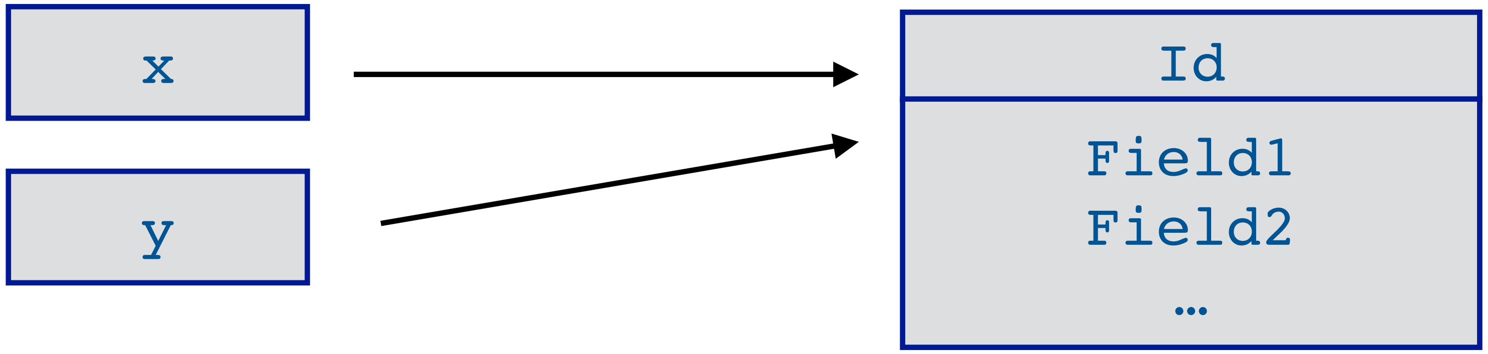
- Dealing with concurrency, the notion of process comes in
- Processes
  - Wait for external events and produce effects on the world
  - Answers change over time
  - Processes have **mutable state**

Identity and state

- **Identity**
  logical entity associated to a series of values
- **State**
  the value of an identity at some time

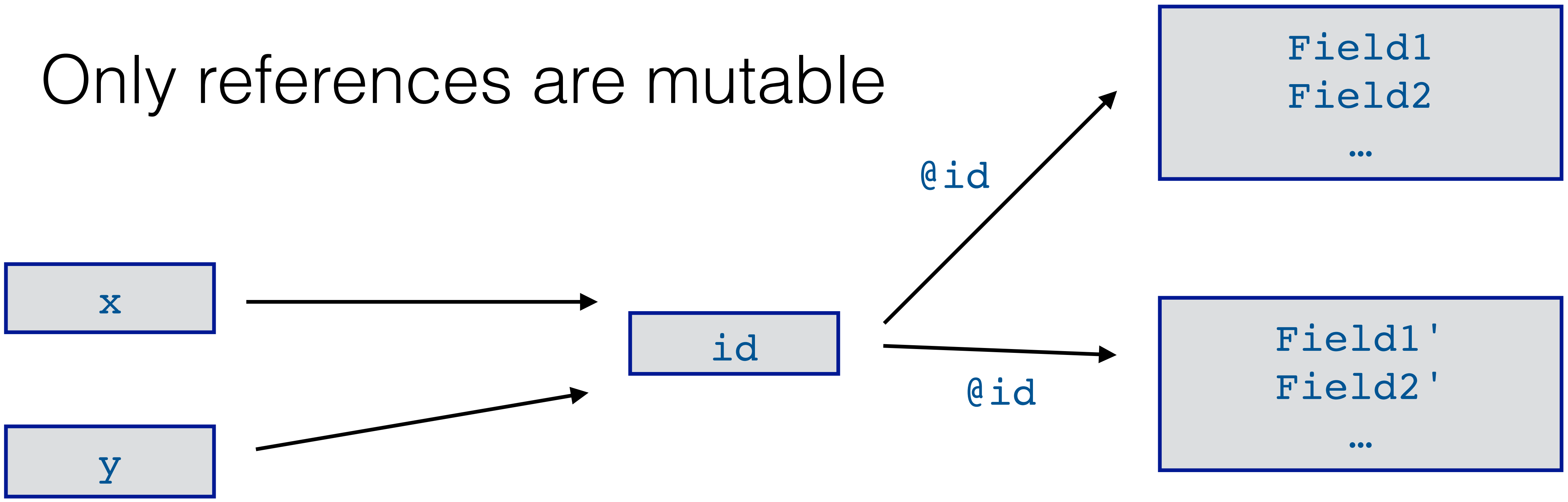
Imperative world

- Identity and state are mixed up
- The state of an identity is changed by **locally modifying** the associated value
- Risk of inconsistency
- Changing state requires **locking**



The Clojure way

- **Identity** and **state** are kept **distinct**
- Symbols refer to **identities** that refers to **immutable values** (never inconsistent)
- Only references are mutable



Mutable references

- Only references change, in a controlled way
- Four types of mutable references:
  - **Atoms** - shared/synchronous/autonomous
  - **Agents** - shared/asynchronous/autonomous
  - **Refs** - shared/synchronous/coordinated
  - (**Vars** - Isolated changes within threads)

Atoms

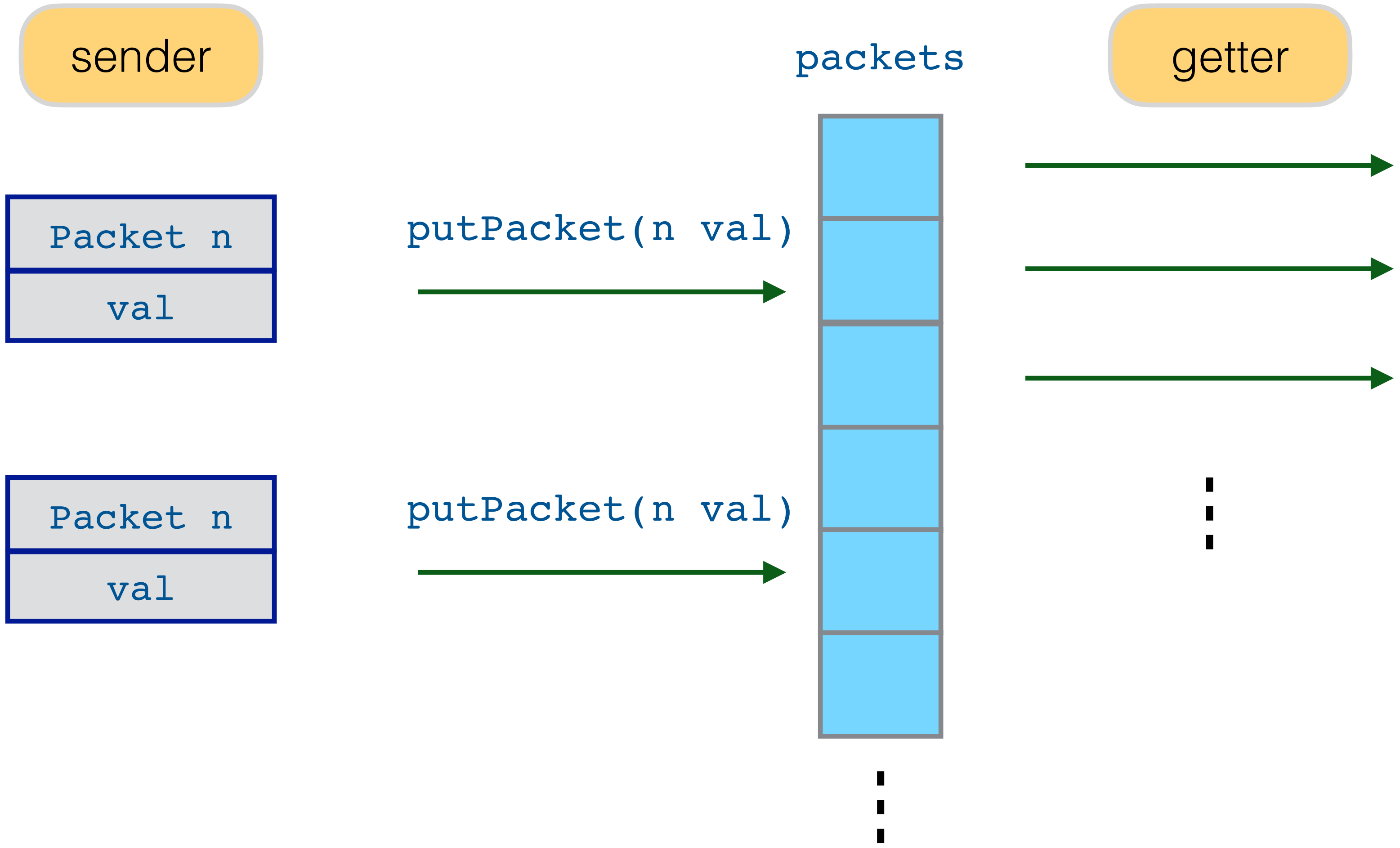
- Atomic update of a **single** shared ref
- Changes occur **synchronously** on caller
    - Change is requested
    - Caller “blocks” until change is completed

Updating Atoms

- `swap! atom f`
    - a function computes the new value from the old one
    - called repeatedly until the value at the beginning matches the one right before change
- `reset! atom new`
  changes without considering old value
- `compare-and-set! atom old new`
  changes only if old value is identical to a specified value

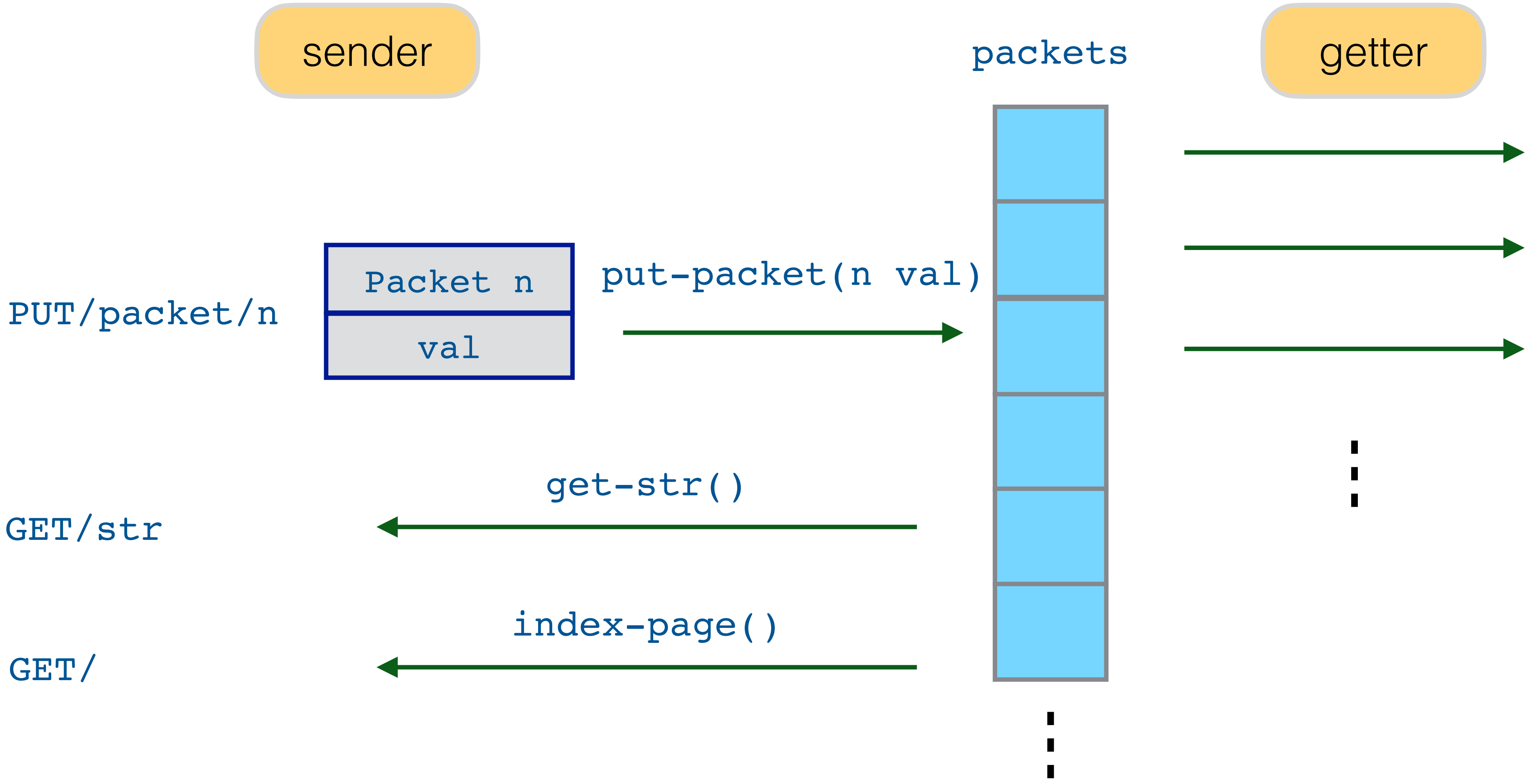
Example

- Consider the packet example



Example

- and turn it into a web service



Routes

```
(defroutes main-routes

  ; receive a packet
  (PUT "/packet/:n" [n :as {:keys [body]}]
      (put-packet (edn/read-string n)
                                  (slurp body)))

  ; get the current string
  (GET "/str" [] (get-str))

  ; getting the current string as an html page
  (GET "/" [] (index-page))
```

Views

```
; state

; current collected string
(def msg (atom ""))

; packets in a lazy sequence of promises
(def packets (repeatedly promise))

; take each packet as long as it is available and all
; its predecessesors have been realised and join it
; the msg
(future
  (doseq [packet (map deref packets)]
    (swap! msg #(str % packet))))
```

Handlers

```
; handler for PUT/packet/n
; when a packet arrives, the promise is realised
(defn put-packet [n content]
  (if (nil? (deliver (nth packets n) content))
    "FAILED\n" "OK\n"))

; handler for GET/str
; client asking for the current string
(defn get-str [] (str @msg \newline))

; handler for GET/
; html page showing the current string
(defn index-page []
  (html5
   [:head [:title "Packets”" ... ]
   [:body [:h1
     (str "String:" @msg)]]]))
```

Agents

- Atomic update of a **single** shared ref
- Changes occur **asynchronously**
  - Change is requested (via **send**) which immediately returns
  - Executed asynchronously (queued and sequentialised on target)
- Useful for performing updates that do not require coordination

Operating on Agents

- `send`
    - a function computes the new value from the old one
    - asynchronously
- `@, deref`
  access the current value (some updates possibly queued)
- `await`
  wait for completion

Example: Back to the server

- In-memory logging could be done via agent

```
; state

; log
(def log-entries (agent []))

....

; adding a log-entry
(def log [entry]
     (send log-entries conj [(now) entry]))
```

Handlers

```
; handler for PUT/packet/n
; when a packet arrives, the promise is realised
(defn put-packet-log [n content]
  (if (nil? (deliver (nth packets n) content))
    (do (log (str "Packet " n " duplicated"))
        "FAILED\n")
    “OK\n")
  )
```

Software Transactional Memory

- Software transactions ~ DB concept
  - **Atomic**
    From outside: either it succeeds and produce all side effects or it fails with no effect
  - **Consistent**
    From a consistent state to a consistent state
  - **Isolated**
    Effect of concurrent transactions is the same as a sequentialised version (some order)

Software Transactions in Clojure

- Based on **refs**

```
(def account (ref 0))
```

- Refs can only be changed within a transaction
- Changes to refs are visible outside only after the end of the transaction

Software Transactions in Clojure

- A transaction can be **rolled back** if
  - it fails
  - it is involved in a deadlock

- A transaction can be retried hence it should avoid side effects (on non refs)

Transactions on refs

- Transactions enclosed in `dosync`

```
; transfers some amount of money between two accounts
(defn transfer [from to amount]
  (dosync
   (alter from - amount)
   (alter to + amount)))
```

- Side effects (ops on **atoms**) shouldn’t be there
- Operation on **agents** executed only upon successful try (agents work well with transactions)

Concluding …

- Functional paradigm pairs nicely with concurrency
- Clojure takes a pragmatic view, it is functional but with support to (controlled) mutable state
- Why don’t we work with imperative languages altogether then?
  - mutable state as an “exception”
  - actually, mutable refs to immutable values

Concluding

- Futures and promises
- Software Transactional Memory
- Sometimes we miss channel based concurrency … also Rich Hickey did (implemented from 1.5)
- No direct support for distribution and fault tolerance (but integration with Java … you have Akka there)