Clojure programming language

Denis Miginsky

LISP

LISP = LISt Processor

First implementation: end of 1950-s, IBM 704

Author: John McCarthy

Basic syntax: LISP program is a sequence of **S**(ymbolic)**-expressions**.
Each S-expression is one of the following:

- an atom (constants, symbols, strings)
- a list of S-expressions separated by spaces (or some other delimiters) and bounded with parentheses
- a bit of optional syntactic sugar

LISP is the first…

- dynamic language
- functional language
- language with abstract data types (lists)
- language with a garbage collector (in 1959!)
- data representation/markup language
- language with macros/meta-programming language
- interactive shell (REPL)

Read Evaluate Print Loop (REPL)

REPL is a prototype for shells and an interactive programming environment. Unlike the "regular" programming technique with explicit source code, compiling and running it, REPL is similar to dialog with so called core.

A core is a set of compiled symbols (functions, variables, etc.). It could be interacted by instructions of the following types:

- a compiling instruction when either a new symbol is introduced or an existing one is replaced
- an evaluating instruction when previously defined functions could be called
- an introspecting instructions
- saving/loading instructions for the whole core

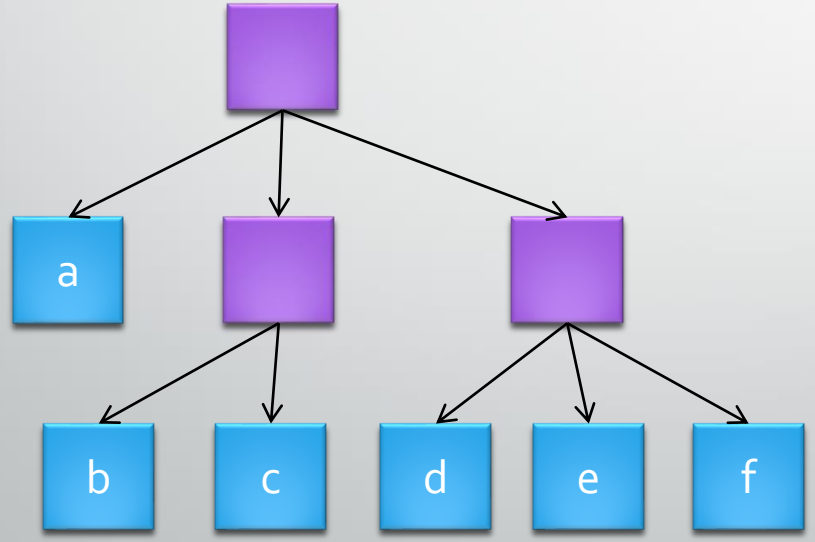
Lists in LISPs

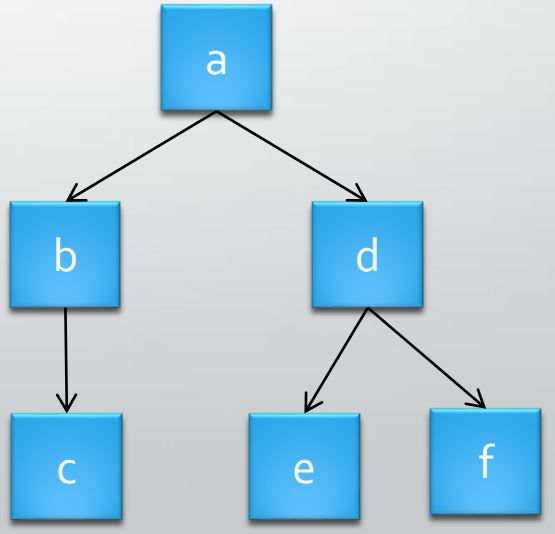
```
;;this is just a comment, not a list

(a b c)                    ;but this is a list
(a (b c) (d e))            ;and this

;;and even this
(if true
    (println "a")
    (println "b"))
;;wait, is it code?
```

Trees

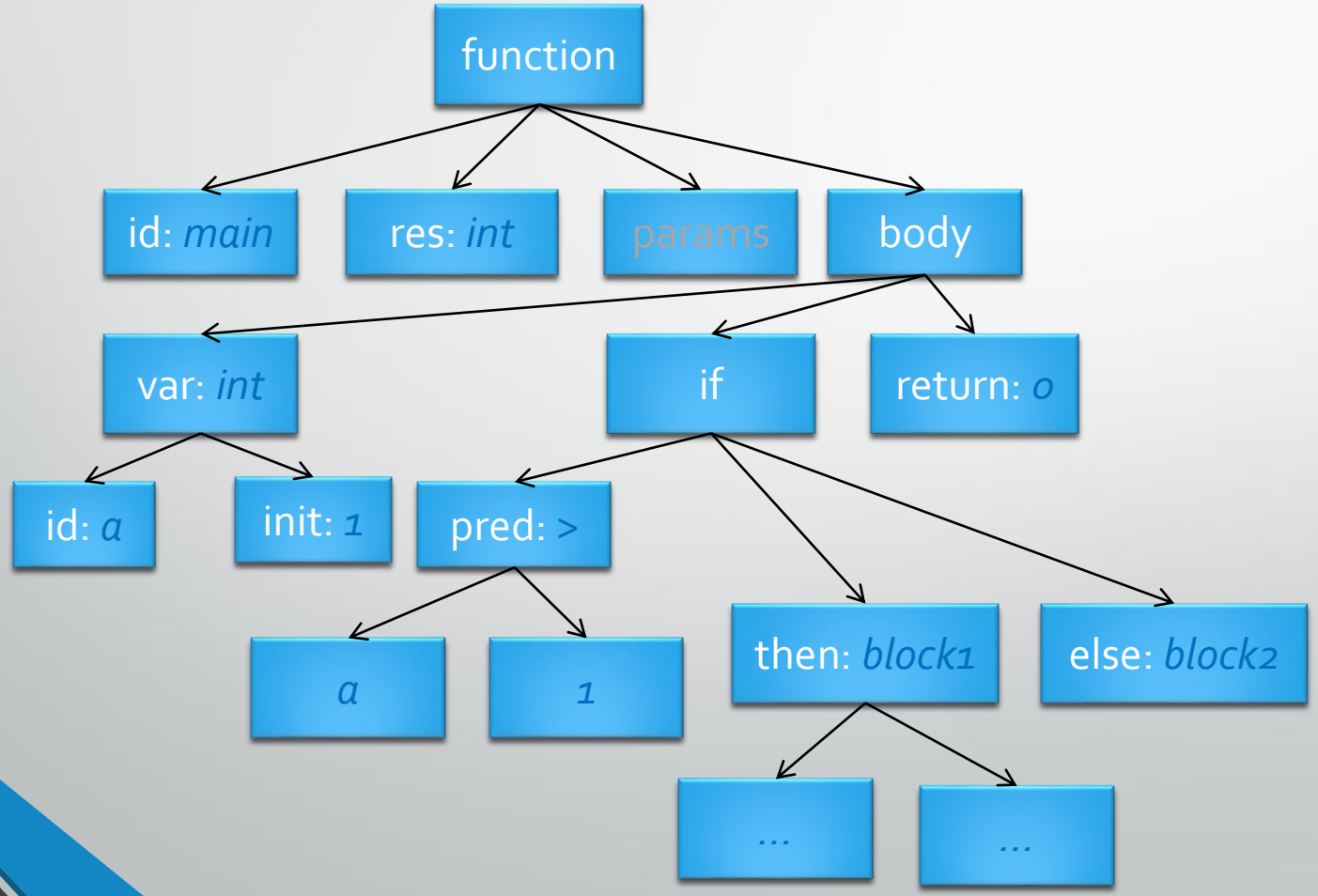
```
'(a '(b c) '(d e f)) ;ignore quotes for a while
```

Possible interpretations:



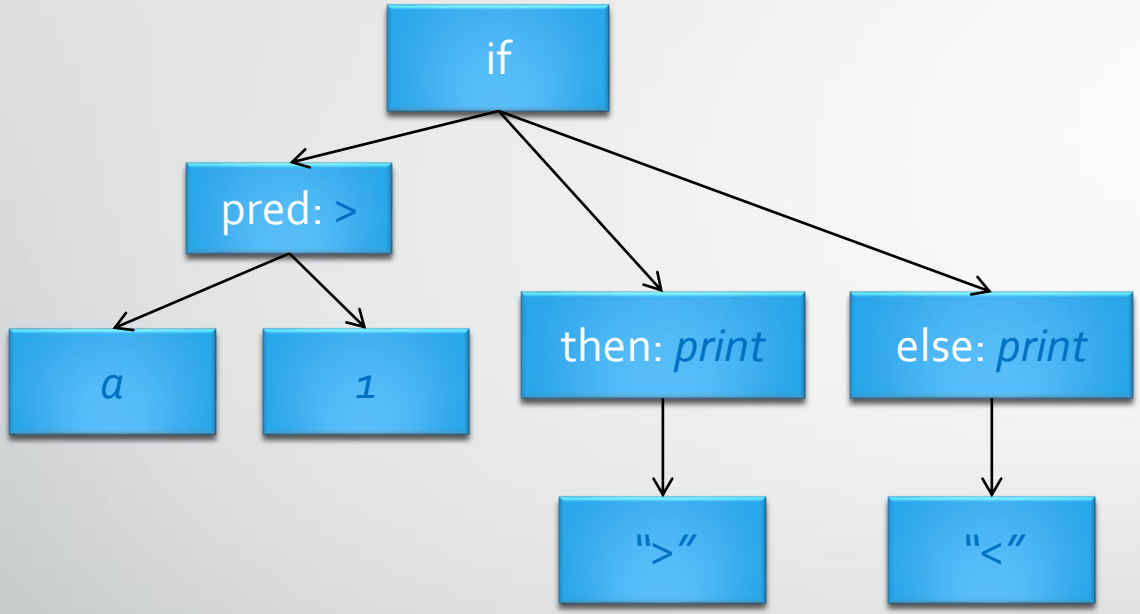


Syntax tree




```
int main (void){
    int a = 1;
    if(a > 1){
       //block1
    }
    else{
       //block2
    }
    return 0;
}
```

Syntax tree: Lisp




```
(if (> a 1)
    (println ">")
    (println "<="))
```

A LISP program is equivalent to a data structure representing its syntax tree.
It is called **homoiconicity**. This is very rare property of programming language.
Other known homoiconic languages: Prolog, Tcl

Functions and special forms

A list is interpreted as a piece of code and evaluated as the following:

1. The first element is evaluated. It must be either function or special form.
2. In case of a function the rest elements of the list must be evaluated and next the function will be applied to them as arguments. It is applicative evaluation order.
3. In case of a special form the rest elements will be passed to this form without evaluation (just like a piece of code in the form of a data structure). This is normal evaluation order. The special form itself can decide is it necessary to evaluate them or not.

```
//C
f(x,y)

;;Lisp
(f x y)
```

Q: are there any special forms in C language?

Clojure programming language

The first release: 2009

Author: Richard Hickey

Runtime: JVM (compiling to Java byte-code)

Ranking: practical modern functional Lisp dialect

Key features: pure functional core, native concurrency support, meta-programming support

Relationships with Java: don’t fix something that is not broken

References

Language and documentation:
https://clojure.org/

Online REPL:
https://repl.it/languages/clojure

Possible IDE:
Eclipse + CounterClockwise
https://github.com/ccw-ide/ccw/wiki/GoogleCodeHome

Build automation:
http://leiningen.org/

Lisp arithmetic

There are no infix operators in Lisp. Everything is a function (or a special form) and has to be written in a prefix form.

C:

```
a*x*x + b*x + c
```

Lisp:

```
(+ (* a x x) (* b x) c)
```

Pros	Cons
Unified and very simple syntax	It is difficult to express complex formulas
No problems with priority and associativity	

Function definition

```
;;documentation entry, not a special form signature
(defn fname doc? [args] prepostmap? forms)
```

- **fname** – function name
- **doc** – documentation string
  (? means "optional" in documentation)
- **args** – names of formal parameters
- **prepostmap** – pre- and post-conditions
- **forms** – function code
  ("forms" means multiple S-expressions)

```
(defn sqr
  "Square"
  [x]
  {:pre [(>= x 0)]}
  (* x x))

(defn fact [n]
  (if (> n 1)
    (reduce * (range 2
                     (inc n)))
    1))
```

Variadic and anonymous functions

```
;;variadic function
(defn poly [x & coeffs]
  (reduce
    ;;anonymous function
    ;;aka lambda-expression
    (fn [acc coef]
      (+
        (* acc x)
        coef))
    0
    coeffs))

(println (poly 2 1 2 3))
```

```
;;definition and immediate application
;;of the anonymous function
((fn [x] (+ x 5)) 3)

;;syntactic sugar
;;equivalent to the previous (almost)
(#(+ % 5) 3)
```

Overloading

```
;;standard + works similar to this
(defn sum
  ([] 0)                    ;0 arity
  ([x] x)                   ;1 arity
  ([x y & rest]             ;2+ arity
    (+ x (apply sum (cons y rest)))))

(println (sum 1 2 3))
```

Scope

```
;;defines the set of variables (in functional sense)
(let [a 10,                ;comma is optional, but conventional for let
                           ;comma is simply a separator
      b (+ a 10),          ;variables could be defined using others
      sqr (fn [x] (* x x)),    ;function variable
      inc dec]             ;it is possible to even "override" global symbols
                           ;it is not recommended, however

  (println a)              ;>> 10
  (println b)              ;>> 20
  (println (sqr 5))        ;>> 25
  (println (inc 10)))      ;>> 9
(println b)                ;ERROR, out of the scope
```

Other scope definitions

```
;;the following two statements are equivalent
;;the first is syntactic sugar for the second one in fact
(let [a 10] (println a))
((fn [a] (println a)) 10)


;;syntactic sugar to define function variables
(letfn [(sqr [x] (* x x))]
  (sqr 10))
```

Conditional statements

```
(if (> a 0)                    ;condition
    (println "pos")            ;then
    (println "non-pos"))       ;else (optional)

;;multiple if-else
(println
  (cond
    (> a 0)    "pos"
    (< a 0)    "neg"
    :default   "nil"))

;;Notes
;; - every statement in Clojure returns something
;;    (possibly nil), including conditional expressions
;; - nil or false are considered as false
;; - everything else is considered as true
```

Basic data types and structures

- numbers (integer, real, rational)
- strings and regular expressions
- keywords
- sequences
- sequence-compatible data structures:
  - lists
  - arrays
  - sets
  - maps

Numbers

- Integers and reals are almost the same as in Java.
- Integers could be promoted to **bignum** arithmetic.
- A division operation could promote integers to rational numbers

```
(* 1000000 1000000 1000000 1000000)
;;>> ArithmeticException integer overflow

;;+', *', etc. will promote arithmetic to bignum when necessary
(*' 1000000 1000000 1000000 1000000)
;;>>1000000000000000000000000N

;;/ for integers may produce rational number, be careful!!
(/ 2 3)
;;>> 2/3
(+ 2/3 5/7)
;;>> 29/21
```

Strings and regular expressions

```
;;Regular Java string
"Some string"
(println (class "Some string"))
;;>> java.lang.String

;;regular Java method for String
(.toUpperCase "abc")
;;>> ABC

;;Clojure function for string concatenation
;;Usually it is more convenient than Java .concat
(str "aa" "bb" "cc")
;;>> aabbcc

#"(?:Regex matching)?(Some string)"
(re-matches #"(a.)\s(\w+)" "ab cd")
;;>> ["ab cd" "ab" "cd"]
```

Lists

```
;;Create the immutable list and evaluate its elements
(list 1 2 3 4 5)        ;regular list of integers
;;>> (1 2 3 4 5)
(list mod 10 8)         ;the list that is also a piece of code
;;>> (#<core$mod clojure.core$mod@63e43ac3> 10 8)
;;Create and evaluate the list
(mod 10 8)
;;>> 2
;;Evaluate the previously created list
(let [l (list mod 10 8)]
  (eval l))
;;>> 2
;;Create the list without evaluating elements (mod will not be checked
;;and resolved as a function)
'(mod 10 8)
;;>> (mod 10 8)
```

Sequences

Sequence is a generalised term for all collection types in Clojure (similar to Collection interface in Java)

There are the following types of "mutator" functions for collections:

- **Regular sequence operations**. They can work with any types of sequences but will not preserve their types
- **Monomorphic operations** for particular collection type. Preserve the type.
- **Polymorphic operations**. Can work with different types of collections preserving their types.

Sequence operations: examples

```
(class (list 1 2 3))
;;>> clojure.lang.PersistentList
(cons 0 (list 1 2 3))          ;construct adds a new head
;;>> (0 1 2 3)
(class (cons 0 (list 1 2 3))) ;it's a sequence, but not a list
;;>> clojure.lang.Cons
;;concatenation of two or more sequences (don't mess with String.concat)
(class (concat (list 1 2) (list 3 4))) ;not a list, once again
;;>> clojure.lang.LazySeq
(conj (list 1 2 3) 0)      ;conjoin adds a new head, works similar to cons
;;>> (0 1 2 3)
(class (conj (list 1 2 3) 0)) ;but it preserves the list itself
;;>> clojure.lang.PersistentList
```

Basic sequence/list operations

```
;;takes a head of the given sequence
(first '(1 2 3))        ;>> 1
;;takes a tail of the given sequence
(next '(1 2 3))         ;>> (2 3)
(rest '(1 2 3))         ;>> (2 3)
;;takes a sequence element by its index
(nth '(1 2 3) 2)        ;>> 3
;;concatenation of two or more sequences
(concat '(1 2) '(3 4)))   ;>> (1 2 3 4)
;;construct adds a new head to the existing sequence
(cons 0 '(1 2 3))    ;>> (0 1 2 3)
;;conjoin is similar to cons, but with reversed arguments'
;;order and preserves  the list (it is polymorphic in fact)
(conj '(1 2 3) 0)    ;>> (0 1 2 3)
;;takes first n elements from the given sequence
(take 2 '(1 2 3))    ;>> (1 2)

;;useful link:
;;https://clojure.org/api/cheatsheet
```

Functional-style collections processing

```
;;per-element mapping
(map (fn [x] (* x x)) '(1 2 3))      ;>> (1 4 9)
;;map with binary function and two collections
(map + '(1 2 3) '(3 2 1))      ;>> (4 4 4)
;;operation with sequences, may break lists
(class (map inc '(1 2 3))      ;>> clojure.lang.LazySeq
(filter (fn [x] (= 0 (mod x 2)))
  '(1 2 3 4 5))                ;>> (2 4)
(remove (fn [x] (= 0 (mod x 2)))
  '(1 2 3 4 5))                ;>> (1 3 5)
;;similar to foldl in Haskell
(reduce * 1 (range 1 6))       ;>> 120      ;== 5!
```

Java basic interaction

```
(class "Ordinary java string")  ;>>java.lang.String

;;java calls
(.toUpperCase "abc")          ;>> "ABC"
(. "abc" toUpperCase)         ;>> "ABC"
(.concat "ab" "cd")           ;>> "abcd"
(.substring "abcd" 1 3)       ;>> "bc"

(new HashMap)                 ;>> Java class instantiation
;;call chain
(.. "str" (getClass) (getName)) ;>> java.lang.String
(.getName (.getClass "str" ))   ;the same without sugar

;multiple operations with the same object
(doto (new HashMap) (.put "a" 1) (.put "b" 2))
(let [m (new HashMap)]              ;the same as previous
  (.put m "a" 1)                      ;but without sugar
  (.put m "b" 2))
```

Java basic interactions

```
(ns ru.nsu.fit.dt          ;header for almost every Clojure source file
  ;clojure namespace import
  (:use clojure.test)
  ;Java classes import
  (:import (java.io File PrintWriter)))

;namespace-qualified Clojure call
(clojure.xml/parse "file.xml")
;Java class static call and static attribute access
(Math/sin Math/PI)

;assignment (for Java attributes only!)
(set! MyClass/someStaticVar 42)
```

Task C1

Given an alphabet in form of a list containing 1-character strings and a number **N**. Define a function that returns all the possible strings of length **N** based on this alphabet and containing no equal subsequent characters.

Use **map/reduce/filter/remove** operations and basic operations for lists such as **str**, **cons**, .**concat**, etc.
No recursion, generators or advanced functions such as **flatten**!

Example: for the alphabet **("a" "b " "c")** and **N**=2 the result bust be **("ab" "ac" "ba" "bc" "ca" "cb")** up to permutation.

Tail recursion

Clojure doesn't support tail recursion optimization.
But it supports explicit tail recursion with recur special form.

```
;;regular recursion
;;can't be optimized to tail variant
(defn fact-1 [n]
  (if (> n 0)
    (* n (fact-1 (dec n)))
    1))

;;invalid tail recursion
;;will produce compile ERROR
(defn fact-2 [n]
  (if (> n 0)
    (* n (recur (dec n)))
    1))
```

```
;;regular recursion
;;can be optimized to tail variant
(defn fact-3
  ([n] (fact-3 n 1))
  ([n acc] (if (> n 0)
             (fact-3 (dec n) (* n acc))
             acc)))

;;tail recursion, finally
(defn fact-4
  ([n] (fact-4 n 1))
  ([n acc] (if (> n 0)
             (recur (dec n) (* n acc))
             acc)))
```

Generator

```
;;generator
(for [x (range 1 6),          ;x and y will pass through all the
      y (range 1 6),          ;elements of [1..5] X [1..5]
      :let [x2 (* x x),       ;auxiliary variables
            y2 (* y y)]
      :when (and (= 0 (mod x 2))      ;additional filter
                 (not= 0 (mod y 2)))]
  (+ x2 y2))                  ;mapping for each element
;;>> (5 13 29 17 25 41)
```

Vectors

Vectors are sequences, but unlike lists they are not treated as executable code.

```
;;vector constructors
[1 2 3]
(vector 1 2 3)
(vec (list 1 2 3))
;;accessing the element by its index
(nth [1 2 3] 1)       ;>>2
;;replacing an element by its index (associate), preserves the vector
(assoc [1 2 3] 1 4) ;>>[1 4 3]
;;appending an element (conjoin), preserves the vector
(conj [1 2 3] 4)     ;>>[1 2 3 4]
;;prepending an element to the sequence
(cons 4 [1 2 3])     ;>>(4 1 2 3)
;;map that produces vector
(mapv - [1 2 3])     ;>>[-1 -2 -3]
```

Maps

```
;;hash-map constructors
{:a 1, :b 2}
(hash-map :a 1, :b 2)
;;signatures for tree-map constructors
(sorted-map & keyvals)
(sorted-map-by comparator & keyvals)
;;mutators (associate, dissociate)
(assoc {:a 1} :b 2)          ;>>{:a 1, :b 2}
(dissoc {:a 1, :b 2} :b)     ;>>{:a 1}
(merge {:a 1, :b 2} {:b 3, :c 2})      ;>>{:c 2, :b 3, :a 1}
;;map itself works as an accessor function
({:a 1, :b 2} :a)            ;>>1
;;keywords can also works (in case they are used as keys)
(:a {:a 1, :b 2})            ;>>1
;;map can be considered as a sequence of pairs
(seq {:a 1, :b 2})           ;>> ([:a 1] [:b 2])
;;special reduce variant for maps
(reduce-kv (fn [acc _ v] (+ acc v))
           0
           {:a 1, :b 2, :c 3})     ;>>6
```

Sets

```
;;hash-set constructors
#{1 2 3}
(hash-set 1 2 3)
;;signatures for tree-set constructors
(sorted-set & vals)
(sorted-set-by comparator & vals)
;;mutators (conjoin, disjoin)
(conj #{1 2 3} 4)          ;>>#{1 4 3 2}
(disj #{1 2 3} 1)          ;>>#{3 2}
;; accessor
(contains? #{1 2} 3)             ;>>false
;;…
(ns my-ns
;; namespace with advanced set and map functions
   (:use clojure.set))
(intersection #{1 2} #{2 3})   ;>>#{2}
;;union, join, select, …
```

Sequential execution

- For **imperative programming** the sequential execution is natural.
- For **pure functional programming** it is useless. All the statements besides the last one is useless in this case.
- **Clojure** is functional, but it allows side effects.

```
;;for some special forms the
;;sequential execution is enabled
;;by default
(fn [x]
  ;;useless, returns a value into nowhere
  (dec x)
  ;;only useful because of side effects
  (println x)
  ;;useful because its value is returned
  ;;by fn
  (inc x))
```

```
;;other special forms takes the
;;only form and doesn't allow
;;sequential execution
(if (> a 0)
   ;;it can be enabled by do
   (do
     (println "pos")
     true)
   (do
     (println "neg")
     false))
```

Operation chains

```
;;common situation, but
;;very unwieldy
(assoc
  (assoc
    (assoc {}
      :a 1)
    :b 2)
  :c 3)

;;more convenient variant
(-> {}
  (assoc :a 1)
  (assoc :b 2)
  (assoc :c 3))
```

```
;;another unwieldy statement
(reduce +
        (map (fn [x] (* x x))
             (filter even?
                     (range 1 100))))

;;looks better if the form
;;of the stream scenario
(->> (range 1 100)
     (filter even?)
     (map (fn [x] (* x x)))
     (reduce +))
```

Destructuring

```
;;destructuring is a weaker form of pattern matching
(let [v [1 [2 3 4] 5 6 7],
      [a [b & c] d & e] v]
  (println a)           ;>>1
  (println b)           ;>>2
  (println c)           ;>>(3 4)
  (println d)           ;>>5
  (println e))          ;>>(6 7)

;;destructuring is also in use in function calls
(defn my-f [[x y & rest] val] …)
(my-f [1 2 3 4] 5)

;;very useful usage pattern
(map (fn [[_ val]] val)
     {:a 1, :b 2, :c 3})       ;>>(3 2 1)
```

Unit-testing in Clojure: target code

```
(ns ru.nsu.fit.dt.defpackage)


(defn fact [n]
  (if (> n 1)
    (reduce * (range 2 (inc n)))
    1))
```

Unit-testing in Clojure: test code

```
(ns ru.nsu.fit.dt.defpackage-test
 (:use ru.nsu.fit.dt.defpackage)
 (:require [clojure.test :as test]))


(test/deftest defpackage-test
  (test/testing "Testing defpackage"
    (test/is (= (fact 0) 1))
    (test/is (= (fact 1) 1))
    (test/is (= (fact 2) 2))
    (test/is (= (fact 5) 120))))

;;…

(run-tests 'ru.nsu.fit.dt.defpackage-test)
```

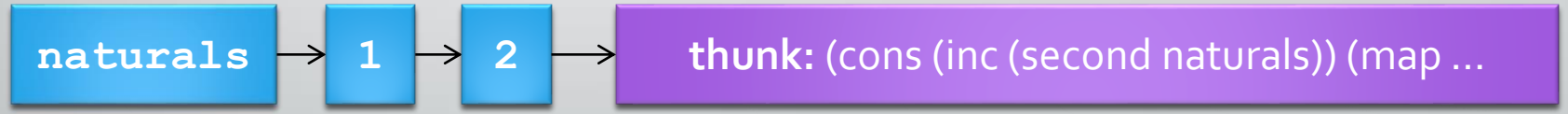
Lazy evaluations

```
;;sequence of natural numbers, infinite
(def naturals
  (lazy-seq
    (cons 1 (map inc naturals))))

(nth naturals 10)
;;>> 11

;;another possible variant
(nth (iterate inc 1) 10)
```

Infinite sequences and "thunks"

```
(def naturals
  (lazy-seq
    (cons 1 (map inc naturals))))
```


```
(nth naturals 10)
```



Fibonacci sequence

```
;;"straightforward" definition via iterate
(let [fibs (->> (iterate (fn [[v1 v2]]
                          [v2 (+ v1 v2)])
                        [0 1])
                (map first))]
  (nth fibs 10))     ;>>55

;;recursive definition
;;its more elegant and little bit more efficient
(def fibs
  (lazy-cat '(0 1)
    (map + fibs (rest fibs))))
```

Lazy constructors

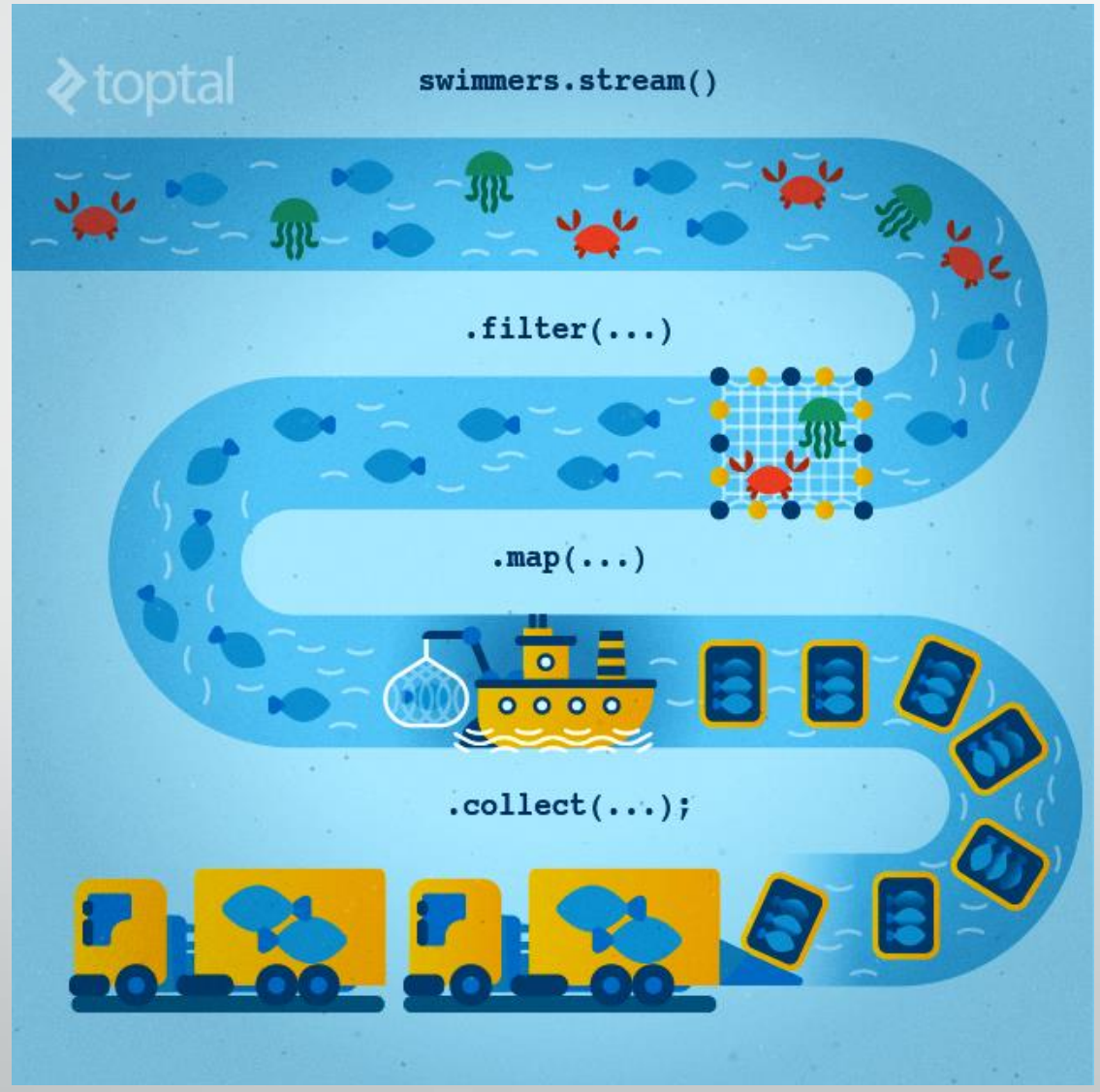
```
;;sequence basic cast
;;usually this is applied automatically
(seq [1 2 3])              ;>> (1 2 3)
(class (seq [1 2 3]))      ;>>clojure.lang.PersistentVector$ChunkedSeq

;;special form to construct thunk (promise) to compute a sequence
(lazy-seq & body)          ;signature
(lazy-seq [1 2 3])



;;special form to construct thunk to concatenate sequences
;;equivalent to (concat (lazy-seq s1) (lazy-seq s2) …)
(lazy-cat s1 s2 & rest)          ;signature
(lazy-cat [1 2 3] [4 5 6])

;;construct an infinite sequence with recurrent definition
(iterate (partial * 2) 1)        ;>>(1 2 4 8 …)
```

Lazy and eager operations

```
;;Lazy operations

(take 10 (iterate inc 1))

(map (fn [x] (* x x))
     (iterate inc 1))

(filter

(for

(rest
(next
```

```
;;Eager operations

;; Who wants to live forever?
;;                    -- F. Mercury
(reduce + (iterate inc 1))
(count (iterate inc 1))
;;eager for the new head only
(cons 0 (iterate inc 1))
(first …
(nth …
;;forces the sequence to be evaluated
(doall seq)
(dorun seq)
;;similar to for, but designed for
;;side effects and returns nothing
(doseq seq-expr & body)
```

Fishing with Java Stream API



Streams

```
;;sequence as a stream
(->> (iterate inc 1)                ;;infinite stream of values
     (filter even?)                 ;;transformation stream->stream
     (map (fn [x] (* x x)))         ;;once again
     (take 100)                     ;;limiting the stream
     (reduce +))                    ;;a consumer, finally

;;basic raw file stream
(with-open [fl (clojure.java.io/reader
                "/tmp/my_file.txt")]
  ;;the same stream, but in terms of lines (useful for text files)
  (->> (line-seq fl)
       (map (fn [x] (Integer/parseInt x)))
       (filter (fn [x] (= 0 (mod x 2))))
       (map (fn [x] (* x x)))
       (take 10)))
```

Task C2

Define the infinite sequence of prime numbers. Use Sieve of Eratosthenes algorithm with infinite cap.

Cover code with unit tests.

Data modification in FP

In pure FP any data structure is immutable.
Functional "mutators" produce new data structure with applied modifications by copying the original one.
Consider 1M-element list and one is going to put 1000 new elements step-by-step, i.e. with **cons**.

Problem 1: all the intermediate results are not necessary and can produce memory leaks.
Solution: **garbage collector** that is essential for any functional language.
Problem 2: constant copying is expensive both in terms of memory consumption and performance.
Solution: **Persistent Data Structures**

Persistent Data Structures

Persistent Data Structure (PDS) is a data structure that keeps its previous state when modified.
For each modification a new version is created and all the versions are stored simultaneously, sharing as much elements as possible.

Unused versions are deleted by a garbage collector, as usual.

Example: Persistent list and Fat node

```
(let [a (list :a :a :a)

      b (list :b :b :b)


      c (cons :c a)
      d (rest a)



      e (concat b a)
...
```