Дисклеймер

> Чистая совесть дорого стоит. —Saul Goodman

Только попытка научить других показывает, насколько плохо ты сам понимаешь чем занимаешься. Я уверен что допустил много глупых неточностей, которые режут слух знающему человеку. Но я продолжу идти по граблям.

Работа над ошибками

Keyword

Я не нашел подтверждения своим словам про то что каждый кейворд уникальный Почитав офф доки, могу исправить определение

Кейворд - константа, как и символ

- которая очень дешево тестируется с другими кейвордами,
- позволяет расширять семантику за счет неймспейсов
- может применяться вместо функции get

```
(= :a :a)
;; => true

(= :b/a :b/a)
;; => true

(= :a :b)
;; => false

(:a {:a 10})
;; => 10
```

Lists

Из списков можно брать значения по индексу

```
(nth (list 1 2 3 4 5) 0)
;; => 1
```

Список - последовательный, связанный список Добавляет новые значения в начало списка, а не в конец, как вектор.

Предисловие по языку

Функциональный

Поощряется функциональный подход. Все имеет результат. Можно управлять функциями как значениями. Можно композировать функции.

Чистая функция - функция, которая при одинаковых входных данных, всегда дает одинаковый результат

```
(defn pure [a] (+ a a))

[(pure 2) (pure 2) (pure 2) (pure 2)]
;; => [4 4 4 4]

(defn pure! [a] (rand-int a))
```

Свобода в создании и использовании функций

```
(fn [])
;; => fn

(defn closure [a]
  (fn []
    (+ a 10)))
;; #'user/closure

(closure 5)
;; => (fn [] (5 + 10))

((closure 5))
;; => 15
```

Почему бы нам не сделать словарь функций

```
(def funcs
  {:plus5 (fn [a] (+ a 5))
   :minus10 (fn [b] (- b 10))})

((:plus5 funcs) 10)
;; => 15

((:minus10 funcs) 10)
;; => 0
```

Иммутабельный

Мы не мутируем (перезаписываем) значение переменной. Мы создаем копию или меняем ссылку на объект

```
int i = 5;
i = 6;
// 6
```

```
(def i 5)

(def i* (inc i))

[i i*]
;; => [5 6]
```

Ленивый

Если какое то значение не используется На него не будут затрачены ресурсы

Но можно заставить вычислить, если мы хотим гарантий в сложных местах например функциями [doall run! doseq]

```
(defmacro bench [& body]
  `(with-out-str
     (time
      ~@body)))

(bench (take 100000 (repeatedly #(rand-int 100))))
;; => "\"Elapsed time: 0.023464 msecs\"\n"
;; => "\"Elapsed time: 0.023835 msecs\"\n"
;; => "\"Elapsed time: 0.046217 msecs\"\n"
;; => "\"Elapsed time: 0.02632 msecs\"\n"

(bench (doall (take 100000 (repeatedly #(rand-int 100)))))
;; => "\"Elapsed time: 11.117991 msecs\"\n"
;; => "\"Elapsed time: 9.675811 msecs\"\n"
;; => "\"Elapsed time: 5.69942 msecs\"\n"
;; => "\"Elapsed time: 9.814532 msecs\"\n"

(bench
 (doall
  (let [a (range 10)
```

```
        b (range 100)])))
;; => "\"Elapsed time: 0.010209 msecs\"\n"
;; => "\"Elapsed time: 0.010019 msecs\"\n"
;; => "\"Elapsed time: 0.006863 msecs\"\n"

(bench
 (doall
  (let [a (range 10)
        b (range 100)]
    [a b])))
;; => "\"Elapsed time: 0.058229 msecs\"\n"
;; => "\"Elapsed time: 0.117121 msecs\"\n"
;; => "\"Elapsed time: 0.091814 msecs\"\n"
```

Персистентные структуры

Структура данных, как гит репозиторий. Хранит значение при инициализации, и шаги воспроизведения. Мы редко копируем что-то полностью, чаще мы меняем последние шаги работы со структурой Поэтому иммутабельность, стоит дешево.

```
(def v [1 2])

(def v2 (conj v 3))
;; [1 2] + 3 = [1 2 3]

(def v3 (conj v2 4))
;; [1 2] + 3 + 4 = [1 2 3 4]

(def v4 (conj v2 5))
;; [1 2] + 3 + 5 = [1 2 3 5]

v
;; => [1 2]

v2
;; => [1 2 3]

v3
;; => [1 2 3 4]

v4
;; => [1 2 3 5]
```

Data Driven

> Меня греет мысль, что бриллиант и женщина, надевшая его на палец, состоят из одного и того же элемента. —Walter White

Мы работаем с данными. Мы получаем данные, и мы отдаем данные. Иногда чтобы что-то сделать, не нужно изобретать абстракцию Почти всегда достаточно того, что предлагает язык.

Концептуальные ограничения, делают наш код болле продуманным, а следовательно надежным и боллее чистым.

Синтаксические ограничения, делают наш код громостским, а следовательно менее надежным и трудночитаемым.

Примеры

1. Юзер

   Допустим, мы хотим сделать какого юзера Будем хранить его в мапке, не создавая класс или что-то еще

   ```
   (def person
     {:name  "A"
      :lname "B"
      :bdate "10.10.1980"
      :age   43})

   (:age person)
   ;; => 43

   (def person-with-inced-age
     (update person-with-bdate
             :age inc))
   ;; => {:name "A", :lname "B", :age 44}
   ```

2. Много юзеров

   Допустим, идея с юзерам понравилась бизнесу, и мы должны хранить много юзеров Теперь у нас будет функция, чтобы сделать из разных данных, юзера

   ```
   (defn ->person [& [name lname age bdate]]
     {:name  name
      :lname lname
      :age   age
      :bdate bdate})
   ```

```
(->person "C" "D" 44 "10.10.1979")
;; => {:name "C", :lname "D", :age 44, :bdate "10.10.1979"}

(->person "V" "F" 45 "10.10.1978")
;; => {:name "V", :lname "F", :age 45, :bdate "10.10.1978"}

(->person 1 2 3 4 5)
```

3. Юзер в SQL

   В какой то момент, мы поняли что все равно будем хранить в базе И написали функцию, которая перевод хэшмапу в валидный SQL запрос

```
(require '[clojure.string :as str])
(defn sql-for-persons [person]
  (format "INSERT INTO PERSONS %s VALUES %s;"
          (seq (mapv name (keys person)))
          (vals person)))

(sql-for-persons (->person "V" "F" 45 "10.10.1978"))
;; => "INSERT INTO PERSONS (\"name\" \"lname\" \"age\" \"bdate\") VALUES (\"V\" \"F\" 45 \"
```

4. Валидация

   Плохо жить без валидации Мы сделали функцию validate!, которая принимает хэшмапы, с набором ключей и функциями, которые нужно применить к ключам

   Сделали хэшмапу person? которая хранит в себе ключи и функции для них

```
(def person?
  {:name string?
   :lname string?
   :age int?
   :bdate string?})

(defn validate! [vmap data]
  (not (some (fn [[k v]] (not ((k vmap) v))) data)))

(validate! person? (->person "C" "D" 44 "10.10.1979"))
;; => true
```

   В какой то момент мы поняли что для некоторых юзеров у нас другая логика валидации Как удобно, что мы заложили разные мапы-валидации на этапе проектирования

```
(def person?*
  {:name string?
   :lname string?
   :age (some-fn nil? int?)
   :bdate (some-fn nil? string?)})

(validate! person?* (->person "C" "D"))
;; => true

(validate! person? (->person "C" "D"))
;; => false
```

История одного проекта

Как то довелось мне писать движок 2D графики на голом DOM дереве в JavaScript (CLJS SVG API vDOM React|Reagent) Это был проект с большой кодовой базой, где я развлекался в создании велосипедов От самописных ORM'ок до систем группировок для виртуального DOM (Это был сложный технический калькулятор, посредством визуального программирования, как фигма)

Самым большим профитом Clojure/ClojureScript была возможность превращать громостские данные, в очень удобные структуры

1. Генерация SVG элементов из мап

   Из обычных хэшмап

```
(def css-for-rect [rect]
  {:width  :0.5vw
   :height :1vh
   :color  (random-color)})

{:shape :rect
 :id    :shape1}

<svg>
  <rect class="rect" id="shape1">
</svg>

.rect {
    width:  0.5vw;
    height: 1vh;
}

#shape1 {color: "#000";}
```

2. Масштабирование

```
(def state
  [{:shape :rect
    :id    :shape1}
   {:shape :circle
    :id    :shape2}
   {:shape :star
    :id    :shape3}
   {:shape :triangle
    :id    :shape4}])

(render-shapes! state)
```

```
<svg>
  <rect class="rect" id="shape1">
  <circle id="shape2">
  <star id="shape3">
  <triangle id="shape4">
</svg>
```

3. Вложенность

Со временем появилась потребность в во вложенных фигурах:

- с наследованием позиций на странице
- или стилей
- или особым порядком
- или высшим приоритетом

```
(def state
  [{:shape :rect
    :id    :shape1
    :order 0
    :parent :shape4
    :children [:shape2 :shape3]}
   {:shape :circle
    :id    :shape2}
   {:shape :star
    :id    :shape3}
   {:shape :triangle
    :id    :shape4}])

(defn render! [& shapes]
  (-> shapes
      (sort-by-order)
      (create-pretty-stuff)
      (generate-links)
```

```
      (compute-values)
      (optimize)
      (render)))
```

```
<svg>
  <triangle id="shape4">
  <!-- rect-triangle -->
  <line/>
  <!-- to-circle -->
  <line/>
  <!-- to-star -->
  <line/>
  <div id="group">
    <rect class="rect" id="shape1">
    <circle id="shape2">
    <star id="shape3">
  </div>
</svg>
```

4. DSL

   Структуры росли в размерах и сложности обхода Могли быть рекурсивными, с разными опциями обхода

   В конце концов родился мини-язык внутри мини языка

```
(def state
  [{:shape :rect
    :id    :shape1
    :order 0
    :parent :shape4
    :children [:shape2 :shape3]}
   {:shape :circle
    :id    :shape2
    :opts {:x 0
           :y 50}}
   {:shape :star
    :id    :shape3
    :opts {:x 10
           :y 50}}
   {:shape :triangle
    :id    :shape4}])

(defn transform-in [path fn-map val]
   (-> (parse-path path)
       (apply-fn-map val)
       (reduce-back)))
```

```
(transform-in
 [:shape1 :children :coll/all]
 {:opts {:x #(+ 100 %)
         :y #(/ % 2)}
  :order inc
  :links :shape1}
 state)

;; [{:shape :rect
;;   :id    :shape1
;;   :order 0
;;   :parent :shape4
;;   :children [:shape2 :shape3]}
;;  {:shape :circle
;;   :id    :shape2
;;   :order 1
;;   :opts {:x 100
;;          :y 25}
;;   :links :shape1}
;;  {:shape :star
;;   :id    :shape3
;;   :order 1
;;   :opts {:x 110
;;          :y 25}
;;   :links :shape1}
;;  {:shape :triangle
;;   :id    :shape4}]
```

Repl

Чаще всего этой аббревиатурой характеризуется интерактивная среда языка программирования LISP, однако такая форма характерна и для интерактивных сред языков

- Erlang
- Groovy
- Haskell
- Java
- JavaScript
- Perl
- PHP
- Python
- Ruby
- Scala

- Smalltalk
- Swift
- Tcl и других

R ead

Функция read читает одно выражение и преобразует его в соответствующую структуру данных в памяти

E val

Функция eval принимает одну такую структуру данных и вычисляет соответствующее ей выражение

P rint

Функция print принимает результат вычисления выражения и печатает его пользователю

L oop

Бесконечный цикл (loop), начинается сначала

Козыри пошли

Деструктуризация

Синтаксическая возможность ”раскладывать” элементы массива (и не только) в отдельные константы или переменные

JavaScript

```
const [a, b] = [1, 2];
const { a, b } = {"a": 1, "b": 2};
```

Простая деструктуризация

```
(let [[a b c] [1 2 3]]
  b)
;; => 2

(let [[a & _] [1 2 3]]
  a)
;; => 1

(let [[_ & bc] [1 2 3]]
  bc)
;; => (2 3)
```

Продвинутая деструктуризация

```
(defn destruct
  [[a b c]
   {:keys [k1 k2]}
   {:strs [s1 s2]}]
  [a b c k1 k2 s1 s2])

(destruct [1 2 3]
          {:k1 4 :k2 5}
          {"s1" 6 "s2" 7})
```

А как же циклы?

Пример цикла

который соберет сумму чисел от 1 до 10

```
int sum = 0;
for (int i = 1; i < 11; i++) {
    sum += i;
}
// sum = 55
```

Функция range сгенерирует список чисел от 1 до 11, которые мы сложим через +

```
(apply + (range 1 11))
;; => 55
```

Пример цикла

который соберет первые 5 чисел массива array

```
int[] array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int[] acc;
int counter = 0;
while (counter < 5) {
    counter += 1;
    acc[counter] = array[counter]
}
// acc = {0, 1, 2, 3, 4}
```

Функция range сгенерирует ленивую бесконечную коллекцию, из которой мы через take возьмем первые 5 элементов

```
(take 5 (range))
;; => (0 1 2 3 4)
```

Работа с коллекциями

Map

```
List<Integer> integers = List.of(1, 2, 3, 4, 5);
Function<Integer, Integer> doubleFunction = i -> i * 2;

integers.stream()
    .map(doubleFunction)
    .collect(Collectors.toList());
//[2, 4, 6, 8, 10]
```

```
(def integers [1 2 3 4 5])
(defn double-function [i] (* i 2))

(map double-function integers)
;; => (2 4 6 8 10)
```

Но скорее всего в кложе мы бы сделали вот так Используя данные

```
(map #(* % 2) [1 2 3 4 5])
;; => (2 4 6 8 10)
```

Filter

```
Arrays.asList('x', 'y', '2', '3', 'a').stream()
   .filter(Character::isLetter)
// ['x', 'y', 'a']

Arrays.asList(1, 2, 3, 4).stream()
   .filter(x -> x%2 == 1)
// [1, 3]

Arrays.asList("abc", "", "d").stream()
   .filter(s -> !s.isEmpty())
// ["abc", "d"]
```

```
(filter #(Character/isLetter %) [\x \y \2 \3 \a])
;; => (\x \y \a)

(filter #(= (rem % 2) 1) [1 2 3 4])
;; => (1 3)

(filter not-empty ["abc", "", "d"])
;; => ("abc" "d")
```

Reduce

```
Arrays.asList(1,2,3).stream()
    .reduce(0, (x,y) -> x+y)
// computes (((0+1)+2)+3) to produce the integer 6

Arrays.asList(5, 8, 3, 1).stream()
    .reduce(Math::max)
// computes max(max(max(5,8),3),1) and returns an Optional<Integer> value containing 8
```

```
(reduce + [1 2 3])
;; => 6

(reduce max [5 8 3 1])
;; => 8
```

Стрелочки

```
;; NOTE: эти формы равны между собой.
;; стрелочный макрос в любом случае превратится, в форму выше
```

```clojure
(+ 1 (count (conj 6 (conj 5 [1 2 3 4]))))

(+ 1
   (count
    (conj 6
          (conj 5
                [1 2 3 4]))))

(->> [1 2 3 4]
     (conj 5) ;; => [1 2 3 4 5]
     (conj 6) ;; => [1 2 3 4 5 6]
     (count) ;; => 6
     (+ 1)) ;; => 7
```

Задачи

max min avg

Заполните массив случайным числами и выведите максимальное, минимальное и среднее значение.

```java
// https://habr.com/ru/articles/440436/
public static void main(String[] args) {

    int n = 100;
    double[] array = new double[n];
    for (int i = 0; i < array.length; i++) {
        array[i] = Math.random();
    }

    double max = array[0]; // Массив не должен быть пустым
    double min = array[0];
    double avg = 0;
    for (int i = 0; i < array.length; i++) {
        if(max < array[i])
            max = array[i];
        if(min > array[i])
            min = array[i];
        avg += array[i]/array.length;
    }

    System.out.println("max = " + max);
```

```
    System.out.println("min = " + min);
    System.out.println("avg = " + avg);
}
```

```
(let [arr (repeatedly 100 #(rand-int 100))]
  {:min (reduce min arr)
   :max (reduce max arr)
   :avg (-> (reduce + arr)
            (/ (count arr))
            (int))})
```

bubble sort

Реализуйте алгоритм сортировки пузырьком для сортировки массива

```
// https://habr.com/ru/articles/440436/
for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array.length - i - 1; j++) {
        if (array[j] > array[j + 1]) {
            double temp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = temp;
        }
    }
}

for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

```
;; https://eddmann.com/posts/bubble-sort-in-clojure/

(defn- bubble [ys x]
  (if-let [y (peek ys)]
    (if (> y x)
      (conj (pop ys) x y)
      (conj ys x))
    [x]))

(defn bubble-sort [xs]
  (let [ys (reduce bubble [] xs)]
    (if (= xs ys)
      xs
      (recur ys))))
```

```
(bubble-sort [3 2 1])
```

prime numbers

Напишите программу, которая выводит на консоль простые числа в промежутке от [2, 100]. Используйте для решения этой задачи оператор "%" (остаток от деления) и циклы.

```
// https://habr.com/ru/articles/440436/
for (int i = 2; i <= 100; i++) {
  boolean isPrime = true;

  for (int j = 2; j < i; j++) {
    if (i % j == 0) {
      isPrime = false;
      break;
    }
  }

  if (isPrime) {
    System.out.println(i);
  }
}
```

```
(defn prime? [num]
  (when (and (not= num 1)
             (->> (range 2 num)
                  (map #(zero? (rem num %)))
                  (some true?)
                  (not)))
    num))

(remove nil? (map prime? (range 2 100)))
```

remove element

Дан массив целых чисел и ещё одно целое число. Удалите все вхождения этого числа из массива (пропусков быть не должно).

```
// https://habr.com/ru/articles/440436/
for (int i = 2; i <= 100; i++) {
public static void main(String[] args) {
```

```
  int test_array[] = {0, 1, 2, 2, 3, 0, 4, 2};
  /*
      Arrays.toString:
      см. https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html
   */
  System.out.println(Arrays.toString(removeElement(test_array, 3)));
}

public static int[] removeElement(int[] nums, int val) {
  int offset = 0;

  for (int i = 0; i < nums.length; i++) {
    if (nums[i] == val) {
      offset++;
    } else {
      nums[i - offset] = nums[i];
    }
  }

  // Arrays.copyOf копирует значение из массива nums в новый массив
  // с длинной nums.length - offset
  return Arrays.copyOf(nums, nums.length - offset);
}
```

```
(remove #(= % 3) [0 1 2 2 3 0 4 2])
```

kv swap

Напишите метод, который получает на вход Map<K, V> и возвращает Map, где ключи и значения поменяны местами. Так как значения могут совпадать, то тип значения в Map будет уже не K, а Collection<K>

```
// https://habr.com/ru/articles/440436/
public static <K, V> Map<V, Collection<K>>
inverse(Map<? extends K, ? extends V> map) {
  Map<V, Collection<K>> resultMap = new HashMap<>();

  Set<K> keys = map.keySet();
  for (K key : keys) {
    V value = map.get(key);
    resultMap.compute(value, (v, ks) -> {
      if (ks == null) {
        ks = new HashSet<>();
      }
```

```
      ks.add(key);
      return ks;
    });
  }

  return resultMap;
}
```

```
(as-> {:a 1 :b 1 :c 2 :d 2 :e 4} m
  (map (fn [[k v]] [v k]) m)
  (group-by first m)
  (update-vals m #(mapv second %)))
```

ООР

Реализовать иерархию классов, описывающую трёхмерные фигуры

как я понял, смысл ооп в данном случае в том что мы не дублируем код, а наследуемся от классов

а сама задача звучит так, есть 3 класса, которые наследуются от базовых нужно реализовать коробку, которая понимает, влезет ли фигура в нее

```
// https://habr.com/ru/articles/440436/
class Shape {
    private double volume;

    public Shape(double volume) {
        this.volume = volume;
    }

    public double getVolume() {
        return volume;
    }
}

class SolidOfRevolution extends Shape {
    private double radius;

    public SolidOfRevolution(double volume, double radius) {
        super(volume);
        this.radius = radius;
    }

    public double getRadius() {
```

```
        return radius;
    }
}

class Ball extends SolidOfRevolution { // конкретный класс
    public Ball(double radius) {
        super(Math.PI * Math.pow(radius, 3) * 4 / 3, radius);
    }
}

class Cylinder extends SolidOfRevolution { // конкретный класс
    private double height;

    public Cylinder(double radius, double height) {
        super(Math.PI * radius * radius * height, radius);
        this.height = height;
    }
}

class Pyramid extends Shape{
    private double height;
    private double s; // площадь основания

    public Pyramid(double height, double s) {
        super(height * s * 4 / 3);
        this.height = height;
        this.s = s;
    }
}


class Box extends Shape {
    private ArrayList<Shape> shapes = new ArrayList<>();
    private double available;

    public Box(double available) {
        super(available);
        this.available = available;
    }

    public boolean add(Shape shape) {
        if (available >= shape.getVolume()) {
            shapes.add(shape);
            available -= shape.getVolume();
            return true;
        } else {
```

```
            return false;
        }
    }
}

public class Main {

    public static void main(String[] args) {
        Ball ball = new Ball(4.5);
        Cylinder cylyinder = new Cylinder(2, 2);
        Pyramid pyramid = new Pyramid(100, 100);

        Box box = new Box(1000);

        System.out.println(box.add(ball)); // ok
        System.out.println(box.add(cylyinder)); // ok
        System.out.println(box.add(pyramid)); // failed
    }
}
```

строки 86 vs 33 символы 1939 vs 764

```
(defn ball [r]
  {:volume (* Math/PI (Math/pow r 3) (/ 4 3))
   :radius r})

(defn cylinder [r h]
  {:volume (* Math/PI r r h)
   :radius r})

(defn pyramid [h s]
  {:volume (* h s (/ 4 3))
   :height h
   :s      s})

(defn put-into-box [size-]
  (fn [& shapes-]
    (loop [shapes shapes-
           in-box []
           size size-]
      (let [[fst & other] shapes
            size* (- size (:volume fst 0))]
        (if (or (not fst)
                (<= size* 0))
          {:in-box     in-box
```

```
          :didn't-fit (when fst
                        (vec (conj other fst)))}
         (recur other
                (conj in-box fst)
                size*))))))

(let [put (put-into-box 1000)]
  (put (ball 4.5)
       (cylinder 2 2)
       (pyramid 100 100)))

;; {:in-box
;;  [{:volume 381.70350741115976, :radius 4.5}
;;   {:volume 25.132741228718345, :radius 2}],
;;  :didn't-fit [{:volume 40000/3, :height 100, :s 100}]}
```

Java

Interop

```
(.toUpperCase "fred")
;; => "FRED"

(.getName String)
;; => "java.lang.String"

(.-x (java.awt.Point. 1 2))
;; => 1

(System/getProperty "java.vm.version")
;; => "1.6.0_07-b06-57"

Math/PI
;; => 3.141592653589793
```

Библиотеки

```
(ns demo
  (:import (java.util Date Calendar)
           (java.net URI ServerSocket)
           java.sql.DriverManager))
```

Исключения

```
(try
  (/ 1 0)
  (catch Exception e
    (.getMessage e))
  (finally ()))
```

Реальный кейс

Нужно получить набор данных [:mask :ip :hosts :broadcast :prefix :hostmin :wildcart :network :hostmax] В бинарном формате

Входные данные :ip и :mask в формате ipv4:

- ip "10.12.1.2"
- mask "240.0.0.0"

Функции

```
(ns calc
  (:require [clojure.string :as str]))

(let [bit-inv-table #(case % 0 1 1 0 :error)
      prefix        #(try (reduce + %) (catch Exception _ {:error :wrong-num}))
      fill-zeros    #(str/join "" (repeat % 0))
      extend        #(->> % count (- 8) fill-zeros ((fn [zeros] (str zeros %))))
      ->int         #(Integer/parseInt %)
      ->bin         #(Integer/toBinaryString %)
      ->bin-num     #(->> % (map ->bin) (map extend) (str/join "."))
      bin->int      #(Integer/parseInt % 2)
      split*        #(try (->> (str/split %  #"\.") (map ->int))
                          (catch Exception _ {:error :wrong-input}))
      invert        #(->> % ->bin extend (re-seq  #"\d") (map ->int) (map bit-inv-table) (str/jo
      update-last   #(->> %1 last %2 (conj (pop (vec %1))))
      view          (fn [v] [v (->bin-num v)])])
```

Значения

```
(ns calc
  (:require [clojure.string :as str]))
```

```clojure
(let [ip         (split* "10.12.1.2")
      mask       (split* "240.0.0.0")
      prefix*    (prefix (map (comp count #(re-seq #"1" %) extend ->bin) mask))
      wildcart   (->> mask (map invert) (map bin->int))
      hosts      (->> prefix* (- 32) (Math/pow 2) (#(- % 2)))
      network    (map bit-and ip mask)
      broadscast (map bit-or network wildcart)
      hostmin    (update-last network inc)
      hostmax    (update-last broadscast dec)])
```

Результат

```clojure
{:ip (view ip)
 :mask (view mask)
 :prefix prefix*
 :wildcart (view wildcart)
 :hosts hosts
 :network (view network)
 :broadcast (view broadscast)
 :hostmin (view hostmin)
 :hostmax (view hostmax)}
```

```clojure
'{:mask      [(240 0 0 0) "11110000.00000000.00000000.00000000"]
  :ip        [(10 12 1 2) "00001010.00001100.00000001.00000010"]
  :broadcast [(15 255 255 255) "00001111.11111111.11111111.11111111"]
  :hostmin   [[0 0 0 1] "00000000.00000000.00000000.00000001"]
  :wildcart  [(15 255 255 255) "00001111.11111111.11111111.11111111"]
  :network   [(0 0 0 0) "00000000.00000000.00000000.00000000"]
  :hostmax   [[15 255 255 254] "00001111.11111111.11111111.11111110"]
  :hosts     2.68435454E8
  :prefix    4}
```

Монстр

```clojure
(ns calc
  (:require [clojure.string :as str]))

(try
  (let [bit-inv-table #(case % 0 1 1 0 :error)
        prefix        #(try (reduce + %) (catch Exception _ {:error :wrong-num}))
        fill-zeros    #(str/join "" (repeat % 0))
        extend        #(->> % count (- 8) fill-zeros ((fn [zeros] (str zeros %))))
```

```
        ->int         #(Integer/parseInt %)
        ->bin         #(Integer/toBinaryString %)
        ->bin-num     #(->> % (map ->bin) (map extend) (str/join "."))
        bin->int      #(Integer/parseInt % 2)
        split*        #(try (->> (str/split %  #"\.") (map ->int))
                            (catch Exception _ {:error :wrong-input}))
        invert        #(->> % ->bin extend (re-seq  #"\d") (map ->int) (map bit-inv-table) (str/
        update-last   #(->> %1 last %2 (conj (pop (vec %1))))
        view          (fn [v] [v (->bin-num v)])


        ip         (split* "10.12.1.2")
        mask       (split* "240.0.0.0")
        prefix*    (prefix (map (comp count #(re-seq #"1" %) extend ->bin) mask))
        wildcart   (->> mask (map invert) (map bin->int))
        hosts      (->> prefix* (- 32) (Math/pow 2) (#(- % 2)))
        network    (map bit-and ip mask)
        broadscast (map bit-or network wildcart)
        hostmin    (update-last network inc)
        hostmax    (update-last broadscast dec)]
    {:ip        (view ip)
     :mask      (view mask)
     :prefix    prefix*
     :wildcart  (view wildcart)
     :hosts     hosts
     :network   (view network)
     :broadcast (view broadscast)
     :hostmin   (view hostmin)
     :hostmax   (view hostmax)})
  (catch Exception _ {:error :something-wrong}))

'{:mask      [(240 0 0 0) "11110000.00000000.00000000.00000000"]
  :ip        [(10 12 1 2) "00001010.00001100.00000001.00000010"]
  :broadcast [(15 255 255 255) "00001111.11111111.11111111.11111111"]
  :hostmin   [[0 0 0 1] "00000000.00000000.00000000.00000001"]
  :wildcart  [(15 255 255 255) "00001111.11111111.11111111.11111111"]
  :network   [(0 0 0 0) "00000000.00000000.00000000.00000000"]
  :hostmax   [[15 255 255 254] "00001111.11111111.11111111.11111110"]
  :hosts     2.68435454E8
  :prefix    4}
```