# Основы языков семейства Lisp

Денис С. Мигинский

# Исторический экскурс в Lisp

Первая реализация – конец 1950-х, IBM 704.

Автор – John McCarthy

Базовая структура данных – регистр из двух слов.

Основные операции (asm macros):
**`car reg`** – первое слово регистра
(CAR = **C**ontents of the **A**ddress part of **R**egister number)

**`cdr reg`** – второе слово регистра
(CDR = **C**ontents of the **D**ecrement part of **R**egister number)

# Синтаксис Lisp

S-выражение, S(-ymbolic)-expression – выражение вида **a1** или **(a1 a2 … )**, где **a1 a2** – атомы или другие S-выражения

Атом:

- идентификатор
- константа (строковые , числовые, рег. выражения и т.д.)

Синтаксический макрос – синтаксический сахар

Интерпретация S-выражений

```
;Lisp
(f a b c)

//C
f(a,b,c)
```

# Историческое представление пары и списка в Lisp

«Аксиоматика»:

**`(a . b)`** – конструирование пары

**`(car (a . b)) -> a`**

**`(cdr (a . b)) -> b`**

**`(a . (b . c))`** – почти список (a b c)


**`(a . (b . (c . nil)))`** – список (a b c)

# Списки в Lisp: современное представление (Clojure)

**`(a b c)** – конструирование списка

**(first `(a b c)) -> a** – «голова» списка (~car)

**(rest `(a b c)) -> `(b c)** – «хвост» списка (~cdr)

**(cons `a `(b c)) -> `(a b c)** – **cons**truct

# Влияние Lisp на современные языки

Первый динамический язык

Первый функциональный язык

Первый язык со сборкой мусора (1959 г. !!!)

Первый язык с абстрактными типами данных

Первый язык представления данных

Первый (и до сих пор один из немногих) язык с поддержкой макрогенерации

Первая среда с REPL (shell-оболочкой)

# Read-Eval-Print Loop (REPL)

REPL – интерактивная среда программирования, основанная на бесконечном цикле выполнения перечисленных функций

Единица исполнения – одна инструкция (она же – естественная единица модульности)

Образ – набор скомпилированных символов (переменные, функции и т.д.)

**Виды инструкций:**

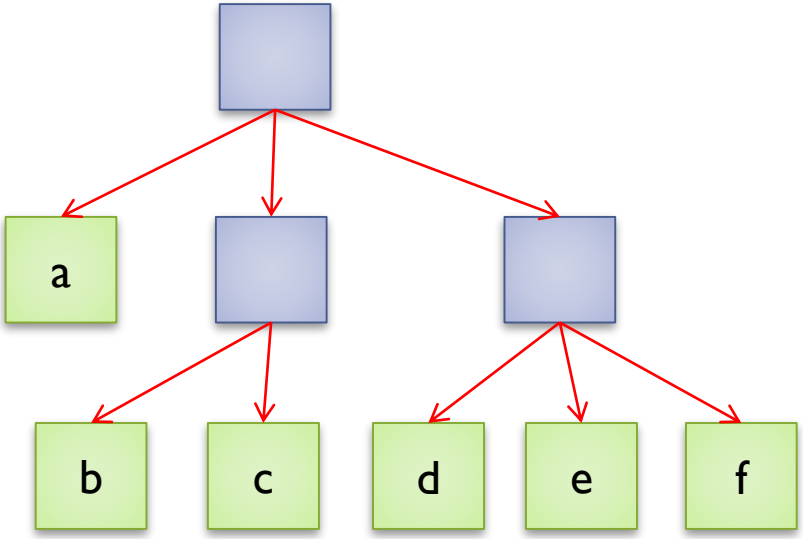
- Вычислительные – порождают результат, не влияют на образ (кроме изменения значений глобальных переменных)
- Определяющие/компилирующие – компилируют новое определение (байт-код, иногда – нативный код) и помещают его в образ (возможно, переопределяют старое)

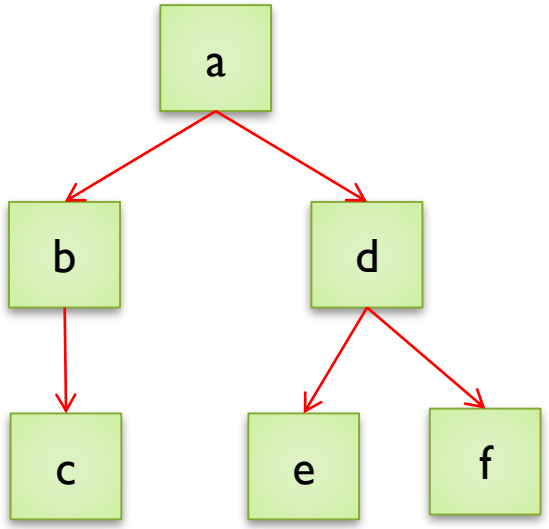
Интроспекция – необходимый инструмент управления образом

# Деревья

```
'(a '(b c) '(d e f))
```

Интерпретации

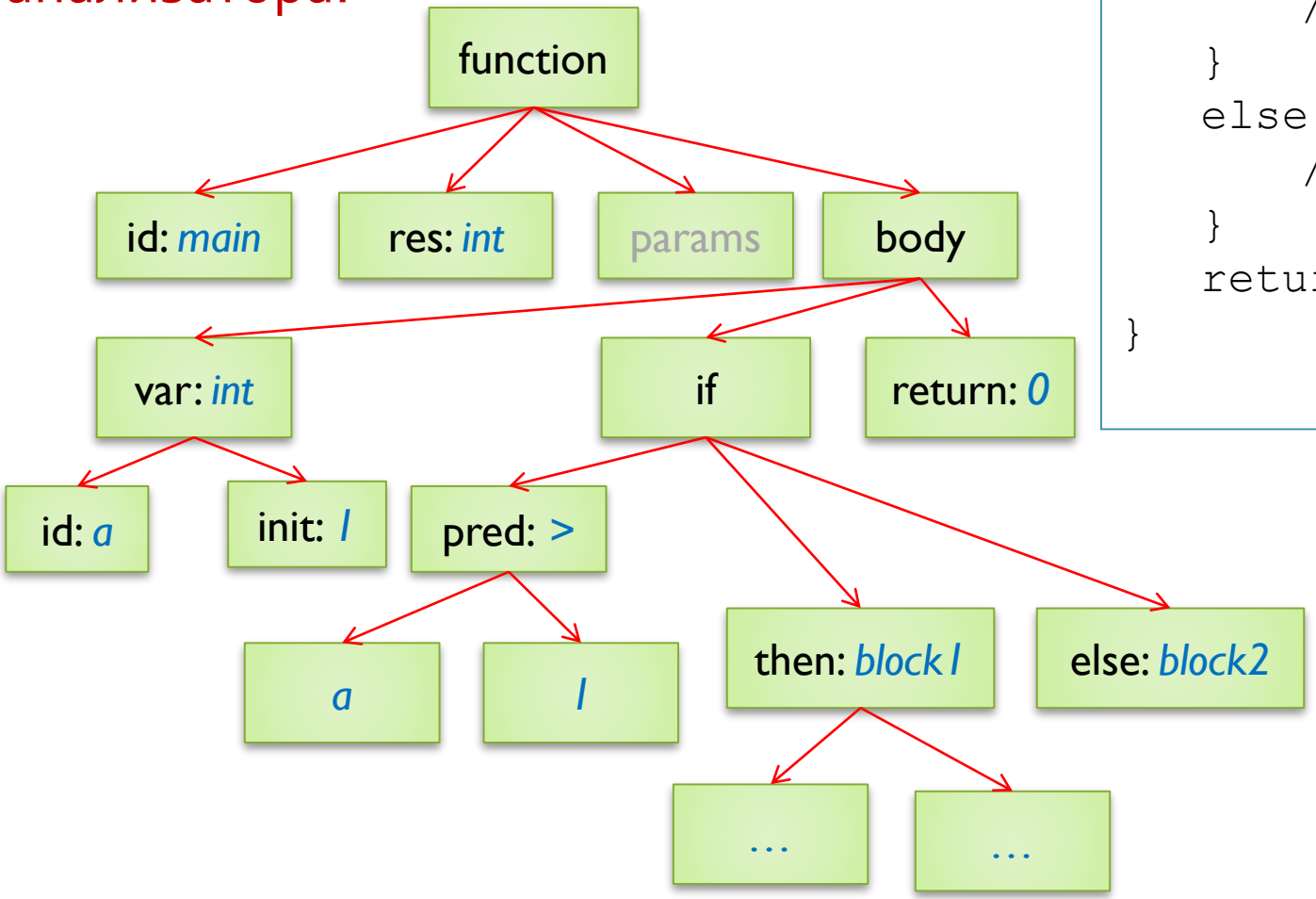


# Абстрактное синтаксическое дерево

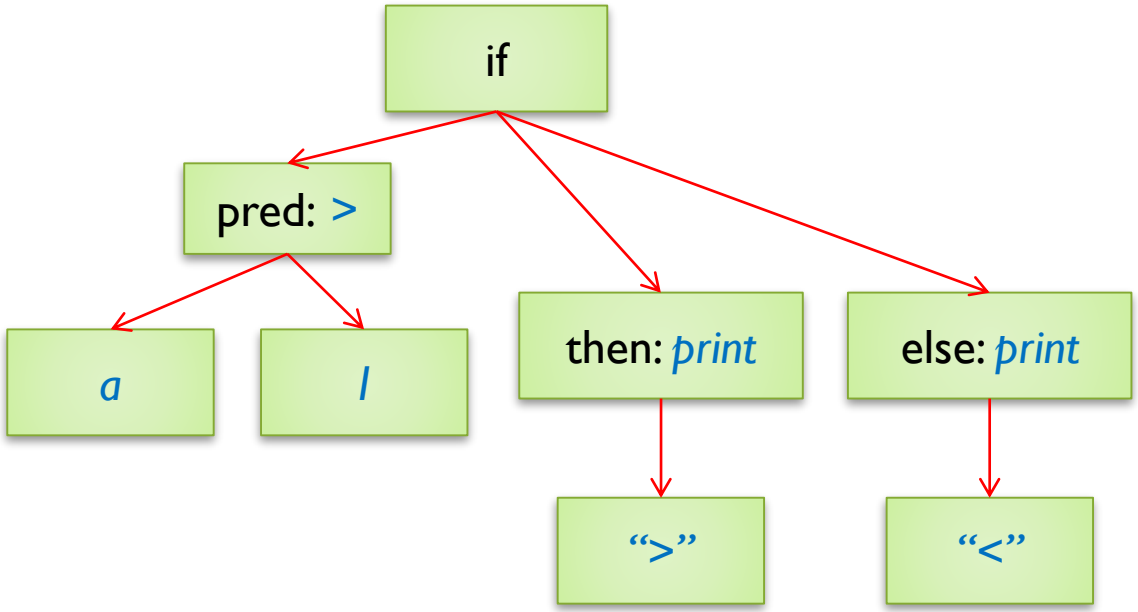
Результат работы синтаксического анализатора:

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



# Связь с абстрактным синтаксическим деревом

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



# Функции и специальные формы

Интерпретация S-выражения: вычисляется первый элемент списка, результат вычислений должен быть либо функцией, либо специальной формой (в противном случае – ошибка компиляции или времени исполнения).

Функция – вычисляются все последующие элементы списка, после чего происходит вызов функции (аппликативный порядок вычислений)

Специальная форма – происходит вызов специальной формы без вычисления аргументов. Спец. форма сама инициирует вычисление аргументов (отложенный порядок вычислений).
*Примеры : управляющие конструкции, логические связки.*

# Язык Clojure

Первый релиз: 2009 г.

Автор: Richard Hickey

Платформа: JVM (компиляция в Java byte-code)

Классификация: диалект Lisp промышленного назначения (или претендующий на таковое)

Особенности: чистая функциональная основа, эффективная многопоточность

Отношение к Java: не будем чинить то, что в Java не ломалось

# Инструментарий Clojure

**Язык и документация:**
http://clojure.org

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

**Рекомендуемая среда разработки:**
IntelliJ IDEA + Cursive plugin
https://cursive-ide.com/

Сборка:
**Leiningen + Apache Maven**
http://leiningen.org/

# Особенности записи арифметических операции

В Lisp нет арифметических (и других) операторов. Все операции являются вызовами функций (или спец. форм) и записываются в префиксной форме.
Отсутствует понятие приоритета операций, т.к. оно имеет смысл только для инфиксной записи.

C:

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

Lisp:

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

# Определение функции

```
(defn fname doc? [args] prepostmap? forms)
```

fname – имя функции

args – список аргументов

forms – тело, одно или несколько S-выражений

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

(defn fact [n]
  (if (> n 0)
    (* n (fact (dec n)))
    1))
```

# «Перегрузка» функций

```
;замечание: библиотечная функция + дает тот же эффект
(defn sum
  ([] 0)
  ([x] x)
  ([x y & rest]
    (+ x (apply sum (cons y rest)))))

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

# Числа

Представление целых и вещественных чисел – по аналогии с Java

Рациональные числа:

```
(println (/ 1.0 2.0))
>>0.5
(println (/ 1 2))
>>1/2
(println (+ 1/2 1/3))
>>5/6
```

1/2 – это форма записи числа, а не применение операции деления!

# Строки, регулярные выражения

```
“Some string” ;строка

#“[\w\s]+”    ;regex
```

Используется преимущественно Java-API

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

(println (.toUpperCase "Some string"))
>>SOME STRING
```

# Определение лексического контекста: let

```
;let определяет лексический
;и привязывает значения к символам

(let [a 10,          ;определение символов
      b (+ a 10)]
  (println a)        ;формы
  (println b))
```

# Списки: конструирование

**(f a b)** – определение списка и его вычисление

**(list a, b, c)** – определение списка из значений a, b, c *(запятая – обычный разделитель, как и пробел)*

**`(a b c)** – определение списка и блокирование его вычисления (прямой апостроф). Будет определен список из символов (не значений!) a, b, c

```
(cons `a `(b c)) -> (a b c) ;construct
(conj `(b c) `a) ->  --//-- ;conjoin

(concat `(a b) `(c d)) -> (a b c d)
;concatenate
```

# Доступ к списку

```
(first '(a b c)) -> a

(rest '(a b c))  -> (b c)

(nth 2 '(a b c)) -> c

(take 2 '(a b c)) -> (a b)
```

# Ветвления

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

;множественный if-else
(println
  (cond
    (> a 0) "pos"
    (< a 0) "neg"
    true    "nil"))

;;Замечания:
;;- if, cond возвращают значения
;;- false, nil – ложь, остальное - истина
```

# Функции с переменным числом параметров, анонимные функции

```
;определение анонимной функции и ее применение
((fn [x] (+ x 5)) 3)

;эквивалентная сокращенная запись
(#(+ % 5) 3)
```

```
;функция с переменным числом параметров
(defn poly [x & coeffs]
  (reduce
    ;анонимная функция
    (fn [acc coef]
      (+
        (* acc x)
        coef))
    0
    coeffs))

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

# Определение лексического контекста: другие формы

```
;defn, fn
;letfn

(letfn [(my-f [x] (+ 5 x))]
  (my-f 6))
;почти эквивалентно предыдущему,
;но имя функции my_f не будет доступно
;для рекурсивного вызова
(let [my-f (fn [x] (+ 5 x))]
  (my-f 6))

;loop - эквивалентно let, но при этом
;определяется точка возврата для recur
```

# Clojure и λ-исчисление

| | λ | Clojure |
|---|---|---|
| **Простая функция** | s=λx.λy.x+y | (defn s [x y] (+ x y))<br>(def s (fn [x y] (+ x y)))<br>(def s-c (fn [x] (fn [y] (+ x y)))) |
| **Аппликация** | s 2 3 | (s 2 3)<br>((s-c 2) 3) |
| **Каррирование** | curr=λf.λv.f v | (defn curr-c [f v] (f v))<br>(defn curr [f v]<br>  (fn [& args] (apply f v args))) |
| **Факториал** | fact 0 = 1<br>fact=λn.n*fact (n-1) | (defn fact [n]<br>  (if (= n 0) 1<br>  (* n (fact (- n 1))))) |
| **Ω-терм** | (λx.x x)(λx.x x) | ( (fn [x] (x x)) (fn [x] (x x)) ) |

**Упражнение.** Вычислить значение Ω-терма

# Вычисления над списками

**Императивное программирование:**

- цикл с итерацией по списку (for-each)
- модификация списка по мере итерации
- накопление результатов в отдельной переменной

**Функциональное программирование (классическое):**

- рекурсия с разделение «головы» и «хвоста» списка
- в рекурсивный вызов передается «хвост» и, при необходимости, результаты промежуточных вычислений

**Функциональное программирование (практическое):**

- Трансформирующие функции высшего порядка для списков

# Рекурсивная обработка

```
;;поэлементный инкремент
(defn inc-coll-1 [coll]
  (if (> (count coll) 0)
    (cons (inc (first coll))
          (inc-coll-1 (rest coll)))
    (list)))


;;факториал
(defn fact-1 [n]
  (if (> n 0)
    (* n (fact-1 (dec n)))
    1))

;;проблема: что если нужно перебрать
;;10^8 элементов?
```

# Задача 1.1

Задан набор (алфавит) символов в виде списка и число **n**. Опишите функцию, которая возвращает список всех строк длины n, состоящих из этих символов и не содержащих двух одинаковых символов, идущих подряд.

Для решения использовать рекурсию и базовые операции над константами и списками (**str**, **cons**, .**concat** и т.п.)

**Пример:** для алфавита ("a" "b " "c") и n=2 результат должен быть ("ab" "ac" "ba" "bc" "ca" "cb") с точностью до перестановки.

# Элементарное взаимодействие с Java: строки

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

;;вызов метода, эквивалентно "abc".toUpperCase() в Java
(.toUpperCase "abc")
(. "abc" toUpperCase)
;;-> "ABC"

(.concat "ab" "cd")
;;-> "abcd"

(.substring "abcd" 1 3)
;;-> "bc"
```

# Элементарное взаимодействие с Java

```
(ns ru.nsu.fit.dt
  ;clojure namespace
  (:use clojure.test)
  ;Java classes
  (:import (java.io File PrintWriter)))

;вызов Clojure с полной квалификацией имени
(clojure.xml/parse "file.xml")
;вызов статического метода/обращение к статическому
;атрибуту Java-класса
(Math/sin Math/PI)

;присваивание (только для Java-атрибутов)
(set! MyClass/someStaticVar 42)
```

# Элементарное взаимодействие с Java (продолжение)

```
;инстанцирование
(new HashMap)

;цепочка вызовов
(.. "str" (getClass) (getName)) ; java.lang.String
(.getName (.getClass "str" ))   ;эквивалентная запись

;последовательные манипуляции объектом
(doto (new HashMap) (.put "a" 1) (.put "b" 2)
(let [m (new HashMap)]
  (.put "a" 1) (.put "b" 2))
```

# Хвостовая рекурсия

**Идея:** если рекурсивный вызов идет последним, то текущее состояние стека можно «забыть» перед этим вызовом.

**«Последний»:** функция должна возвращать результат рекурсивного вызова без преобразований

```
;;ошибка компиляции
(defn fact-2 [n]
  (if (> n 0)
    (* n (recur (dec n)))
    1))
```

# Правильный хвостовой вызов

```
(defn fact-2
  ([n] (fact-2 n 1))
  ([n acc] (if (> n 0)
             (recur (dec n) (* n acc))
             acc)))


(defn inc-coll-2
  ([coll] (inc-coll-2 coll (list)))
  ([coll res-coll]
    (if (> (count coll) 0)
      (recur (rest coll)
             (concat
               res-coll
               (list (inc (first coll)))))
      res-coll)))
```

# Задача 1.2

Изменить решение задачи 1.1 таким образом, чтобы все рекурсивные вызовы были хвостовыми

# Обобщение inc-coll: map

```
(defn my-map [f coll]
  (loop [input-coll coll, res-coll (list)]
    (if (> (count input-coll) 0)
      (recur
        (rest input-coll)
        (concat
           res-coll
           (list (f (first input-coll)))))
      res-coll)))


(my-map inc (list 1 2 3 4))
;;-> (2 3 4 5)
```

# Обработка списков: map

**(map f coll)**
**(map f coll & colls)** – порождение нового списка путем поэлементного преобразования **coll** посредством применения **f**

```
(map (fn [x] (* x x))
     (list 1 2 3 4))
;;-> ???

(map +
     (range 0 5)
     (range 5 10)
     (range 10 15))
;;->  (15 18 21 24 27)
```

# Обработка списков: reduce

**(reduce f coll)**
**(reduce f val coll)** - свертка списка заданной двуместной функцией

```
(defn fact-3 [n]
  (if (> n 0)
    (reduce *
            1
            (range 1 (inc n)))
    1))


(reduce +
        0
        (map (fn [x] (* x x x))
             (range 1 4)))
;;-> ???
```

# Обработка списков: filter

**`(filter pred coll)`** - фильтрация списка по заданному предикату

```
(filter (fn [n] (= 0 (mod n 3)))
        (range 1 21))
;;-> ???
```

# Обработка списков: основные функции

**`(map f coll)`** – отображение
**`(reduce f val coll)`** – свертка
**`(filter pred coll)`** – фильтрация
**`(remove pred coll)`** – дополнение **filter**

**`(some pred coll)`** – возвращает первое истинное значение **(pred x)**, где x из **coll** и **nil**, если нет такого
**`(every? pred coll)`** – тест, все ли элементы **coll** удовлетворяют предикату

# Задача 1.3

Определить функции **my-map** и **my-filter**, аналогичные **map** (для одного списка) и **filter**, выразив их через **reduce** и базовые операции над списками (cons, first, concat и т.п.)

# Задача 1.4

Изменить решение задачи 1.1/1.2 таким образом вместо рекурсивных вызовов использовались map/reduce/filter.

# Модификация структур данных

Модификация всех структур данных, а также переменных в Clojure **запрещена** (кроме особых случаев)

*Примечание: использование **java.lang.String** не нарушает это правило*

Все мутаторы создают новую структуру данных вместо модификации существующей

Проблема 1: потенциальная утечка памяти в цепочках вычислений

Решение: Garbage Collector, обязательный для всех функциональных языков

Проблема 2: перерасход памяти за счет дублирования данных

Решение: Persistent Data Structures

# Persistent Data Structures

*Persistent – постоянный, неизменный*

PDS – структура данных, которая при изменении сохраняет свое предыдущее состояние.

Принцип организации:

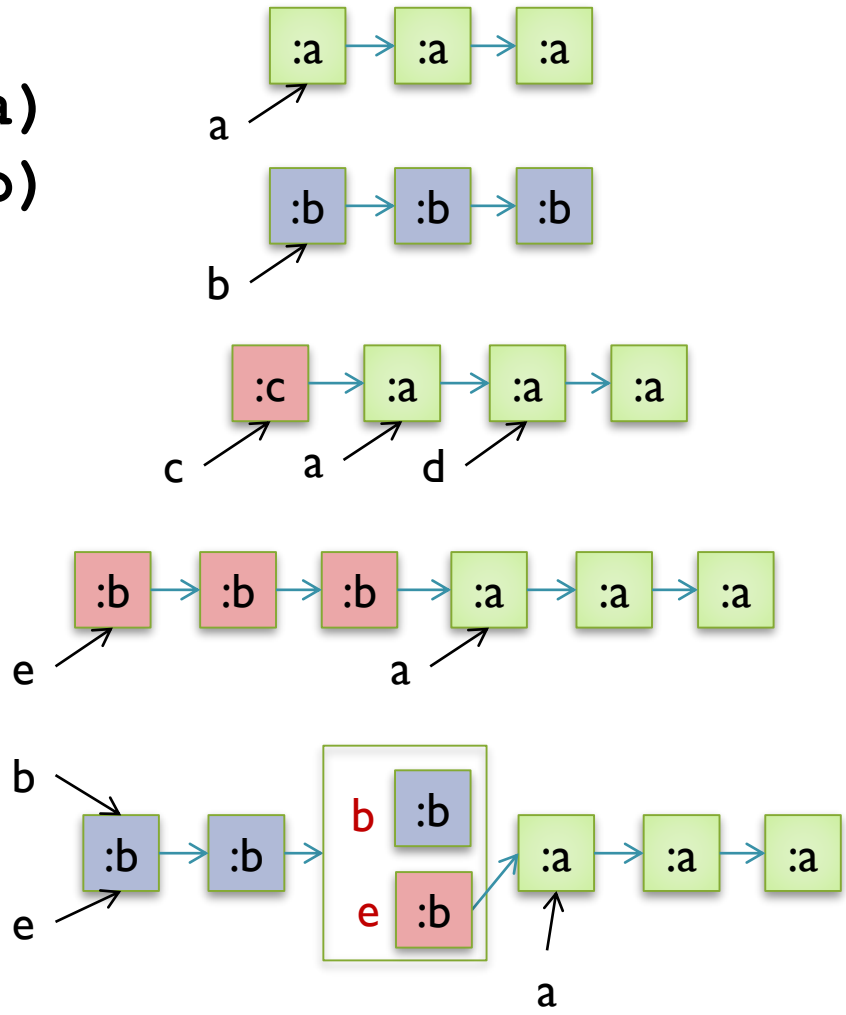
1. Для двух разных состояний общая часть хранится в памяти в единственном экземпляре. Для каждого состояния хранится отдельно уникальная часть.
2. Принцип обобщается на произвольное количество состояний

# Пример: Persistent List

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

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


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



# Генератор for

```
(for [x ["a" "b" "c"]
      y ["a" "b" "c"]
      :when (not (= x y))]
  (.concat x y))

;;-> ("ab" "ac" "ba" "bc" "ca" "cb")
```

# Массивы (vector)

В отличие от списка массив не интерпретируется как применение функции (специальной формы)

**[1 2 3]** – объявление массива
**(vector 1 2 3)** ---//--

**(vec '(1 2 3))** – порождение массива из списка

```
(assoc [1 2 3] 1 4) -> [1 4 3]
;associate
(conj [1 2 3] 4)    -> [1 2 3 4]
;добавляет элемент в конец
(cons 4 [1 2 3])    -> (4 1 2 3)
;добавляет элемент в начало, возвращает список
```

# Ассоциативные массивы

```
{:a 1, :b 2}
```
**(hash-map :a 1, :b 2)** – объявление хэш-таблицы

```
(sorted-map &keyvals)
(sorted-map-by comparator &keyvals) -
```
объявление сортированного ассоциативного массива

```
(assoc {:a 1} :b 2) -> {:a 1 :b 2}
(dissoc {:a 1 :b 2} :b) -> {:a 1}
({:a 1 :b 2} :a) -> 1
(:a {:a 1 :b 2}) -> 1

(map (fn [x] x)
     {:a 1 :b 2}) -> ([:a 1] [:b 2])
```

# Множества

```
#{:a :b :c}
(hash-set :a :b :c)
(sorted-set & entries)
(sorted-set-by comparator & entries)


(conj #{:a :b} :c) -> #{:a :b :c}
(disj #{:a :b :c} :c) -> #{:a :b}
(contains? #{:a :b} :b) -> true

(ns test
   (:use clojure.set))
(intersection #{:a :b} #{:b :c}) -> #{:b}
;...
```

# Последовательное исполнение

```
;do объединяет несколько форм в одну
;и исполняет их, возвращая результат
;последней формы

(if (> a 0)
    (do
      (println "pos")
      true)
    (do
      (println "neg")
      false))
```

# Спец. форма: цепочка операция

```
(-> {}
  (assoc :a 1)
  (assoc :b 2)
  (assoc :c 3))

;эквивалентная запись
(assoc
  (assoc
    (assoc {}
      :a 1)
    :b 2)
  :c 3)

;->> - то же самое что и ->, но объект
;подставляется на последние позиции форм
```

# Destructuring (реорганизующее присваивание)

```
(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)

(defn my-f [[x y & rest] val] …)

(my-f [1 2 3 4] 5)
```

# Map-Based Destructuring

```
(let [m {:a 1, :b 2, :c 3}
      {a :a, b :b} m] ;a=1, b=2

 …)
```

# Задача 2.1

Напишите функцию, которая ищет **n**-ое простое число с помощью решета Эратосфена

# Модульное(компонентное) тестирование (unit-testing)

Концепция модульного тестирования подразумевает автоматическую проверку функциональности каждой минимальной программной единицы, для которой это возможно.

В данном контексте модуль, это:

- функция
- группа связанных функций
- метод
- интерфейс
- класс

Каждый модуль покрывается набором тестов, покрывающих всю функциональность, в т.ч. Проверку ошибок.

# Модульное тестирование: тест

Тест – это функция, возвращающая true, если и только если тест прошел успешно.

Тесты разрабатываются параллельно с основным кодом, иногда даже раньше самого кода.

Все тесты запускаются при каждой сборке, среда тестирования выдает сводный отчет по всем тестам.

Тесты запускаются последовательно. По возможности правило обеспечивается изоляция тестов, т.е. если один «упал», остальные продолжат выполнение.

# Модульное тестирование, пример: основной код

```
(ns ru.nsu.fit.dt.defpackage
  (:gen-class))

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

# Модульное тестирование, пример: тестирующий код

```
(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)
```

# Мемоизация

Мемоизация – оптимизация производительности путем кэширования возвращаемых значений функций.

Применима только для функций, не имеющих побочных эффектов (чистых функций).

Имеет смысл для «тяжелых» функций, когда время вычислений функции заведомо превышает время поиска по хэш-таблице.

Расходует память пропорционально количеству комбинаций параметров.

# Мемоизация в действии

```
(def fact-mem (memoize fact))

(defn -main [& args]
  (time (fact 100))
  (time (fact 100))
  (time (fact 100))
  (time (fact-mem 100))
  (time (fact-mem 100)))
```

```
>>
"Elapsed time: 2.199713 msecs"
"Elapsed time: 0.996379 msecs"
"Elapsed time: 0.979387 msecs"
"Elapsed time: 1.087113 msecs"
"Elapsed time: 0.032623 msecs"
```

# Задача 3.1: численное интегрирование

Реализовать функцию (оператор), принимающую аргументом функцию от одной переменной ***f*** и возвращающую функцию одной переменной, вычисляющую (численно) выражение

$$\int_0^x f(t)dt$$

Оптимизировать с использованием мемоизации для задач типа построения графиков (т.е. многократный вызов функции в разных точках)

Использовать метод трапеций с постоянным шагом.

Показать прирост производительности с помощью **time**.

Обеспечить покрытие тестами.