PL Party

Обзор функциональных языков, их особенностей
и подходов к решению задач

Paradigms → Features

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

— *Shriram Krishnamurthi*

Чем интересно ФП

- Красота
- Простота
- Стабильность
- Прагматика
- Генератор идей: от проектов на лисп-машинах до React
- Образование: топовые учебные материалы

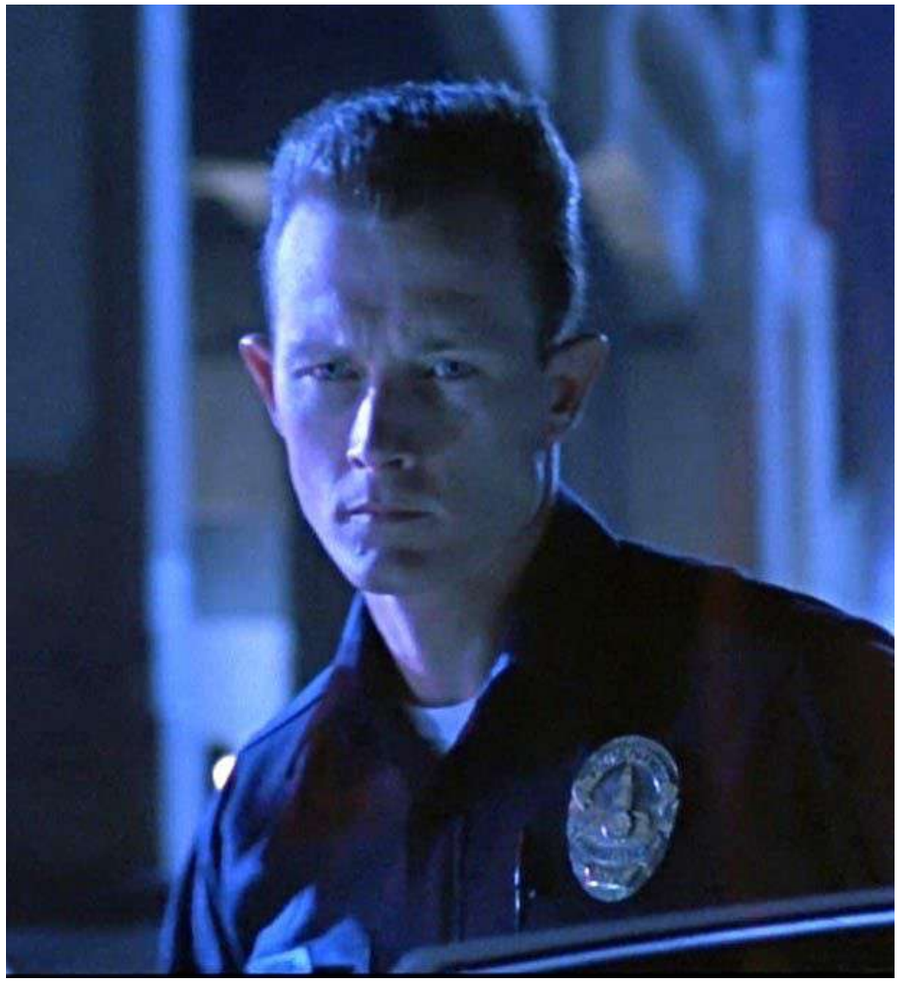
Как выглядит знакомство с функциональным программированием через JavaScript/Python/Java/…:

— Вот все говорят: «Карузо! Карузо!» А я послушал — так ничего особенного.

— Вы слышали Карузо?!

— Нет. Мне Изя напел.

Lisp



- Scheme
- Common Lisp
- Racket
- Clojure

MetaLanguage

- StandardML
- OCaml
- Elm
- Haskell
- ReasonML

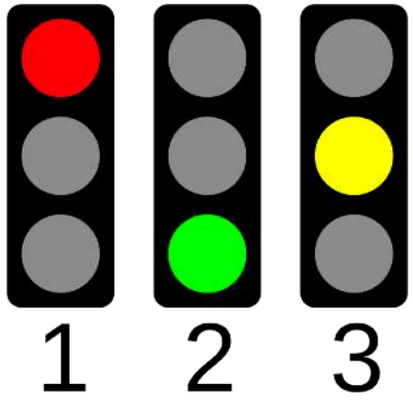
Императивный, ООП

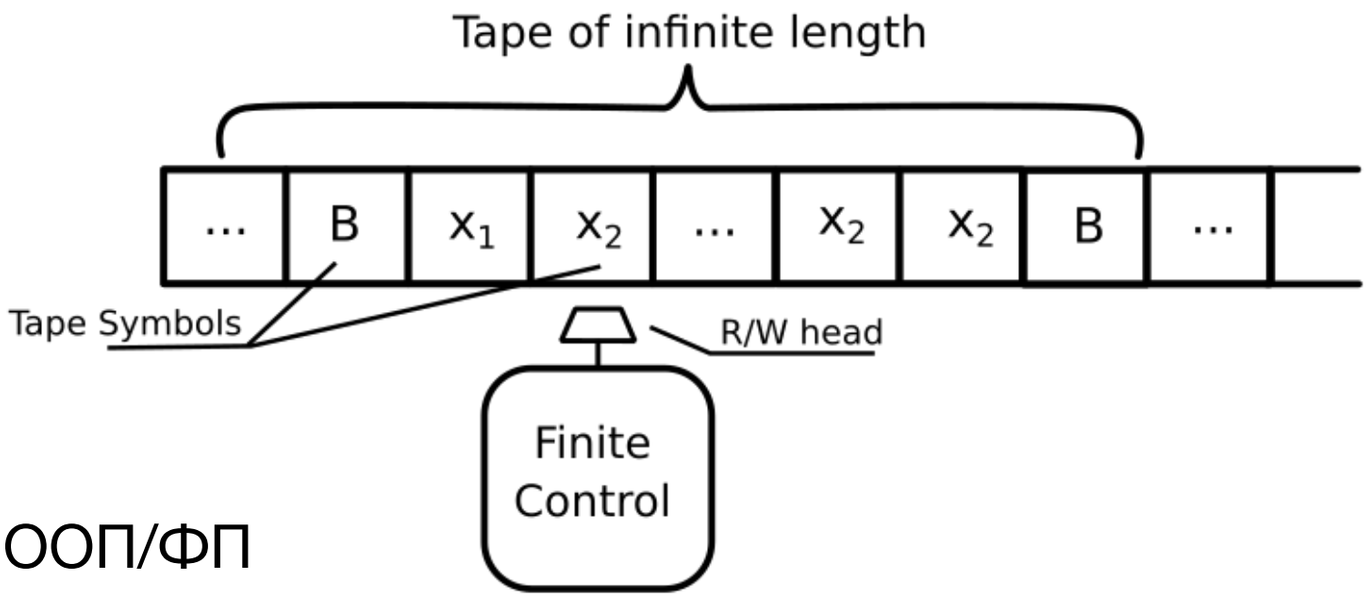
- Состояние
- Инкапсуляция

Функциональный

- Данные
- Функции

> *It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.*
>
> *— Alan J. Perlis*





Примеры на задачах

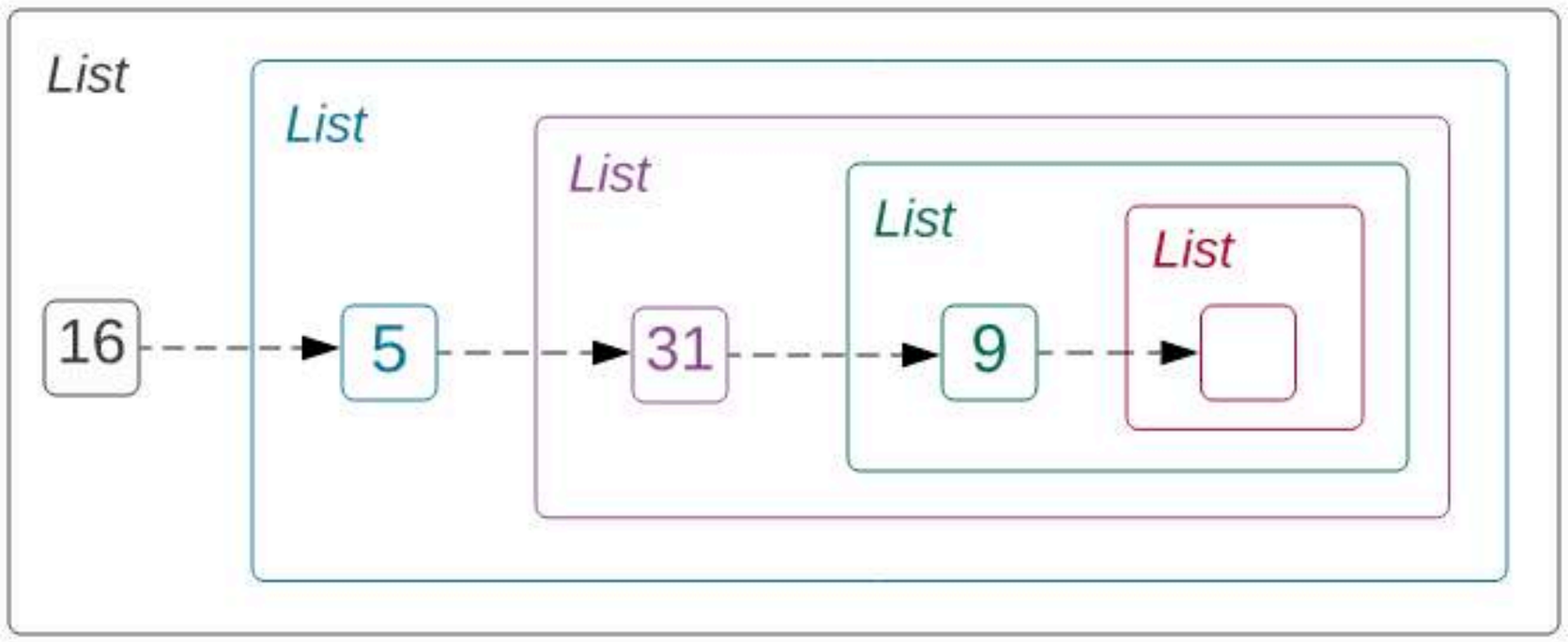
- Моделирование светофора в ООП/ФП
- Машина Тьюринга в ООП/ФП

Определения

- Информация
- Данные
- Функция
- Полиморфизм
- Инкапсуляция
- Объект
- Состояние
- Мутация

Данные

[Linked] List



На примере списков рассмотрим работу с данными

Список в Lisp

```
; List
'(1 2 3)

; Cons-ing a
(cons 1 '()) ; (1)
(cons 1 (cons 2 '())) ; (1 2)
(cons 1 (cons 2 (cons 3 '()))) ; (1 2 3)
```

- Recursive data type
- Constructing

Список может быть:

- пустым
- или начинаться с элемента (head) и заканчиваться другим списком, в том числе пустым (tail).

Список в ML

```elm
type List
    = []
    | first :: List rest


lst : List Int
lst =
    16 :: (5 :: (31 :: (9 :: [])))
```

```elm
check : List Int -> String
check l =
    case l of
        [] ->
            "Empty"

        x :: xs ->
            "First is "
                ++ String.fromInt x
                ++ " rest is "
                ++ Debug.toString xs
```

Список может быть или пустым или начинаться с элемента (head) и заканчиваться списком, в том числе пустым (рекурсивное определение).

- Type inference
- Type annotation
- Pattern matching
- Polymorphism

Данные

Lisp

ML

Open/closed world assumption

Clojure Data Structures

- Immutable
- Persistent
- Structural sharing

Datomic

```
'(1 2 3)    ; List
[1 2 3]     ; Vector
{:a 3 :b 4} ; Map + keywords
#{:a :b :c} ; Set + keywords
```

Функции

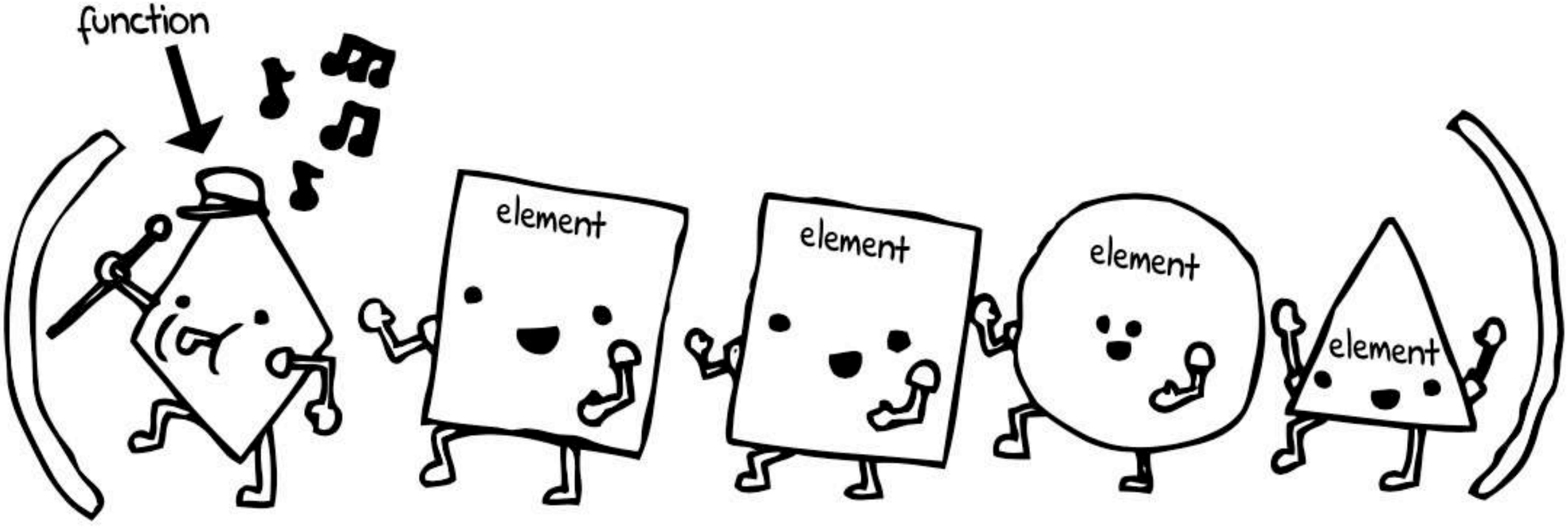
анонимная функция → крышка → лямбда

$$\lambda x.x^2 + 5x + 7\ 4$$

анонимная функция → крышка → лямбда

$$\lambda x.x^2 + 5x + 7\ 4$$

```
(lambda x: x^2 + 5*x + 7)(4)
```




```clojure
(+ 1 2)
(prn "Hello world")
(some even? [1 2 3])

(defn a<b->1?0 [a b]
  (if (< a b) 1 0))
```

Partial application

```
sum a b c =
    a + b + c


sum3bc =
    sum 3


sum34c =
    sum 3 4


res =
    sum 3 4 5
```

Частичное применение работает в ML всегда, потому что все функции — каррированные по-умолчанию и нет сайд-эффектов. Все функции — чистые.

Функции от нескольких аргументов — череда функций от одного аргумента.

Применение аргументов — последовательность частичных применений.

Partial application

```
sum a b c =
    a + b + c


sum3bc =
    sum 3


sum34c =
    sum 3 4


res =
    sum 3 4 5
```

```
> sum 3
<function> : number -> number -> number
> sum 3 4
<function> : number -> number
> sum 3 4 5
12 : number
```

Partial application and Piping

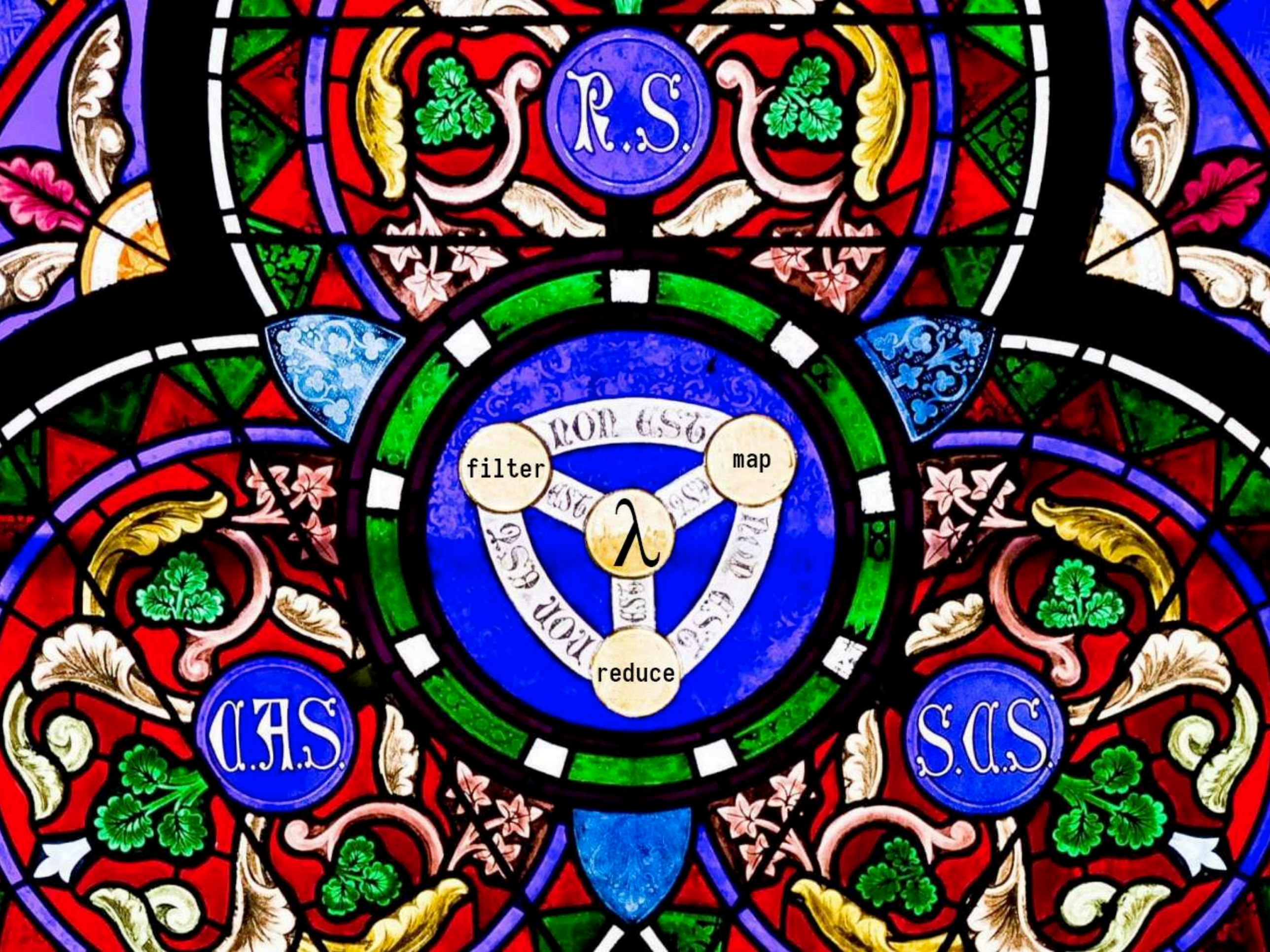
```
List.range 1 lim
    |> filter isEven
    |> map square
    |> fold (+) 0
```

Partial application and Piping

```
fold (+)
    0
    (map (\n -> n * n)
        (filter (\n -> modBy 2 n == 0)
            (List.range 1 10)
        )
    )
```

→

```
List.range 1 lim
    |> filter isEven
    |> map square
    |> fold (+) 0
```


filter
map
λ
reduce

map

```
def my_map(fn, lst):
  result = []
  for el in lst:
    result.append(fn(el))
  return result

my_map(lambda x: x + 10, [1, 2, 3])  # [11, 12, 13]
```

map

```python
def my_map(fn, lst):
  result = []
  for el in lst:
    result.append(fn(el))
  return result

my_map(lambda x: x + 10, [1, 2, 3])  # [11, 12, 13]
```

```haskell
map fn list =
    case list of
        [] ->
            []

        x :: xs ->
            fn x :: map fn xs
```

map

```python
def my_map(fn, lst):
    result = []
    for el in lst:
        result.append(fn(el))
    return result

my_map(lambda x: x + 10, [1, 2, 3])  # [11, 12, 13]
```

```haskell
map : (a -> b) -> List a -> List b
map fn list =
    case list of
        [] ->
            []

        x :: xs ->
            fn x :: map fn xs
```

+ аннотация

reduce

```
def my_reduce(reduce_fn, initial, lst):
    reduced = initial
    for x in lst:
        reduced = reduce_fn(reduced, x)
    return reduced

add = lambda x, y: x + y
my_reduce(add, 0, [1, 2, 3]) # 6
```

reduce

```python
def my_reduce(reduce_fn, initial, lst):
    reduced = initial
    for x in lst:
        reduced = reduce_fn(reduced, x)
    return reduced

add = lambda x, y: x + y
my_reduce(add, 0, [1, 2, 3]) # 6
```

```haskell
fold func acc list =
    case list of
        [] ->
            acc

        x :: xs ->
            fold func (func x acc) xs
```

```
> fold (+) 0 [1, 2, 3]
6 : number
```

reduce

```python
def my_reduce(reduce_fn, initial, lst):
    reduced = initial
    for x in lst:
        reduced = reduce_fn(reduced, x)
    return reduced

add = lambda x, y: x + y
my_reduce(add, 0, [1, 2, 3]) # 6
```

```haskell
fold : (a → b → b) → b → List a → b
fold func acc list =
    case list of
        [] →
            acc

        x :: xs →
            fold func (func x acc) xs
```

- Итеративный процесс
- Аккумулятор
- Рекурсивный вызов
- Паттерн-матчинг

filter

```python
def my_filter(fn, lst):
  result = []
  for el in lst:
    if fn(el):
        result.append(el)
  return result

is_even = lambda x: x % 2 == 0
my_filter(is_even, [1, 2, 3, 4]) # [2, 4]
```

filter

```python
def my_filter(fn, lst):
  result = []
  for el in lst:
    if fn(el):
        result.append(el)
  return result

is_even = lambda x: x % 2 == 0
my_filter(is_even, [1, 2, 3, 4]) # [2, 4]
```

```haskell
filter pred list =
    case list of
        [] ->
            []

        x :: xs ->
            if pred x then
                x :: filter pred xs

            else
                filter pred xs
```

filter

```elm
filter : (a -> Bool) -> List a -> List a
filter pred list =
    case list of
        [] ->
            []

        x :: xs ->
            if pred x then
                x :: filter pred xs

            else
                filter pred xs
```

Найти сумму квадратов чётных чисел в диапазоне от `0` до `lim`.

Найти сумму квадратов чётных чисел в диапазоне от `0` до `lim`.

```python
def sum_of_even_squares(lim):
    "Returns sum of squares of even numbers in [0, lim] range."
    result = 0
    for i in range(0, lim + 1):
        if i % 2 == 0:
            result += i * i
    return result
```

Найти сумму квадратов чётных чисел в диапазоне от `0` до `lim`.

```python
def sum_of_even_squares(lim):
    "Returns sum of squares of even numbers in [0, lim] range."
    result = 0
    for i in range(0, lim + 1):
        if i % 2 == 0:
            result += i * i
    return result
```

```python
def my_sum_of_even_squares(lim):
    "Returns sum of squares of even numbers in [0, lim] range."
    return my_reduce(add, 0, my_map(square, my_filter(is_even, range(0, lim + 1))))
```

Найти сумму квадратов чётных чисел в диапазоне от `0` до `lim`.

```
sumOfSquares lim =
    fold (+) 0 (map (\n -> n * n) (filter (\n -> modBy 2 n == 0) (List.range 1 lim)))
```

```
(defn sum-of-squares [lim]
  (reduce + (map #(* % %) (filter even? (range 0 (inc lim))))))
```

Найти сумму квадратов чётных чисел в диапазоне от `0` до `lim`.

```elm
sumOfSquares : Int -> Int
sumOfSquares lim =
    let
        isEven =
            \n -> modBy 2 n == 0

        square =
            \n -> n * n
    in
    List.range 1 lim
        |> filter isEven
        |> map square
        |> fold (+) 0
```

```clojure
(defn sum-of-squares [lim]
  (->> (range 0 (inc lim))
       (map #(* % %))
       (filter even?)
       (reduce +)))
```

Lazy evaluation

Итераторы и потоки

Итератор

filter, map, reduce в Python возвращают итераторы

```
s = [1, 2, 3, 4, 5] # sequence
i = iter(s)         # a lazy sequence

next(i) # 1
next(i) # 2
next(i) # 3
next(i) # 4
next(i) # 5
next(i) # Error
```

```
m = map(square, [1, 2, 3, 4, 5])
next(m) # 1
next(m) # 4
next(m) # 9

list(m) # [16, 25]
```

next изменяет состояние

В Clojure всё всегда одинаково

```
(def s (lazy-seq [1 2 3 4 5]))
(take 3 s) ; (1 2 3)
(drop 2 (map #(* % %) s)) ; (9 16 25)
s ; (1 2 3 4 5)
```

Lexical Scope

Python

```python
a = 3

def f():
  return 10 + a

a = 33

print(f()) # 43
```

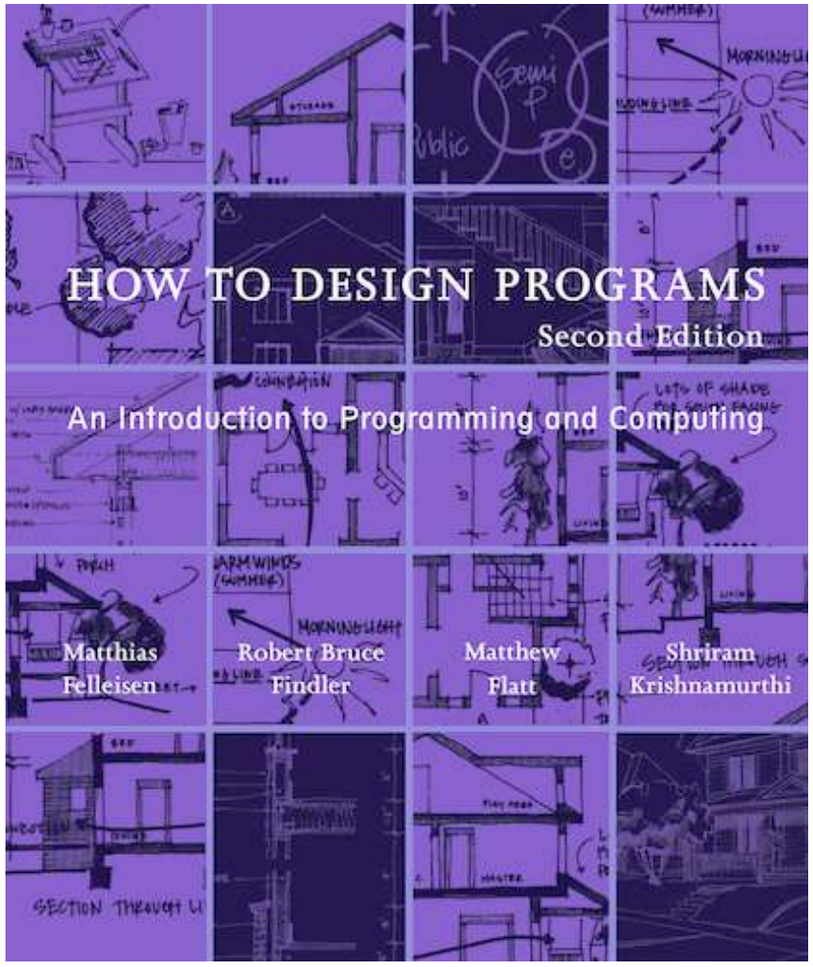
OCaml

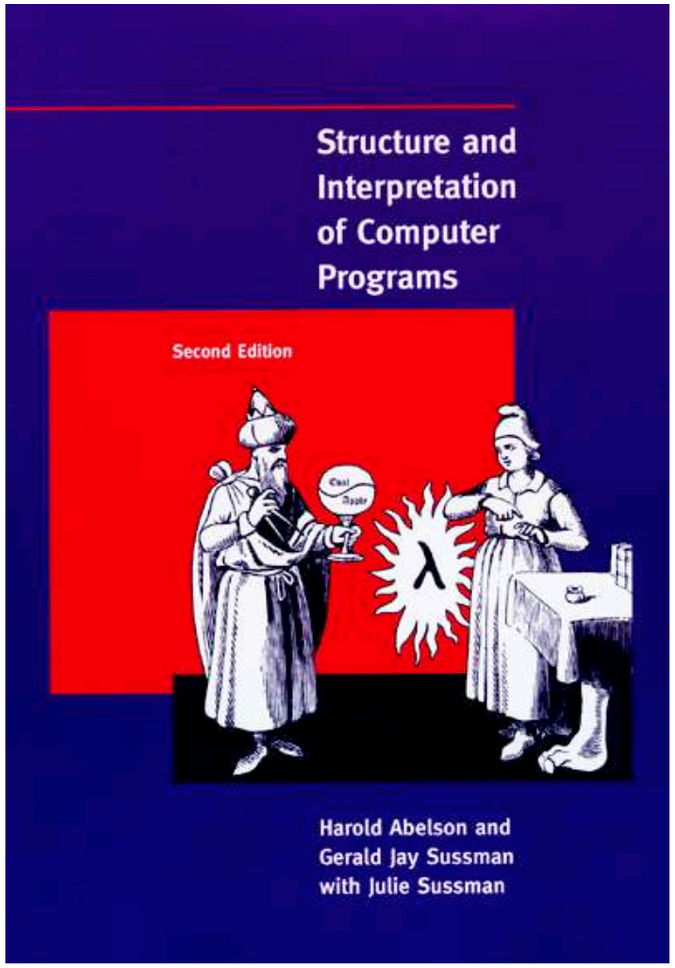
```ocaml
let a = 3;;

let f = fun () -> 10 + a;;

let a = 33;;

print_string (string_of_int (f ()));; (* 13 *)
```

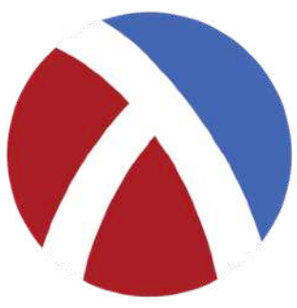
Где научиться?

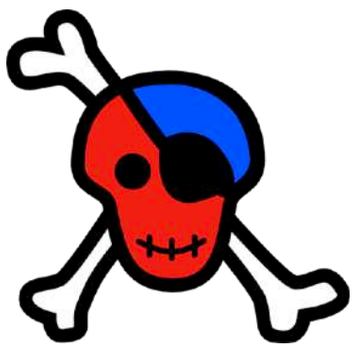
Основы

Программирование в целом, основы ФП и ООП. Racket, Scheme, Pyret.





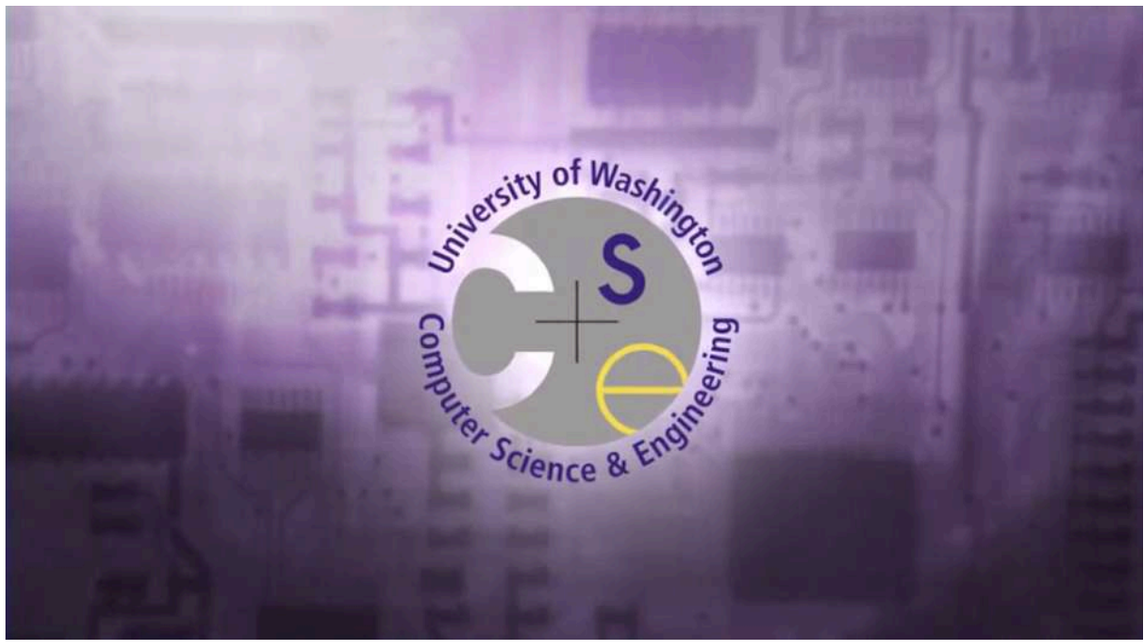




Подробности:

http://andreymiskov.ru/posts/intro-cs-bravit/

Разные языки в сравнении: ФП, ООП, статика, динамика



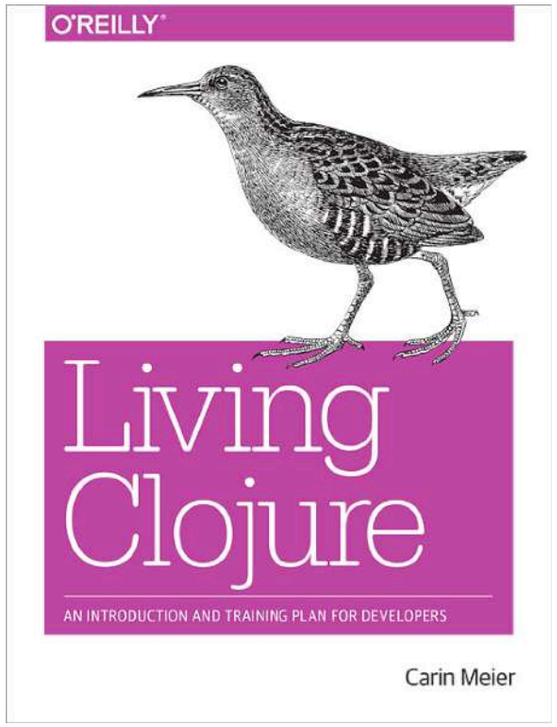
Если интересно сравнить, как разные языки решают разные задачи, подойдёт курс Programming Languages от Вашингтонского университета. В трёх частях рассматриваются StandardML (прародитель OCaml, Elm, Haskell и др.), Racket (lisp) и Ruby.

Part I: StandardML, static typing, type inference, functional programming:
https://www.coursera.org/learn/programming-languages

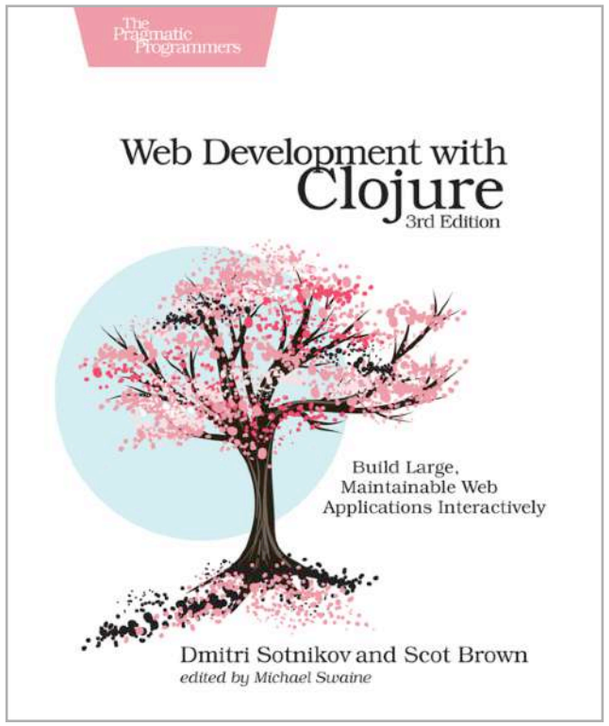
Part II: Lisp, dynamic typing, functional programming:
https://www.coursera.org/learn/programming-languages-part-b

Part III: Ruby, dynamic typing, OOP:
https://www.coursera.org/learn/programming-languages-part-c

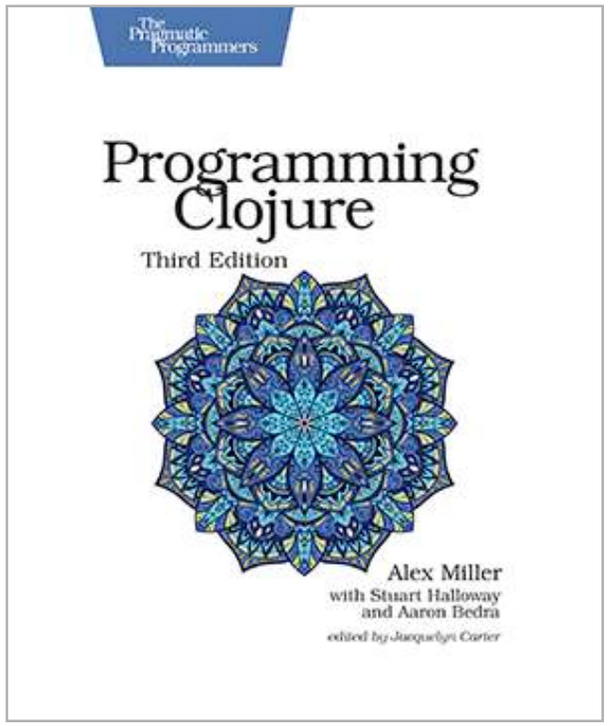
Clojure



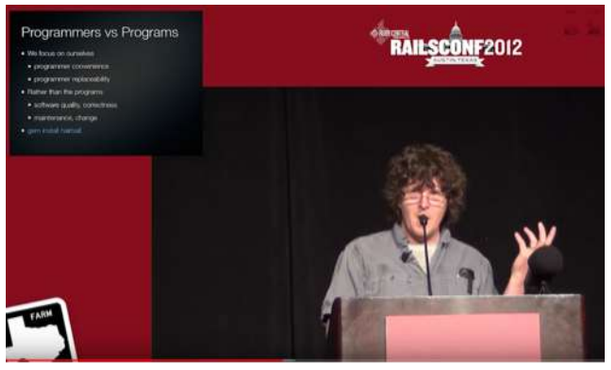
Введение в язык +
7-недельный план практики



Фуллстэк веб-разработка на
Clojure/Script. Бэкенд — Luminus,
Фронтенд — re-frame, re-agent.



Больше подробностей об
устройстве Clojure



Доклады Рича Хикки, создателя Clojure. Эти стоит посмотреть в первую очередь:

- Simple made easy: https://youtu.be/oytL881p-nQ
- Simplicity matters: https://youtu.be/rI8tNMsozo0

Elm

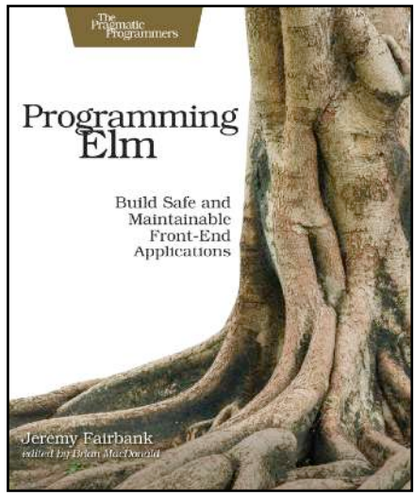
Основы языка просто и доступно.
https://elmprogramming.com/

↓

Практика: https://exercism.io/my/tracks/elm

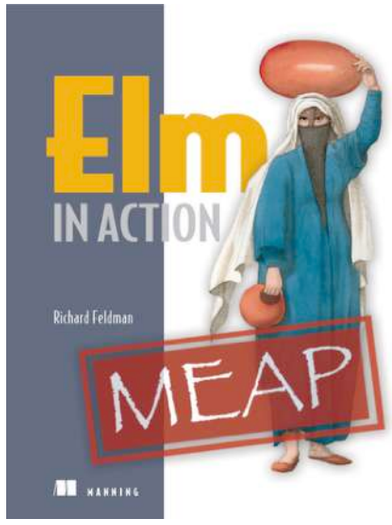
↓

Обзор возможностей языка от его создателя:
https://guide.elm-lang.org/



Основы разработки на Elm

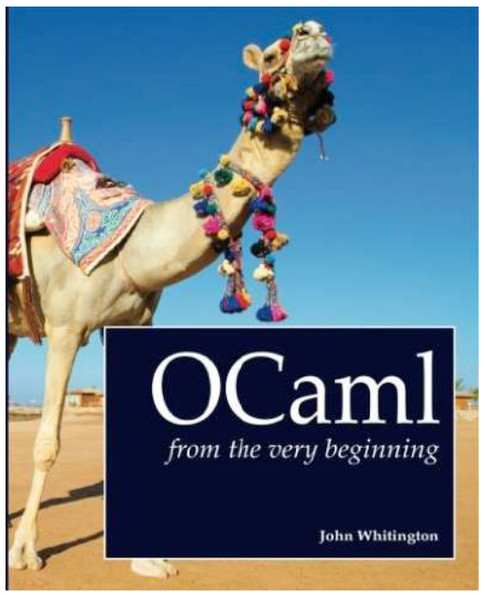
→



Разработка на Elm, автор из core team

Курсы на Frontend Masters: https://frontendmasters.com/teachers/richard-feldman/

OCaml



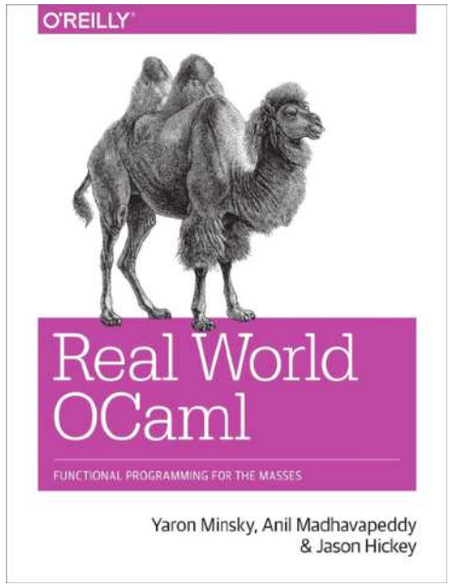
OCaml from the Very Beginning

Курс Introduction to Functional Programming in OCaml, о перезапусках можно узнать в Твиттере: https://twitter.com/ocamlmooc

CS 3110

Cornell University, курс CS3110: https://www.cs.cornell.edu/courses/cs3110/2019fa/

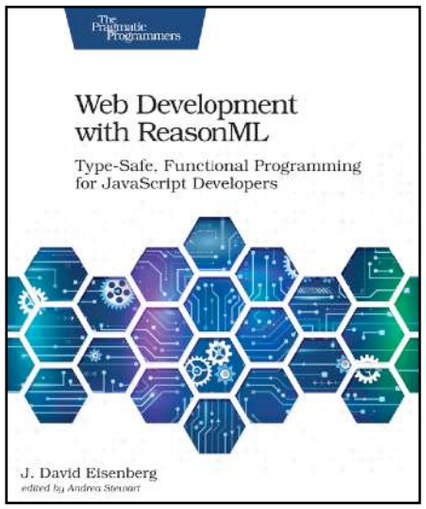
Книга Functional Programming in OCaml, прилагающаяся к курсу: https://www.cs.cornell.edu/courses/cs3110/2019fa/textbook/



Лекция Ярона Мински (автора книги) о том, чем хорош OCaml: https://youtu.be/DM2hEBwEWPc

Книга доступна бесплатно на https://dev.realworldocaml.org/

ReasonML

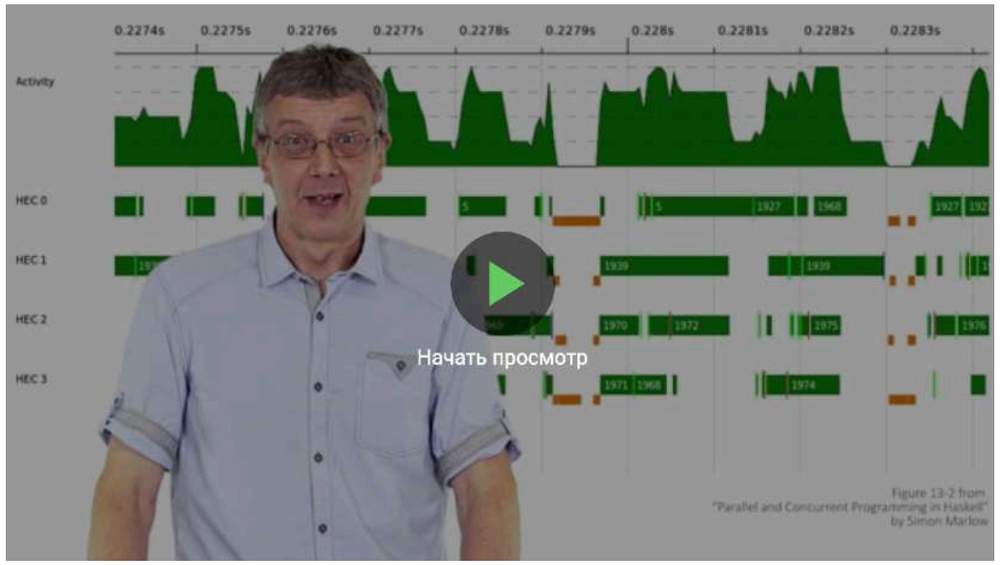


Основы языка и примеры разработки

REASON

Документация на официальном сайте
https://reasonml.github.io/

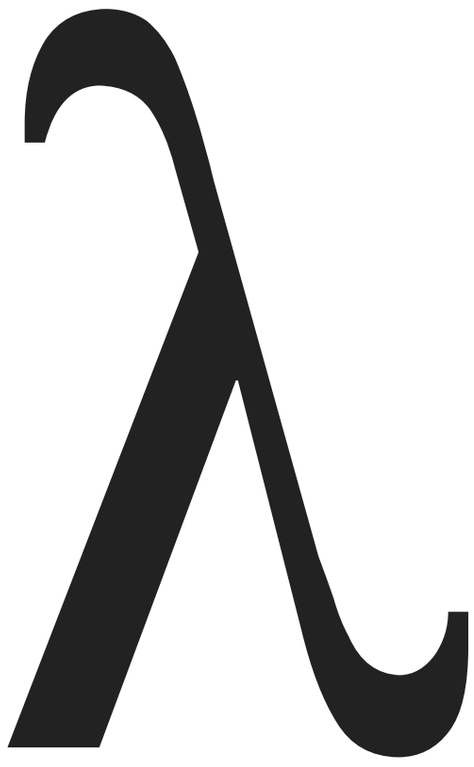
Haskell



Курс «Функциональное программирование на языке Haskell»

Виталий Брагилевский
@_bravit

Всем желающим начать изучать Haskell рекомендую эту книжку: dmkpress.com/catalog/comput..., там даже про разбор JSON есть!



http://andreymiskov.ru/