Clojure

ЛИСП

Функциональное программирование

Развитая платформа

Concurrency

Clojure
ЛИСП

Маленькое ядро

Почти нет синтаксиса

Код-как-данные

Clojure

ФП

First-class functions

Lexical closures

Ленивость

Dynamic typing

Clojure
JVM

Быстрая
Качественная
Уровенем выше ОС,
  лучше абстракции
Валом библиотек
Валом инструментов

Clojure
Concurrency

Иммутабельность
Персистентные структуры данных
Software transactional memory
Lock-free

Особенности

Многоплатформенность

Маленькое ядро
Мало синтаксиса
↓

+ JVM
+ .NET
+ JavaScript
+ Python
+ …?

Особенности

Interop

Уважает платформу

Писать джаву на кложе проще,
чем на джаве

clj strings = java strings

clj numbers = java numbers

clj nil = java null

Особенности

Перпендикулярность

Construct	Complects
State	Everything that touches it
Objects	State, identity, value
Methods	Function and state, namespaces
Syntax	Meaning, order
Inheritance	Types
Switch/matching	Multiple who/what pairs
var(iable)s	Value, time
Imperative loops, fold	what/how
Actors	what/who
ORM	OMG
Conditionals	Why, rest of program

Особенности

Перпендикулярность-2

State = value + time

OOP = functions + data

Особенности

Перпендикулярность-3

Construct	Get it via...
Values	final, persistent collections
Functions	a.k.a. stateless methods
Namespaces	language support
Data	Maps, arrays, sets, XML, JSON etc
Polymorphism a la carte	Protocols, type classes
Managed refs	Clojure/Haskell refs
Set functions	Libraries
Queues	Libraries
Declarative data manipulation	SQL/LINQ/Datalog
Rules	Libraries, Prolog
Consistency	Transactions, values

Особенности

Превозносит данные

Программы перемалывают данные

Классы это тюрьма для данных

Стандартные структуры

с богатой семантикой

Особенности

Превозносит данные

> “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

Особенности

Открытость

Мультиметоды, протоколы

Метаданные

Predicate dispatch вместо
  pattern matching

Composable abstractions

```
$().click().css() или
(-> ($ ) (click ) (css))
```

Особенности

Открытость-2

Управление всем

Vars, namespaces, загрузка кода

— всё программируется

Дебаггер с брекпоинтами

уместился в одну главу книги

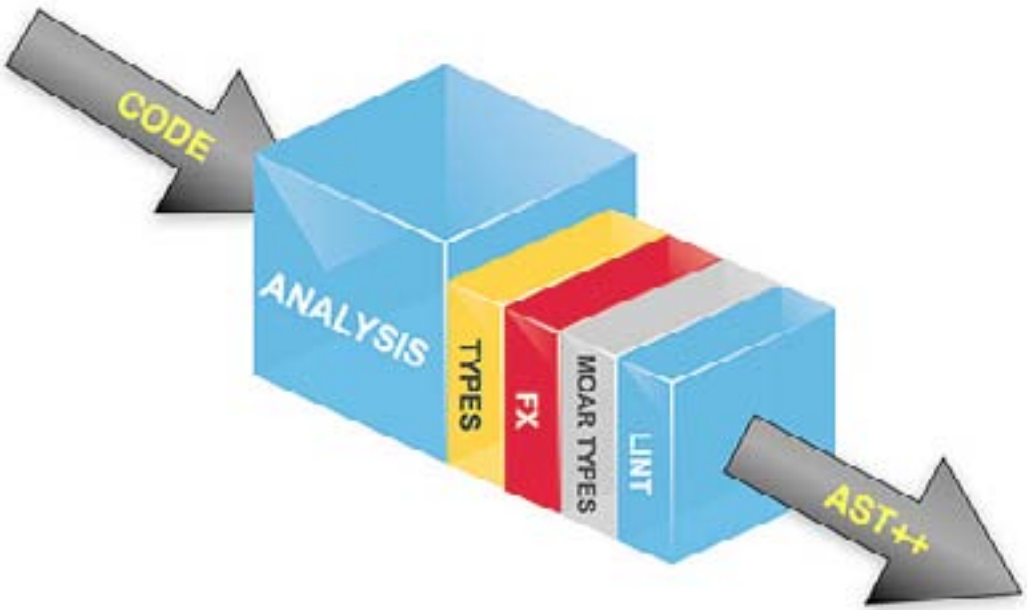
Особенности

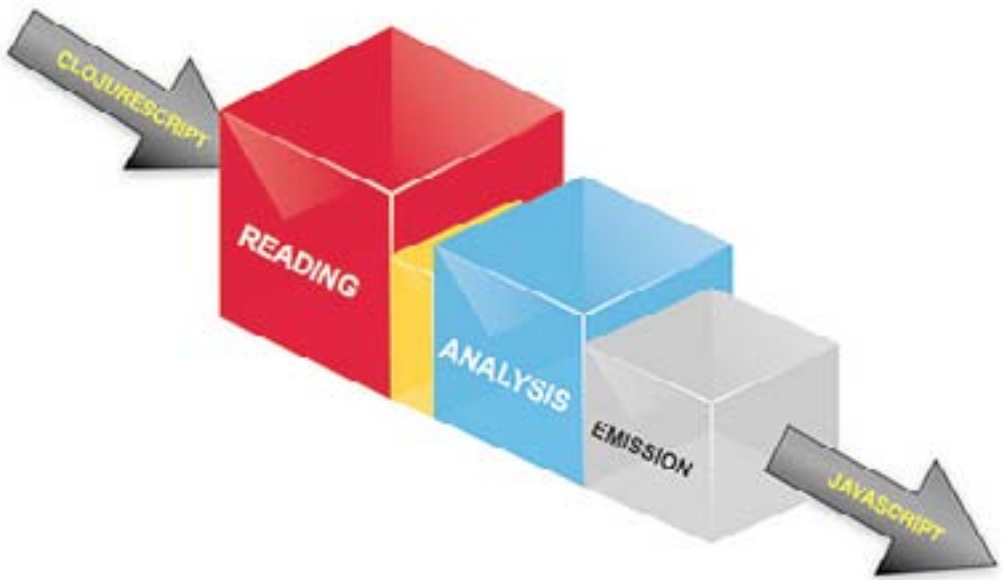
Открытость-3

Compilation pipeline

Compiler-as-a-service

Как будто собираешь свой маленький компьютер





Особенности

Макросы

Все то, о чем мы
так долго мечтали

Особенности

Переиспользуемость

Нет custom reader
есть reader literals though
Неймспейсится всё

Особенности

Динамичность

Новое для JVM

REPL

Компиляция в байткод на лету

Основы

Как это выглядит

```
“string”
:keyword
‘(\L \i \s \t)
 [:V :e :c :t :o :r]
#{“S” “e” “t”}
 {:key value, :key2 value2}
\C
#”[Rr]egexp?”
```

Основы

А в видеоверсии — сделал

```
#inst ”2012-06-26T11:00:00.00+04:00”
  java.util.Date

#uuid “uuid-string”
  java.util.UUID
```

Основы

Как это выглядит-2

```
(ns echo.clojure-u.lecture1
   (:use clojure.java.io
         [clojure.string :only [replace]]))

(def x 10)

(defn hello [arg & args]
   (println “Hello, “ arg))
```

Основы

Как это выглядит — всё вместе

```
(condp some [1 2 3 4]
  #{0 6 7} :>> inc
  #{4 5 9} :>> dec
  #{1 2 3} :>> #(+ % 3))
```

Основы

Что вообще происходит?

```
⟨some #{0 6 7} [1 2 3 4]⟩
 ↳⟨or ⟨#{0 6 7} 1⟩  ; nil
      ⟨#{0 6 7} 2⟩  ; nil
      ⟨#{0 6 7} 3⟩  ; nil
      ⟨#{0 6 7} 4⟩⟩ ; nil ==> nil

⟨some #{4 5 9} [1 2 3 4]⟩
 ↳⟨or ⟨#{4 5 9} 1⟩ ; nil
      ...           ; nil
      ⟨#{4 5 9} 4⟩ ; 4    ==> 4

:>> ⟨dec 4⟩ ==> 3
```

Основы

Destructuring

```
(let [kv …
      [k v] kv])

(let [{k :k, v :v} m])

(let [{:keys [k v]} m])
```

Основы

Функции

```
(fn [x y]
  (+ x y))

#(+ %1 %2)

#(+ % %)
```

Основы

Функции-2

```
(def f (fn [x y] (+ x y)))

(defn f [x y]
  (+ x y))

(defn f
  ([x] …)
  ([x y] …))
```

Основы

Функции-3

```
(defn constrained-sqr
  [x]
  {:doc  "Documentation"
   :pre  [(pos? x)]
   :post [(> % 16), (< % 225)]}
  (* x x))
```

Основы

Concurrency — vars

Хранят значения «глобальных» функций и переменных, то, что хранится в namespace.

Thread-local rebind ⟨binding⟩

Основы

Concurrency — vars

```
Create/modify:
  (def x 1)
  (def ^:dynamic *debug* false)
  (binding [*debug* true] …)

Read value:
  x
  *debug*

Read var ref itself:
  #'x
```

Основы

Concurrency — atoms

Modify by applying a func
Not coordinated

Create:

```
(def a (atom {}))
```

Modify:

```
(swap!  a assoc :x 1)
(reset! a {:x 2})
```

Read:

```
@a
```

Основы

Concurrency — refs

Modify by applying a func

Coordinated

Accessible/modifiable inside ⟨dosync⟩ only

Основы

Concurrency — refs

```
Create:
  (def a (ref {}))

Modify:
  (dosync
    (alter   a assoc :x 1)
    (ref-set a {:x 2}))

Read:
  (dosync @a)
```

Основы

Concurrency — agents

Modify by “sending” a func
Will be applied in
  a different thread
Coordinated with STM
Always available to read

Основы

Concurrency — agents

```
Create:
  (def a (agent {}))

Modify:
  (~~dosync~~
    (send      a assoc :x 1)
    (send-off a {:x 2}))

Read:
  @a
```

Основы

Concurrency — delivering results

```
(let [f (future (do …))]
  @f)

(let [p (promise)]
  (future (deliver p :res))
  @p)
```

Никита Прокопов

tonsky.livejournal.com

Echo, Ульяновск

26 июня 2012

aboutecho.com

echorussia.ru

Обсуждение лекции:

tonsky.livejournal.com/265121.html