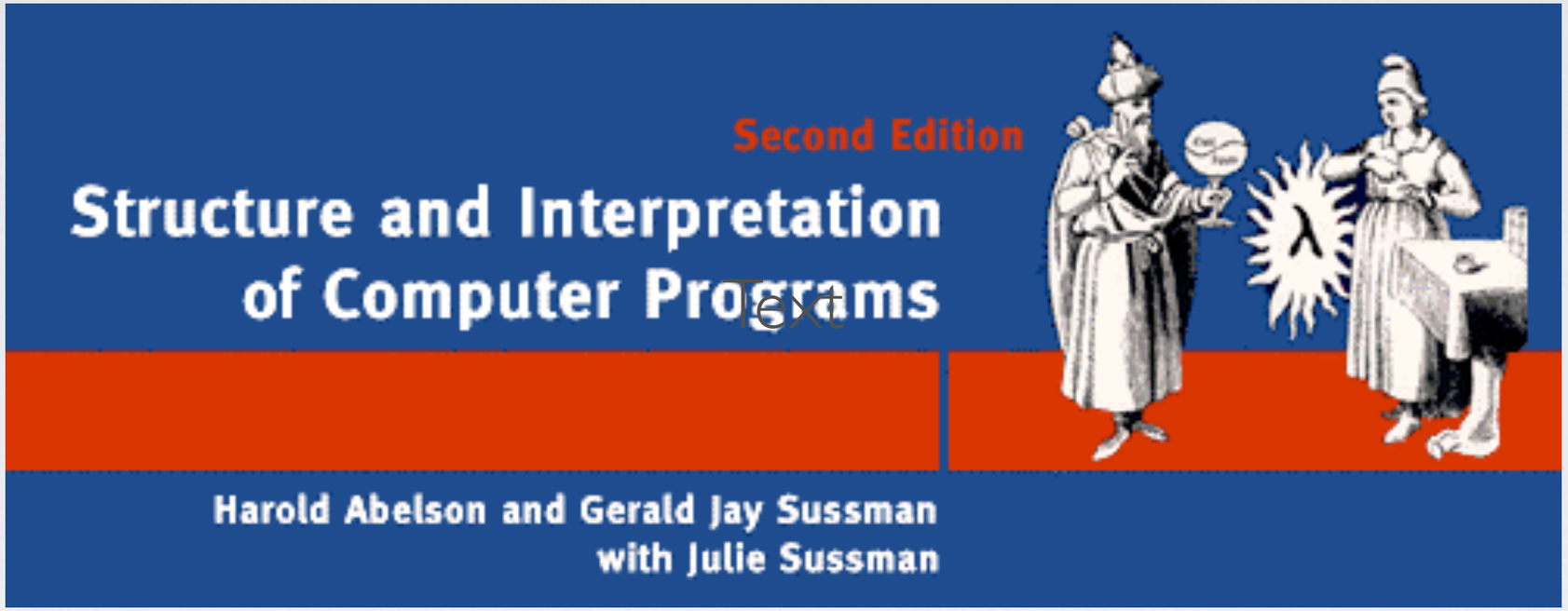
INTRODUCING CLOJURE TO A JAVA DEVELOPER

Edoardo Causarano

IMPERATIVE AND FUNCTIONAL PROGRAMMING



http://mitpress.mit.edu/sicp/

https://github.com/search?q=sicp

IMPERATIVE AND FUNCTIONAL PROGRAMMING

Substitution Model for Program Evaluation

immutable variables

referential transparency

IMPERATIVE AND FUNCTIONAL PROGRAMMING

```
(sum-of-squares (+ 5 1) (* 5 2))
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

1.1.5 The Substitution Model for Procedure Application

IMPERATIVE AND FUNCTIONAL PROGRAMMING

A language that supports the concept that "equals can be substituted for equals" in an expression without changing the value of the expression is said to be referentially transparent

IMPERATIVE AND FUNCTIONAL PROGRAMMING

Environment Model for Program Evaluation

variable assignment, mutability
local state
stack frame lookup of variables

IMPERATIVE AND FUNCTIONAL PROGRAMMING

```
(define (make-simplified-withdraw balance)
   (lambda (amount)
      (set! balance (- balance amount))
         balance))

(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

IMPERATIVE AND FUNCTIONAL PROGRAMMING

Pitfalls of imperative programming

IMPERATIVE AND FUNCTIONAL PROGRAMMING

"As we have seen, the set! operation enables us to model objects that have local state. However, this advantage comes at a price. Our programming language can no longer be interpreted in terms of the substitution model of procedure application..."

SICP 3.1.3

IMPERATIVE AND FUNCTIONAL PROGRAMMING

```
(set! product (* counter product))
(set! counter (+ counter 1))

(set! counter (+ counter 1))
(set! product (* counter product))
```

ERROR

SICP 3.1.3 p.309

IMPERATIVE AND FUNCTIONAL PROGRAMMING

"In general, programming with assignment forces us to carefully consider the relative orders of the assignments to make sure that each statement is using the correct version of the variables that have been changed. This issue simply does not arise in functional programs."

SICP 3.1.3

IMPERATIVE AND FUNCTIONAL PROGRAMMING

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently.”

SICP 3.1.3

LISP LANGUAGES

homiconicity and metaprogramming

functions as first order variables and higher order functions

pure functions and immutability

HOMOICONICITY

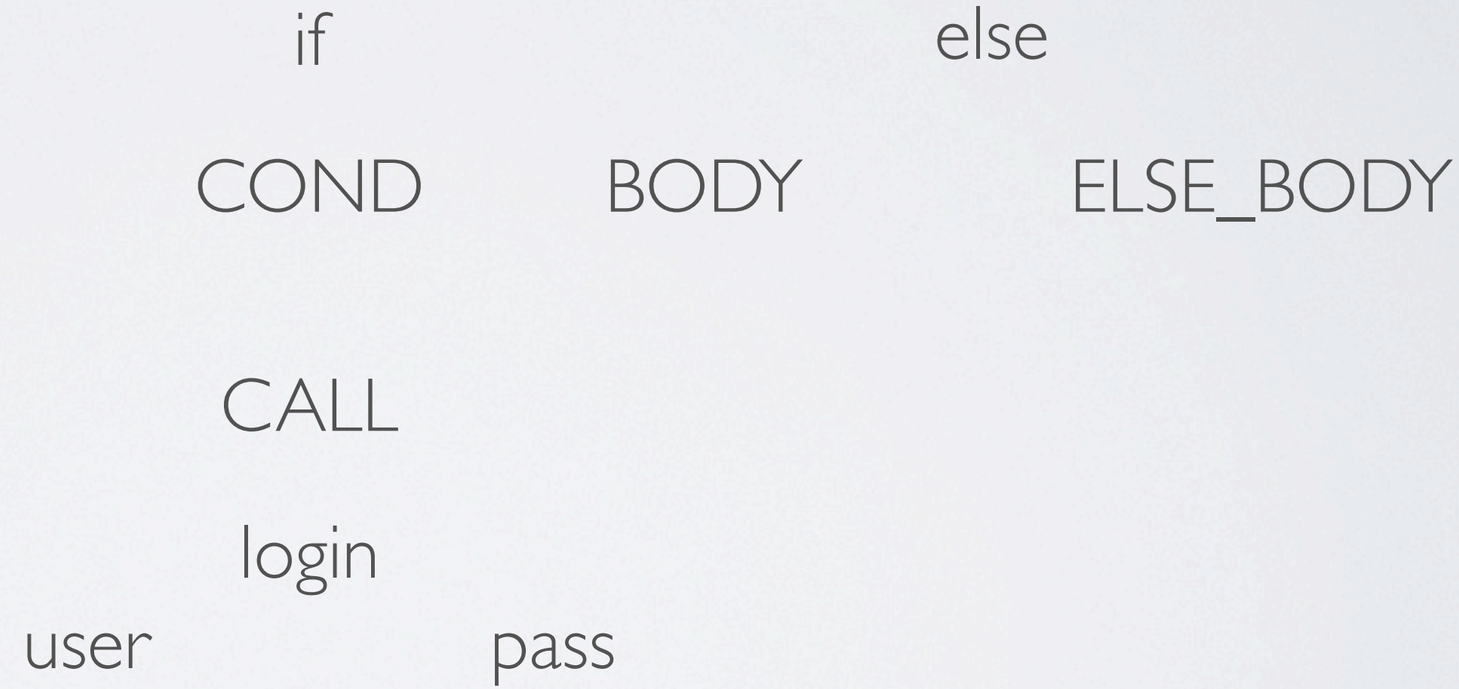
program instructions and program data have the same representational form

code data structures can be manipulated and executed (macros)

HOMOICONICITY

AST - Abstract Syntax Tree

```
if (login(user, pass)) {
  BODY;
} else {
  ELSE_BODY;
  }
```



HOMOICONICITY

- fundamental data structure is the list
- list evaluation
  - `first` - function symbol
  - `rest` - arguments to function

```
(if predicate then else?)

(if (login user pass)
(do-stuff))
```

HOMOICONICITY

```
                    if

        login                   (do-stuff) (do-else-stuff)


username      password
```

FUNCTIONS AND HIGHER ORDER FUNCTIONS

functions can be passed to other functions as argument

```
(map (fn [a] (+ a a)) [1 2])
=> [2 4]
```

OOP AS SEEN FROM CLOJURE

Draw 1-1 mappings between OOP Style (Java) and Functional Style (Clojure) is not entirely possible.

Even parallels between concepts are slightly off.

DATATYPES & PROTOCOLS

design by contract

`deftype` and `defrecord`

`defrecord` is also a persistent map

```
(defrecord Company [name address])
(defrecord Address [street city zip])

(def bk (Company. "De Bierkoning”
  (Address. "Paleisstraat 125”
        “Amsterdam”
        “1012ZL”)))

(:name bk)
-> "De Bierkoning"

(-> bk :address :city)
-> "Amsterdam”
```

DATATYPES & PROTOCOLS

`defprotocol` spefication only

`deftype defrecord` can implement multiple protocols

```
(defprotocol jazz
  foo [x]
  bar [x])

(deftype Yagh [a b]
  jazz
  (foo [x] a)
  (bar [x] (* b b))

  blues
  ...
  )

(bar (Yagh. 2 3))
=> 9
```

POLIMORPHISM - ARITY

```
(defn argcount
  ([] 0)
  ([x] 1)
  ([x y] 2)
  ([x y & more] (+ (argcount x y) (count more))))

(argcount)
=> 0

(argcount  a b c d)
=> 4
```

POLIMORPHISM - MULTIMETHODS

Use **`defmulti`** to define one and **`defmethod`** to implement the special case

```
(defn rect [width height] {:Shape :Rect, :width width, :height height})
(defn circle [radius] {:Shape :Circle, :radius radius})

(defmulti area :Shape)

(defmethod area :Rect [r]
    (* (:wd r) (:ht r)))
(defmethod area :Circle [c]
    (* (. Math PI) (* (:radius c) (:radius c))))
```

STATE

"A state is the value of an identity at a point in time"

Programming Cloujre, Ch 5.

STATE

Clojure does not have *pure functions*, side effects can occur anywhere

Referentially transparent, pure functions can be wrapped in a `(memoize ...)`

Concurrent access to State is not controlled with *locking* but with
STM (Software Transactional Memory)

STATE

In Clojure, state is classified as

- Refs - coordinated, synchronous mutation of shared state
- Atoms - uncoordinated, synchronous mutation of shared state
- Agents - asynchronous mutation of shared state
- Vars - thread-local state

STATE

In Clojure, state is classified as

- Refs - coordinated, synchronous mutation of shared state
- Atoms - uncoordinated, synchronous mutation of shared state
- Agents - asynchronous mutation of shared state
- Vars - thread-local state

STATE - REFS

```
(def a (ref 0))
(def b (ref 0))

(dosync
   (ref-set a 1)
   (alter b inc)
   )

(deref a)
=> 1

@b
=> 1
```

STATE - ATOMS

```
(def a (atom 0))

(deref a)
=> 0

(reset! a “Howdy!”)
(deref a)
=> 1
```

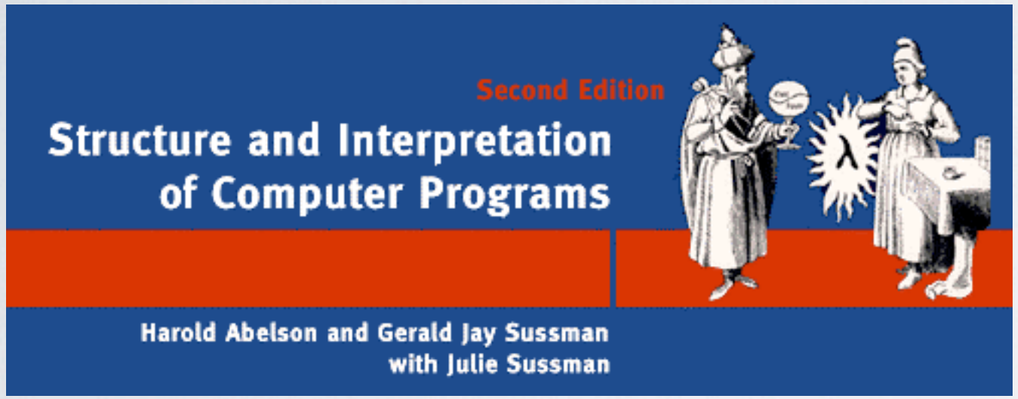
STATE - AGENTS

```
(def a (agent 0))

(send a inc)

... time passes,

(deref a)
```

STATE - VARS

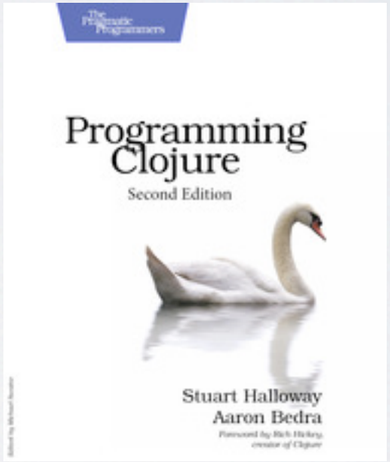
```
(binding [foo “howdy!”])

(println foo)
=> howdy!

(.start (Thread. #(println foo)))
=> nil
```

REFERENCE



Sussman J, Sussman J
"Structure and Interpretation of Computer Programs"
MIT Press, 1984

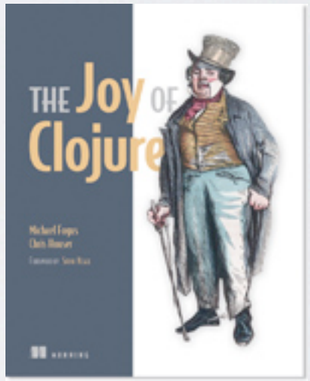
Halloway S, Bedra A
"Programming Clojure (2nd edition)"
Pragmatic Programmers, 2012



REFERENCE

Emerick C, Carper B, Grand C
"Clojure Programming - Practical Lisp for the Java World"
O'Reilly Media, 2012

Fogus M, Houser C
"The Joy of Clojure"
Manning Publications, 2011



FEEDBACK

edoardo.causarano@gmail.com