http://wedesoft.de/downloads/cambridge2014.pdf

Fundamentals of Computing

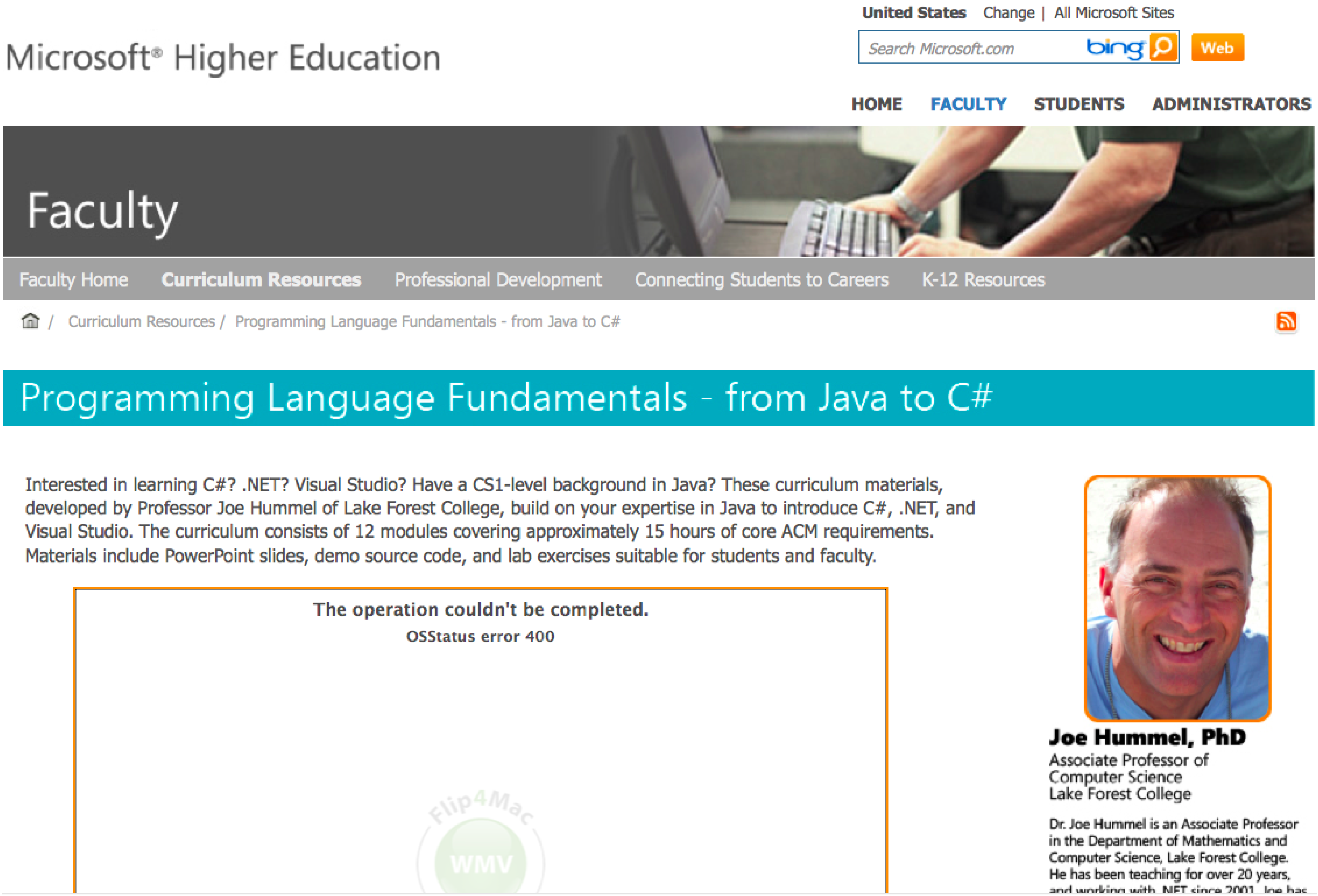
motivation

The Hundred-Year Language

Paul Graham

http://wedesoft.de/downloads/cambridge2014.pdf

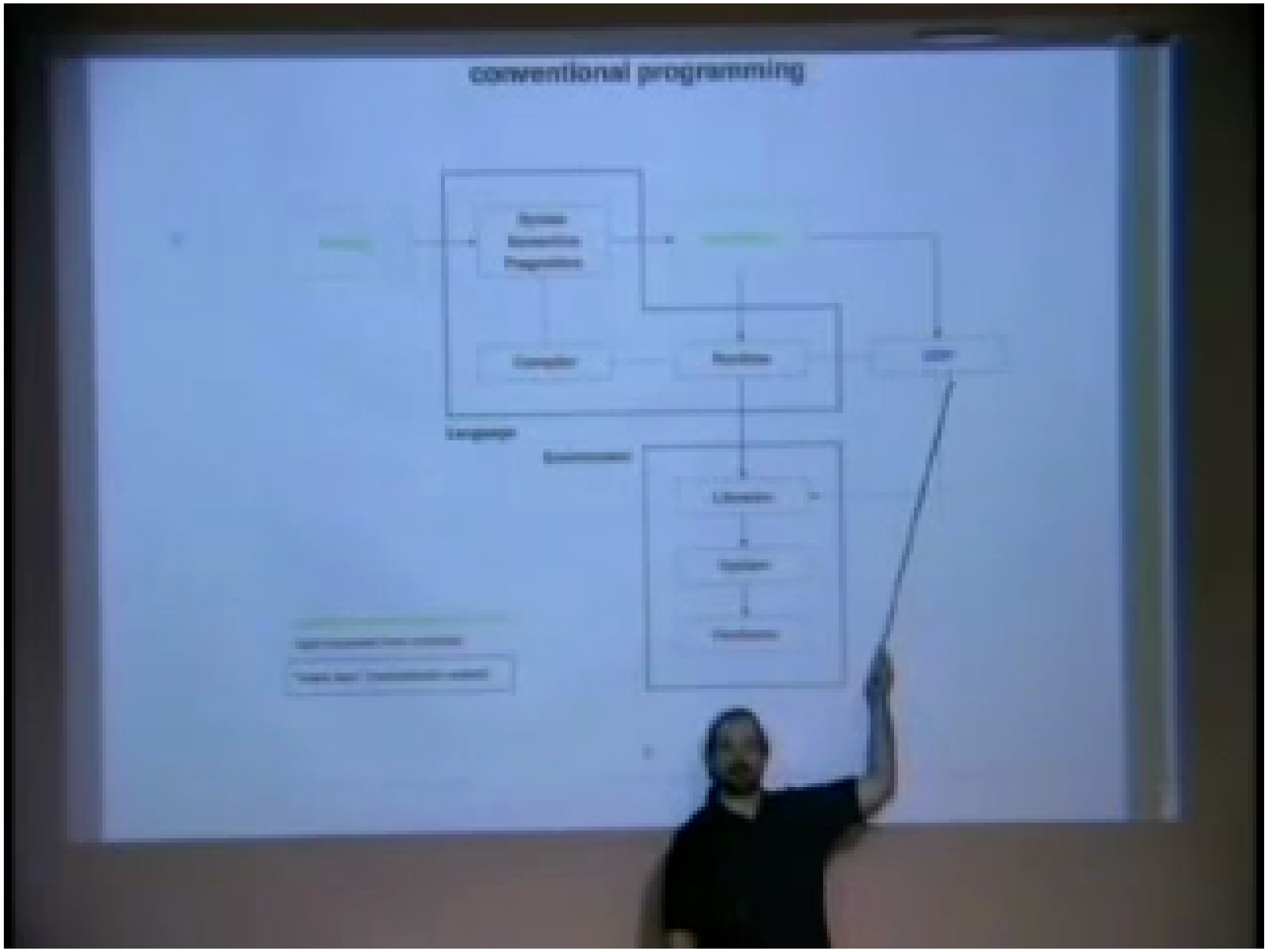
motivation
not fundamentals of computing



motivation

Building Your Own Dynamic Language

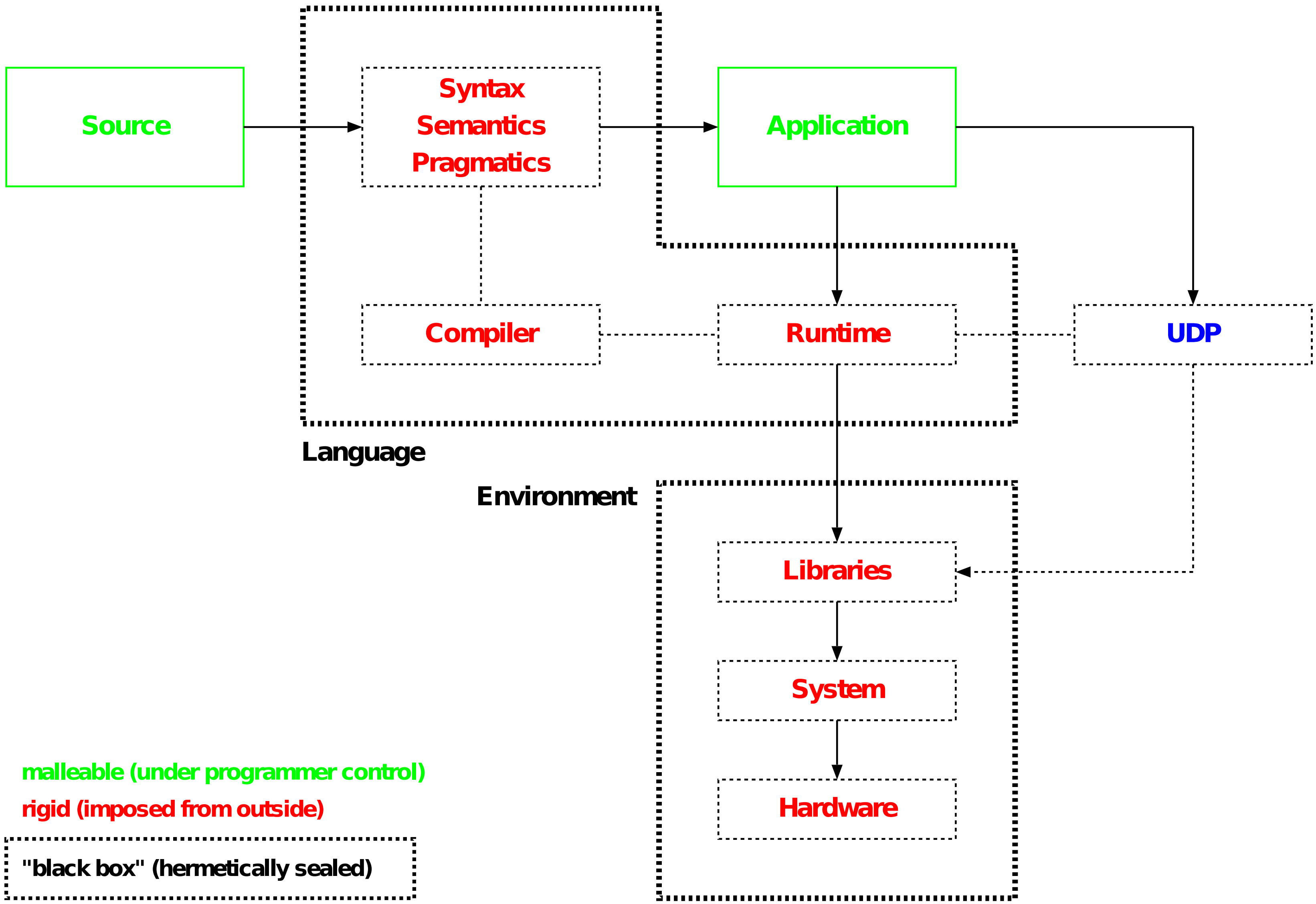
Ian Piumarta



http://piumarta.com/papers/EE380-2007-slides.pdf

motivation

Building Your Own Dynamic Language



λ-calculus
core

	Ruby	Scheme
$x, y, z, \ldots \in \mathcal{L}$	`x, y, z`	`x y z`
$M \in \mathcal{L} \Rightarrow \lambda x.M \in \mathcal{L}$	`lambda { |x| M }`	`(lambda (x) M)`
$M, N \in \mathcal{L} \Rightarrow (MN) \in \mathcal{L}$	`M.call(N)`	`(M N)`
recursive definition	Ruby	Scheme

abbreviations:

- $\lambda ab.M \equiv \lambda a.\lambda b.M$
- $MNO \ldots = ((\ldots (MN)O) \ldots)$

λ-calculus
Church encoding

$$\mathrm{id} \equiv \lambda x.x \qquad \mathrm{if}^{1} \equiv \lambda abc.abc$$

$$\mathrm{true} \equiv \lambda a.\lambda b.a \qquad \mathrm{and} \equiv \lambda mn.mnm$$

$$\mathrm{nil} \equiv \mathrm{false} \equiv \lambda a.\lambda b.b \qquad \mathrm{or} \equiv \lambda mn.mmn$$

$$\mathrm{not} \equiv \lambda m.\lambda ab.mba$$

$$=_{\mathrm{bool}} \equiv \lambda ab.ab(\mathrm{not}\ b)$$

[1]**lazy** evaluation

$$\mathrm{pair} \equiv \lambda htx.(\mathrm{if}\ xht)$$

$$\mathrm{head} \equiv \lambda l.(l\ \mathrm{true})$$

$$\mathrm{tail} \equiv \lambda l.(l\ \mathrm{false})$$

$$\mathrm{empty} \equiv \lambda l.(l\ (\lambda htb.\mathrm{false})\ \mathrm{true})$$

λ-calculus
binary numbers

$$
\begin{aligned}
0 &\equiv \text{nil} \\
1 &\equiv (\text{pair true nil}) \\
2 &\equiv (\text{pair true (pair false nil)}) \\
3 &\equiv (\text{pair true (pair true nil)}) \\
&\vdots
\end{aligned}
$$

Scheme

variable definitions $\Leftrightarrow$ λ-expressions

Scheme uses **eager** evaluation

```
((lambda (f)
  ((lambda (x y)
    (f x y))
   2 3))
  (lambda (a b) (+ a b)))
; 5
(let ((f (lambda (a b) (+ a b)))
      (x 2) (y 3))
  (f x y))
; 5
; (define f (lambda (a b) (+ a b)))
(define (f a b) (+ a b))
(define x 2) (define y 3)
(f x y)
; 5
```

Scheme

recursion ⇔ higher-order function

```
(define (neg l)
  (if (null? l)
      '()
      (cons (- (car l))
            (neg (cdr l)))))
(neg (list 1 2 3))
; (-1 -2 -3)
(sum (list 1 2 3))
(define (sum l)
  (if (null? l)
      0
      (+ (car l)
         (sum (cdr l)))))
(sum (list 1 2 3))
; 6
```

```
(use-modules (srfi srfi-1))
(map - (list 1 2 3))
; (-1 -2 -3)
(fold + 0 (list 1 2 3))
; 6
```

Scheme

currying

```
(use-modules (srfi srfi-1))
(use-modules (srfi srfi-26))
(map (cut + <> 1) (list 1 2 3))
; 2 3 4
```

Scheme
quote, eval & apply

```
(quote +)
; +
(quote (+ 1 2))
; (+ 1 2)
(car (quote (+ 1 2)))
; +
(eval (quote (+ 1 2)) (current-module))
; 3
(eval (car (quote (+ 1 2))) (current-module))
; #<procedure + (#:optional _ _ . _)>
(apply + (list 1 2))
; 3
```

Scheme

(hygienic) macros ⇔ lazy evaluation

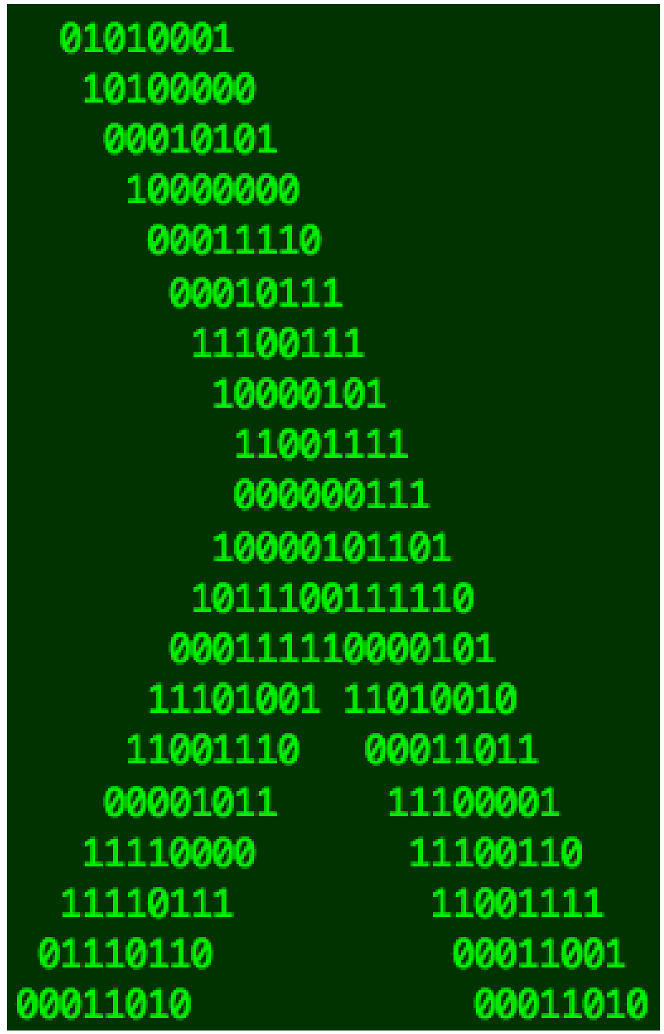
```
(define-syntax-rule (lazy expr) (lambda () expr))
(define-syntax-rule (force expr) (expr))
(define y (let ((x 2)) (lazy (+ x 3))))
y
; #<procedure 105992de0 at <current input>:10:0 ()>
(force y)
; 5
```

closures, monads, prompts, delimited continuations, combinators, reification, multiple dispatch, generics using functions, Y-combinator, Iota & Jot, reflection, inspection, readers, Factor, Haskell

More References

Binary Lambda Calculus

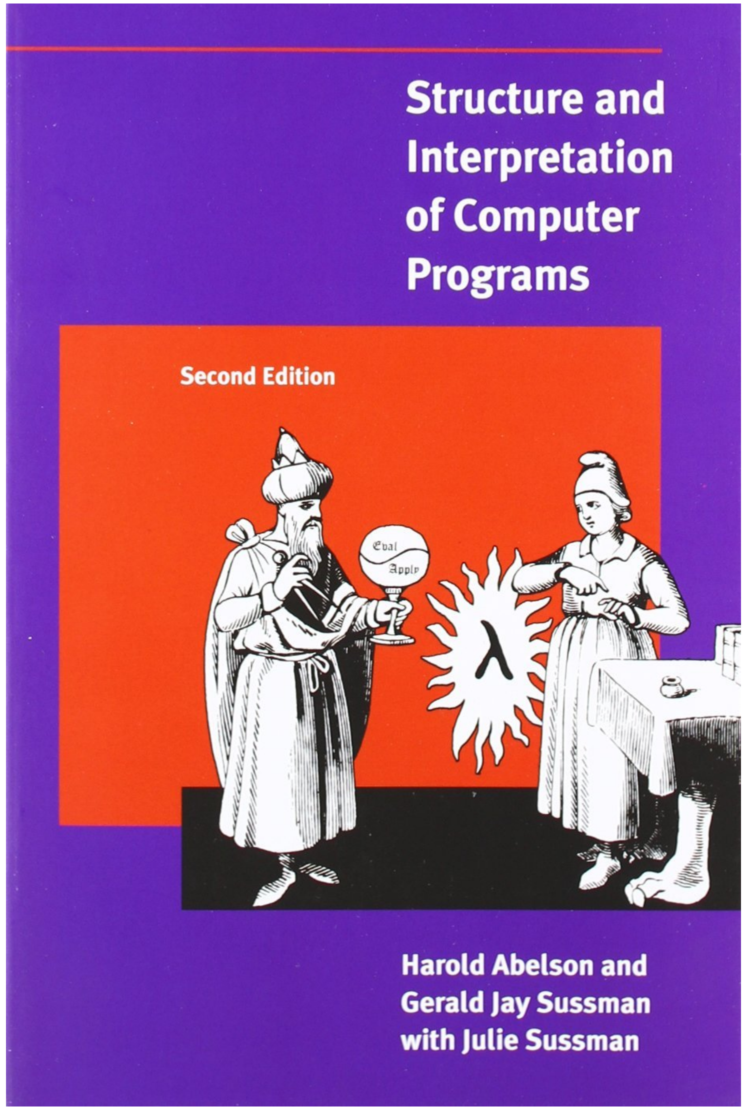
John Tromp



More References

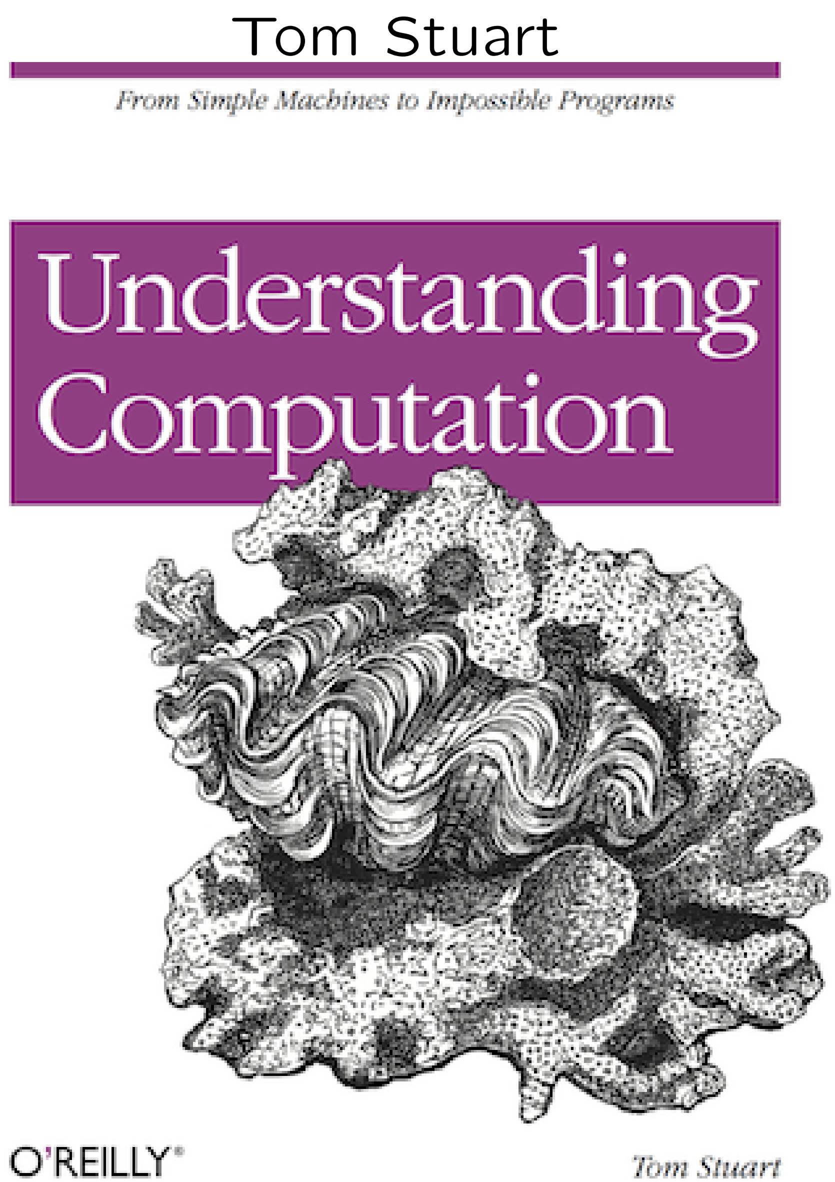
Structure and Interpretation of Computer Programs

Harold Abelson & Gerald Sussman



More References

Understanding Computation



More References

Schemer books

The { Little / Seasoned / Reasoned } Schemer: Q&A-style books