

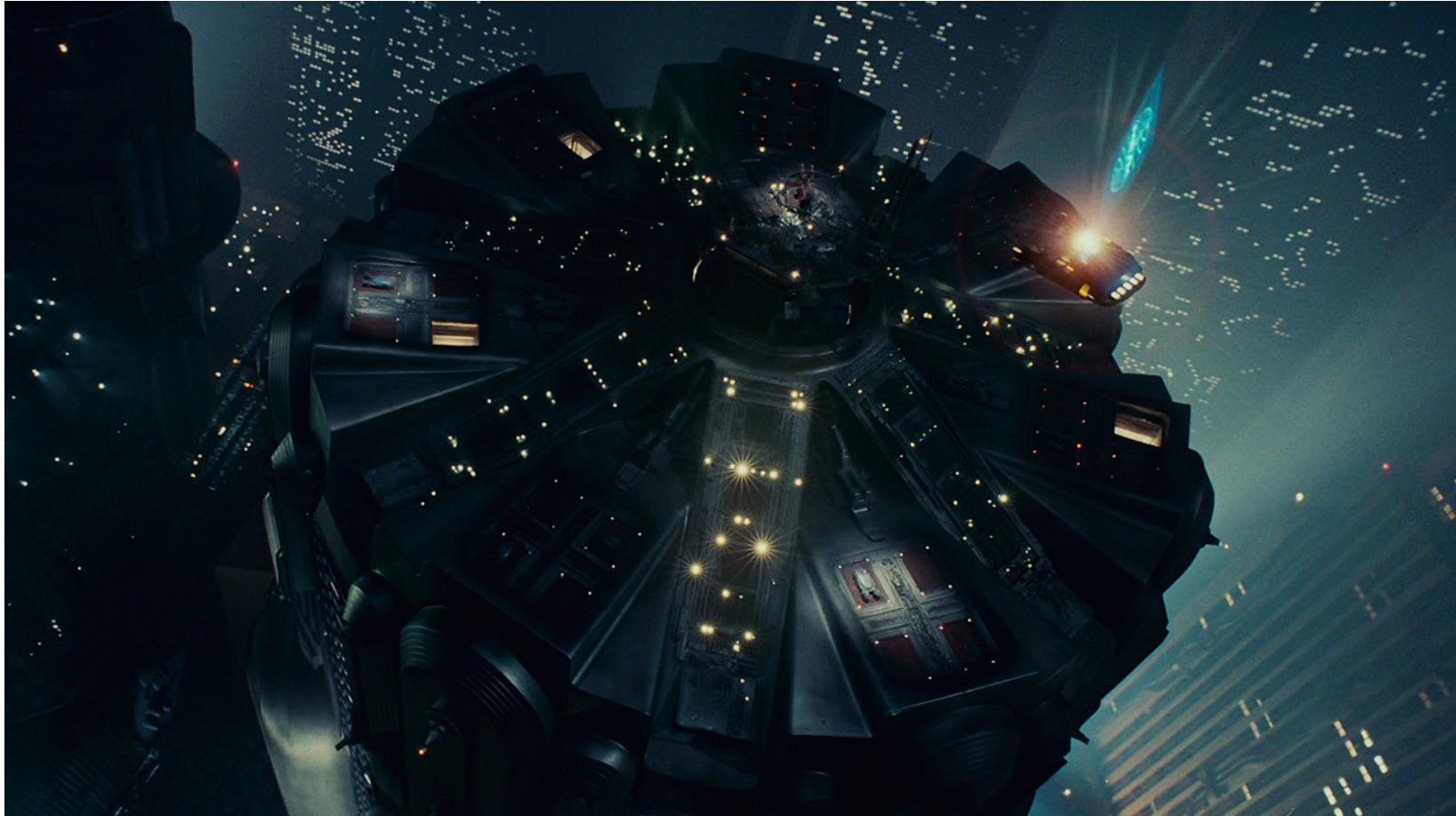
# Fundamentals of Computing



# motivation

## The Hundred-Year Language

Paul Graham







# motivation

## not fundamentals of computing

Microsoft® Higher Education



United States | Change | All Microsoft Sites

Search Microsoft.com  

HOME **FACULTY** STUDENTS ADMINISTRATORS

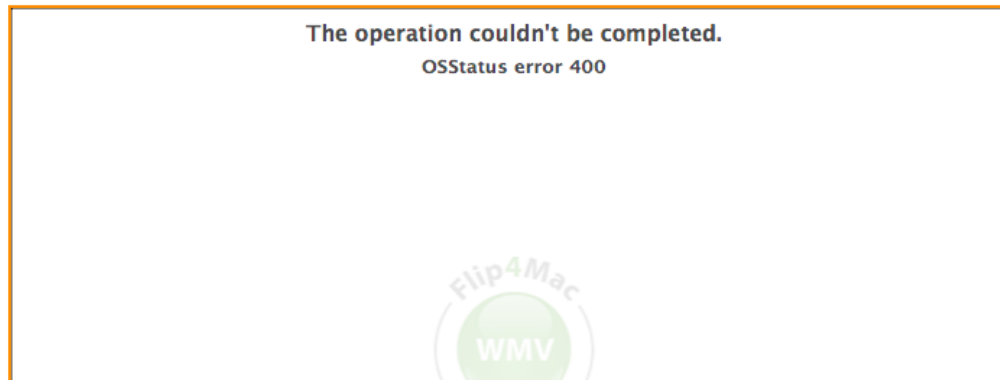
### Faculty

Faculty Home **Curriculum Resources** Professional Development Connecting Students to Careers K-12 Resources

 / Curriculum Resources / Programming Language Fundamentals - from Java to C# 

## Programming Language Fundamentals - from Java to C#

Interested in learning C#? .NET? Visual Studio? Have a CS1-level background in Java? These curriculum materials, developed by Professor Joe Hummel of Lake Forest College, build on your expertise in Java to introduce C#, .NET, and Visual Studio. The curriculum consists of 12 modules covering approximately 15 hours of core ACM requirements. Materials include PowerPoint slides, demo source code, and lab exercises suitable for students and faculty.



**Joe Hummel, PhD**

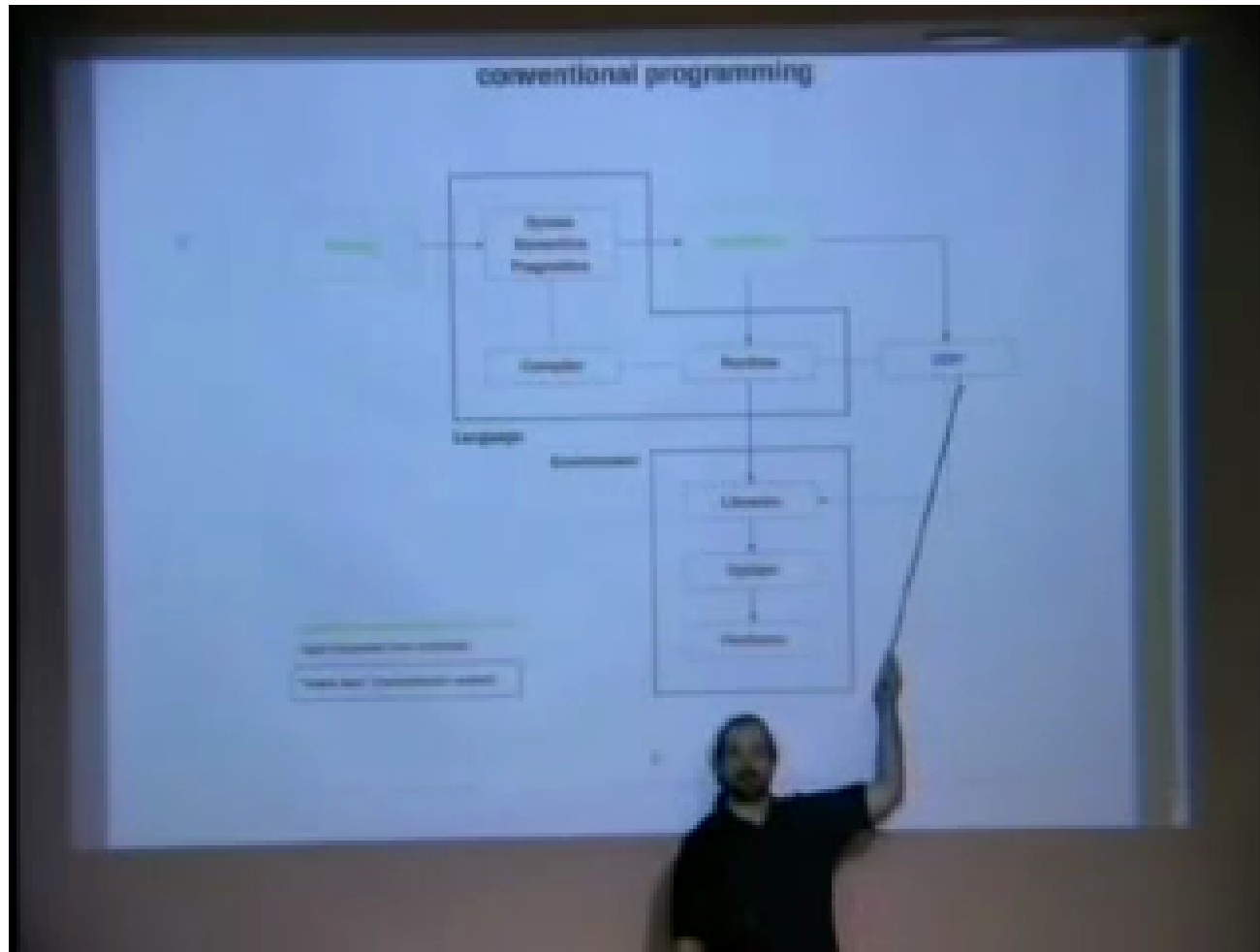
Associate Professor of  
Computer Science  
Lake Forest College

Dr. Joe Hummel is an Associate Professor in the Department of Mathematics and Computer Science, Lake Forest College. He has been teaching for over 20 years, and working with .NET since 2001. Joe has

# motivation

## Building Your Own Dynamic Language

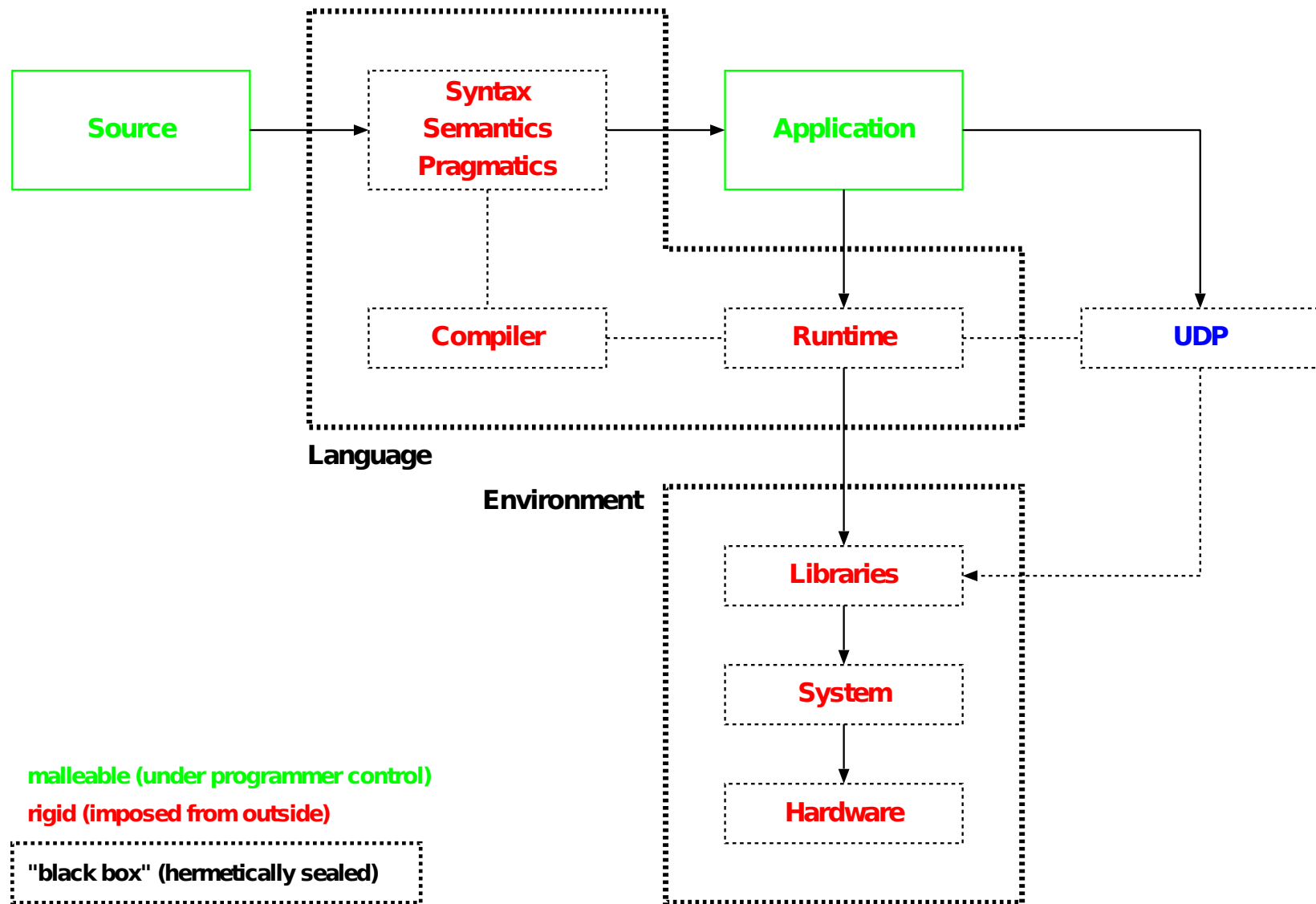
Ian Piumarta



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

# motivation

## Building Your Own Dynamic Language



# λ-calculus

## core

$x, y, z, \dots \in \mathcal{L}$	$x, y, z$	$x \ y \ z$
$M \in \mathcal{L} \Rightarrow \lambda x.M \in \mathcal{L}$	<code>lambda {  x  M }</code>	<code>(lambda (x) M)</code>
$M, N \in \mathcal{L} \Rightarrow (MN) \in \mathcal{L}$	<code>M.call(N)</code>	<code>(M N)</code>
recursive definition	Ruby	Scheme

abbreviations:

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

# $\lambda$ -calculus

## Church encoding

$$\text{id} \equiv \lambda x.x \quad \text{if}^1 \equiv \lambda abc.abc$$

$$\text{true} \equiv \lambda a.\lambda b.a \quad \text{and} \equiv \lambda mn.mnm$$

$$\text{nil} \equiv \text{false} \equiv \lambda a.\lambda b.b \quad \text{or} \equiv \lambda mn.mmn$$

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

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

<sup>1</sup>**lazy** evaluation

$$\text{pair} \equiv \lambda ht x.(\text{if } xht)$$

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

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

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

# $\lambda$ -calculus

## binary numbers

$0 \equiv \text{nil}$

$1 \equiv (\text{pair true nil})$

$2 \equiv (\text{pair true (pair false nil)})$

$3 \equiv (\text{pair true (pair true nil)})$

$\vdots$



# Scheme

## variable definitions $\Leftrightarrow$ $\lambda$ -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 $\Leftrightarrow$ 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 $\Leftrightarrow$ 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

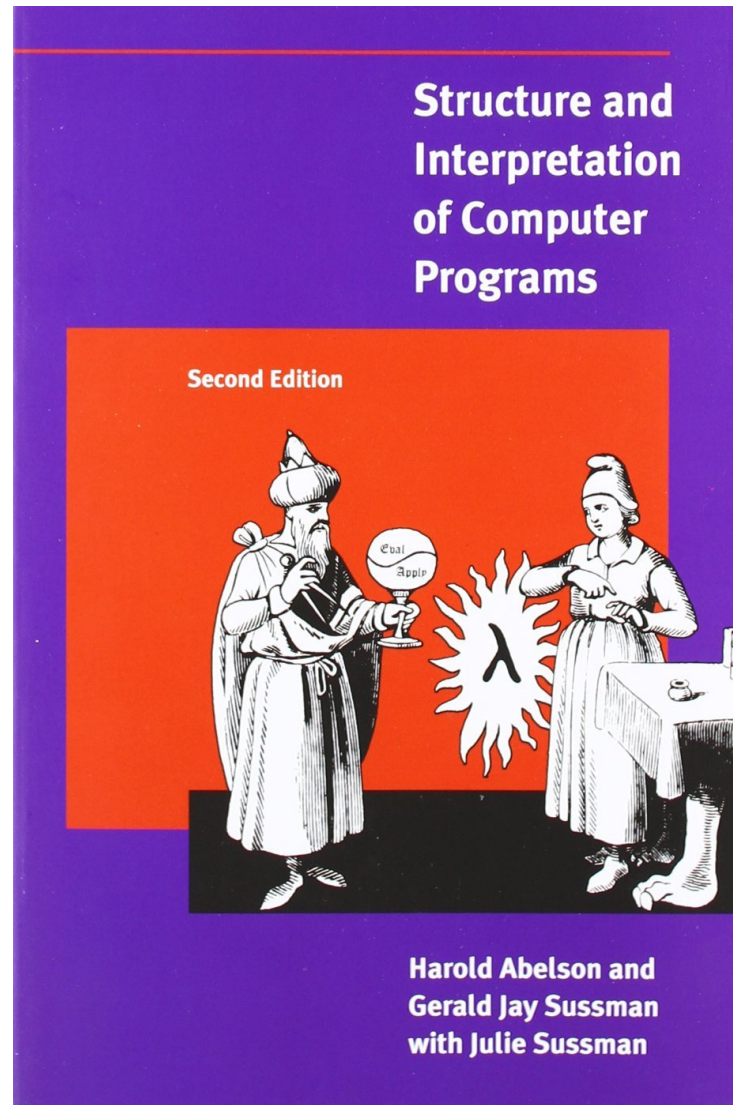
John Tromp

```
01010001
10100000
00010101
10000000
00011110
00010111
11100111
10000101
11001111
000000111
10000101101
1011100111110
000111110000101
11101001 11010010
11001110 00011011
00001011 11100001
11110000 11100110
11110111 11001111
01110110 00011001
00011010 00011010
```

# More References

## Structure and Interpretation of Computer Programs

Harold Abelson & Gerald Sussman



# More References

## Understanding Computation

Tom Stuart

*From Simple Machines to Impossible Programs*

## Understanding Computation



O'REILLY®

*Tom Stuart*

# More References

## Schemer books

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

