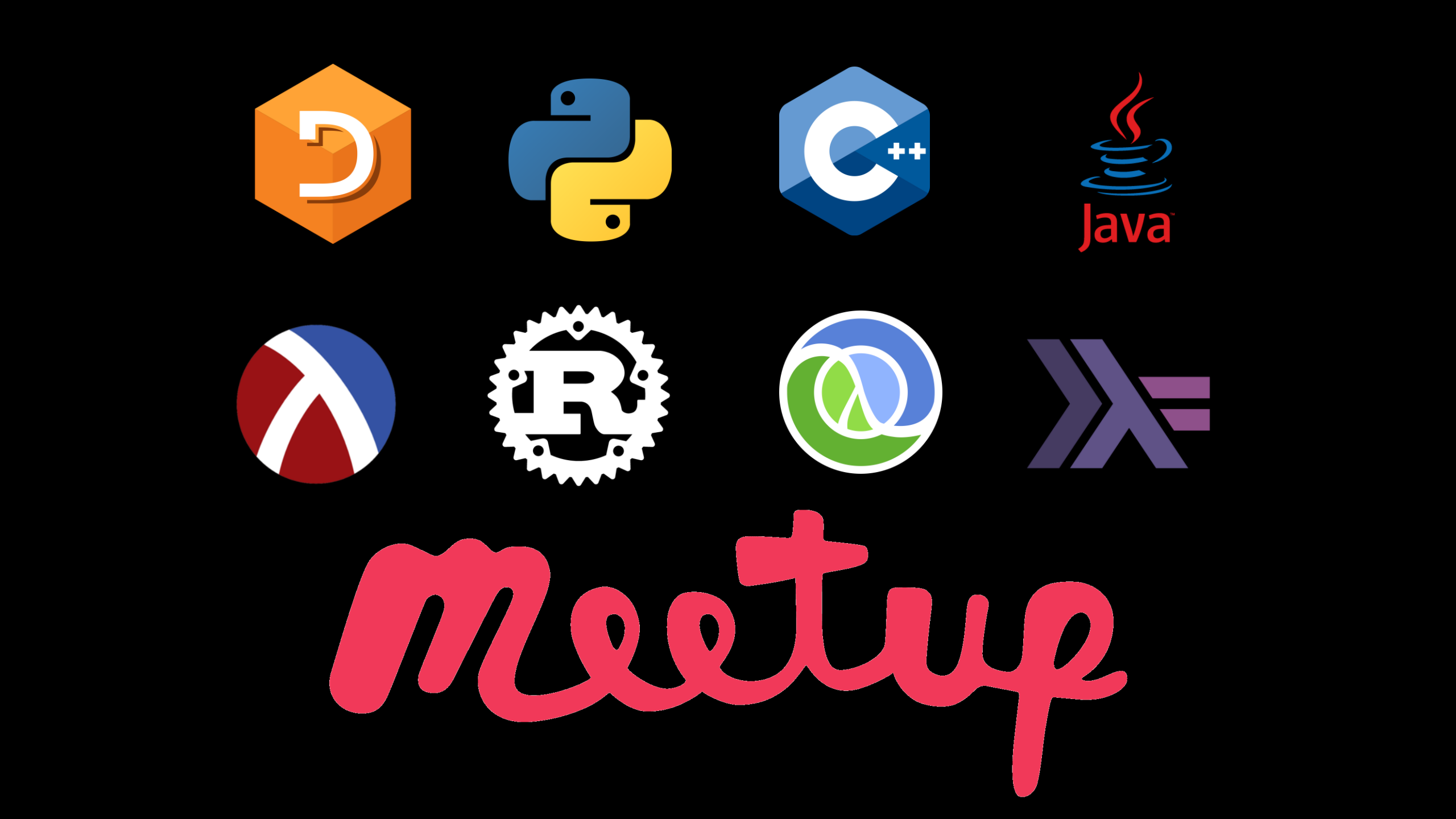
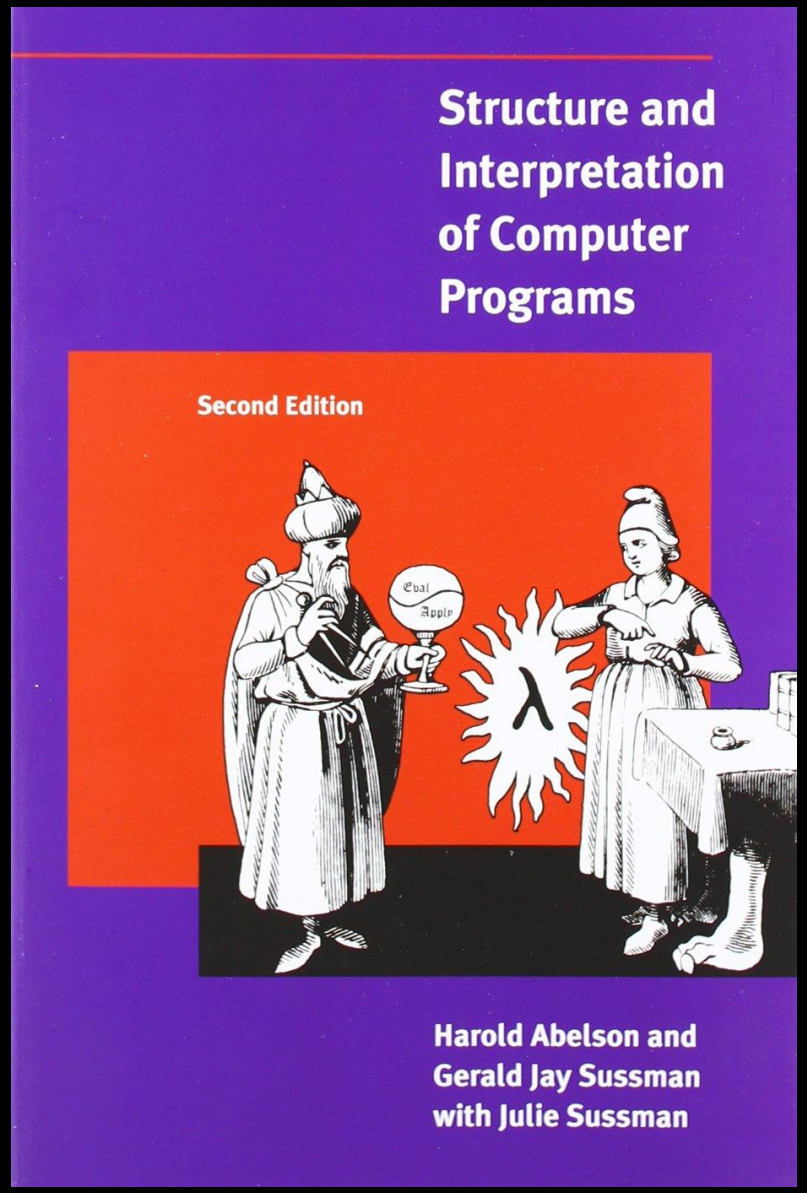

Java
meetup



Structure and Interpretation of Computer Programs

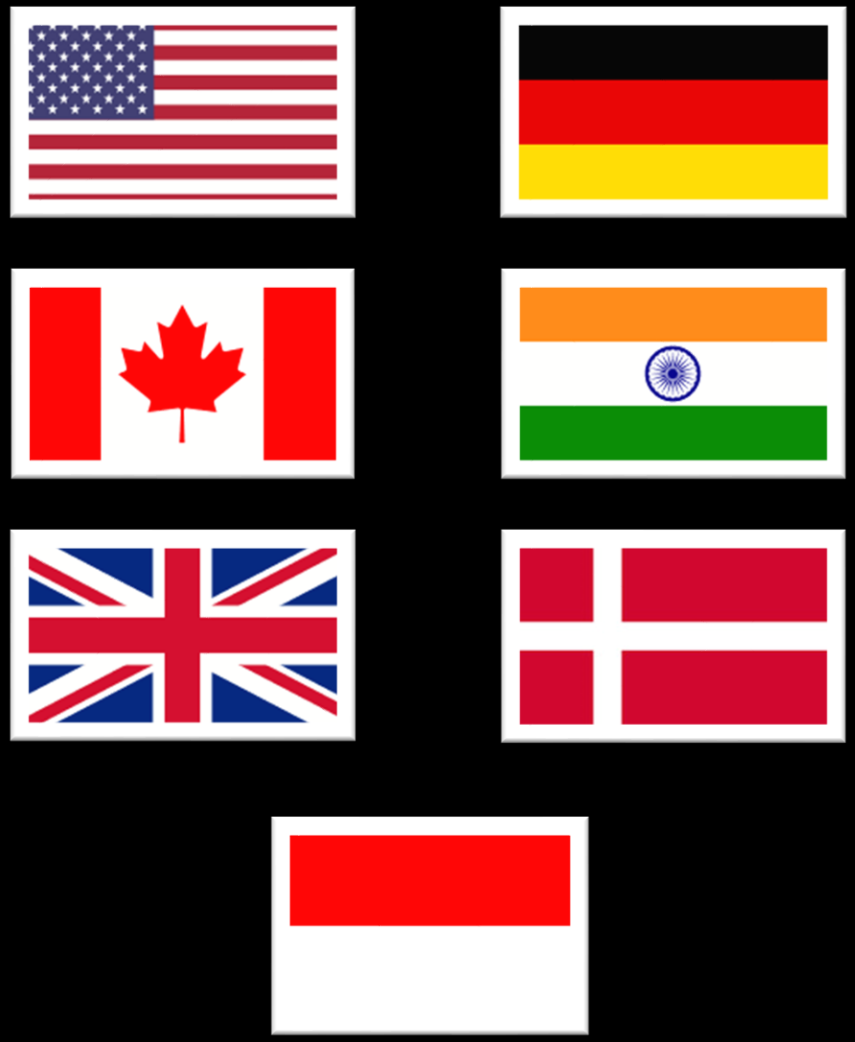
Chaper 2.1

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

In aggregate




DISCORD

Programming Languages Virtual Meetup

1 Tweet

Following

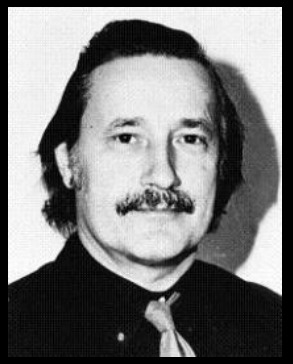
Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020

Who is Christopher Strachey?



- Co-founder of **Denotational Semantics** (Scott-Strachey semantics)
- Developed CPL, predecessor to BCPL
- Went to King's College (same as Turing)
- His influential set of lecture notes Fundamental Concepts in Programming Languages formalised the distinction between L- and R- values and introduced the terminology "ad hoc polymorphism", "parametric polymorphism", and "referential transparency".
- Coined the term currying

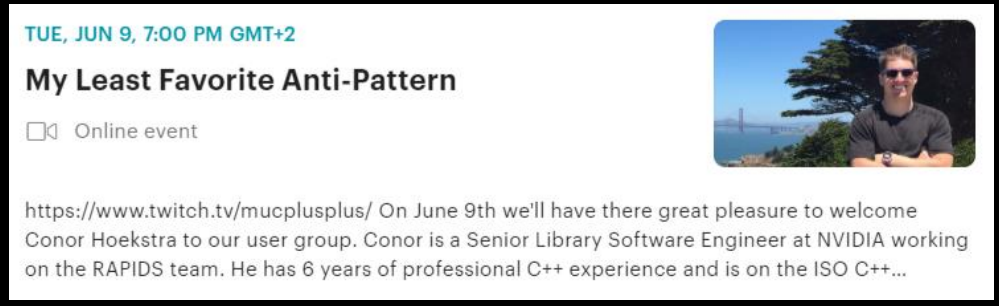
referential transparency

vs

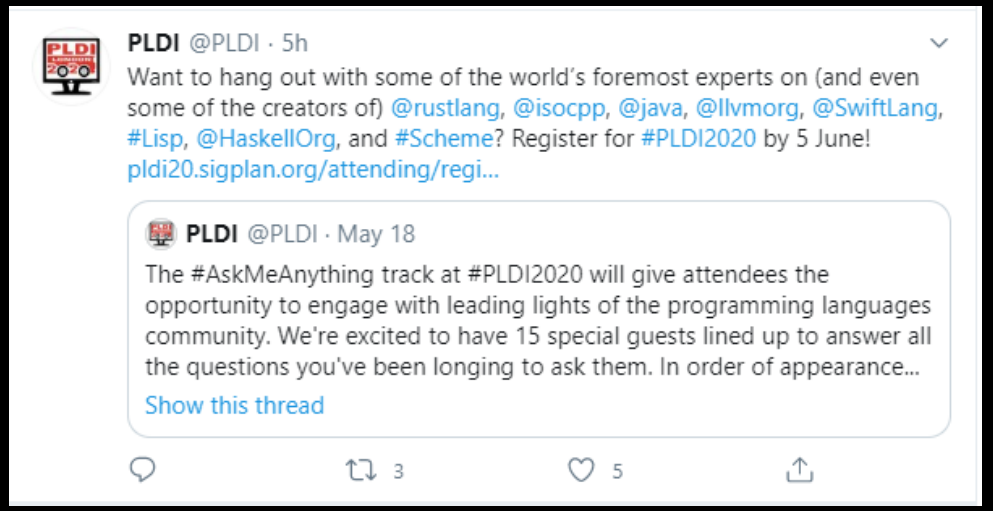
pure

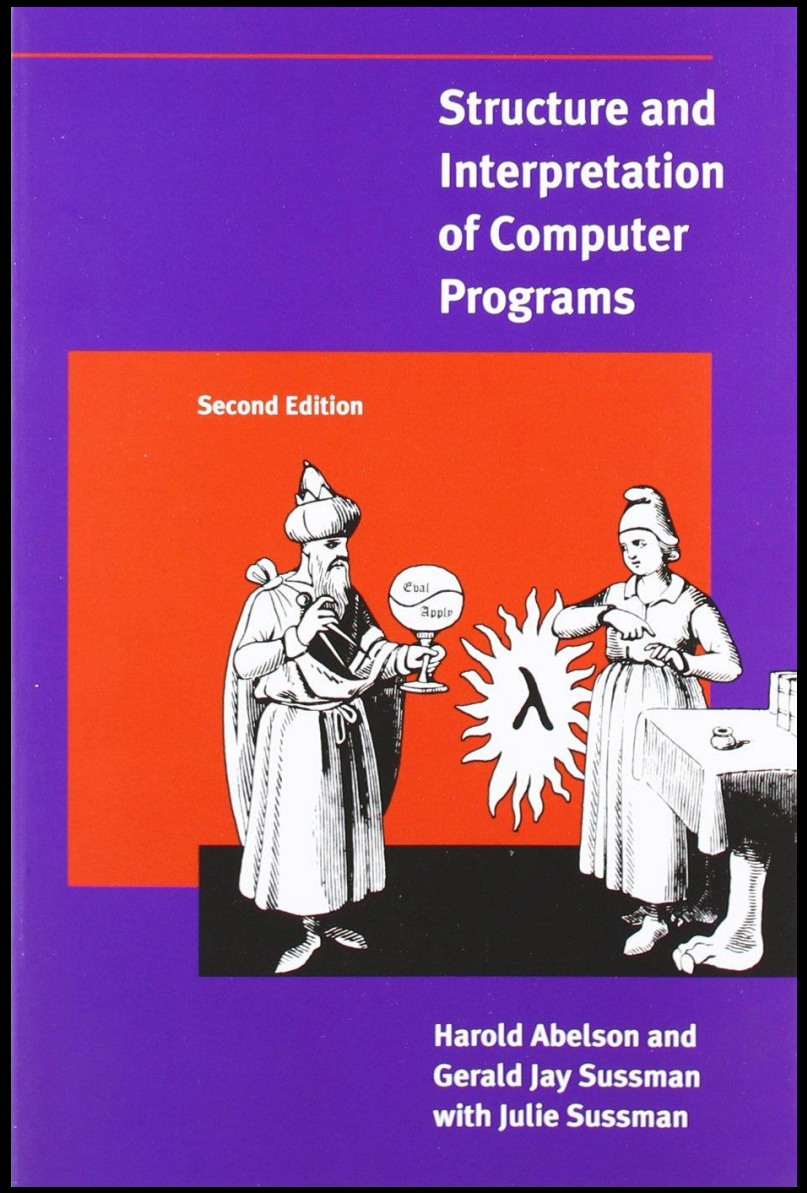
https://stackoverflow.com/questions/4865616/purity-vs-referential-transparency

Other Announcements



- **No meeting next week**
- **Giving "My Least Favorite Anti-Pattern" talk at:**
  - **MUC++ - June 9th 1:00 PM EST**
  - **Italian C++ - June 13th 9:00 AM EST**
- **PLDI 2020** Registration is open
- Set up June 22 PLDI Recap



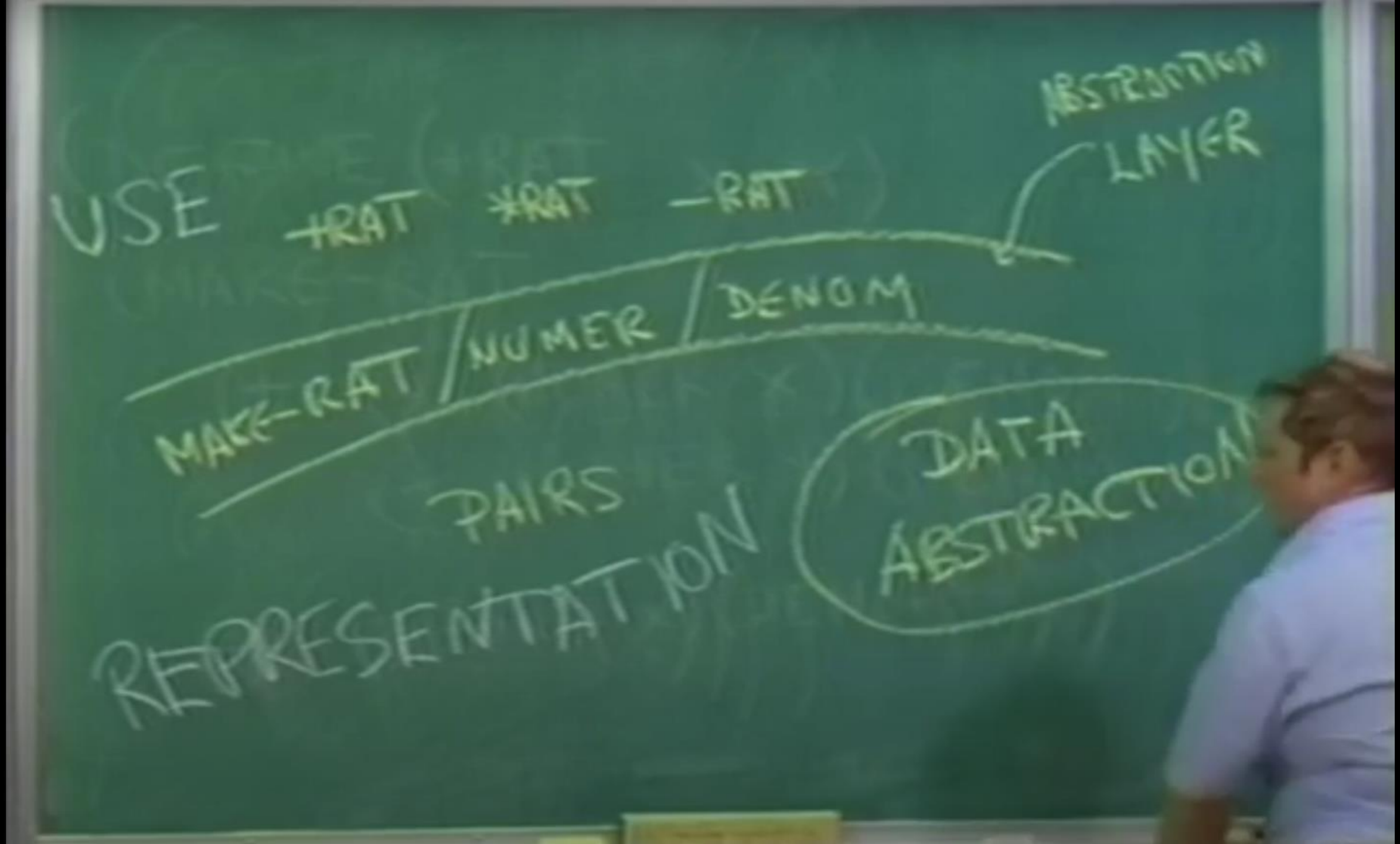


Structure and Interpretation of Computer Programs

Chaper 2.1

1. **Building Abstractions with Procedures**
    1. The Elements of Programming
    2. Procedures and the Processes they Generate
    3. Formulating Abstractions with Higher-Order Procedures

data abstraction
constructors
selectors

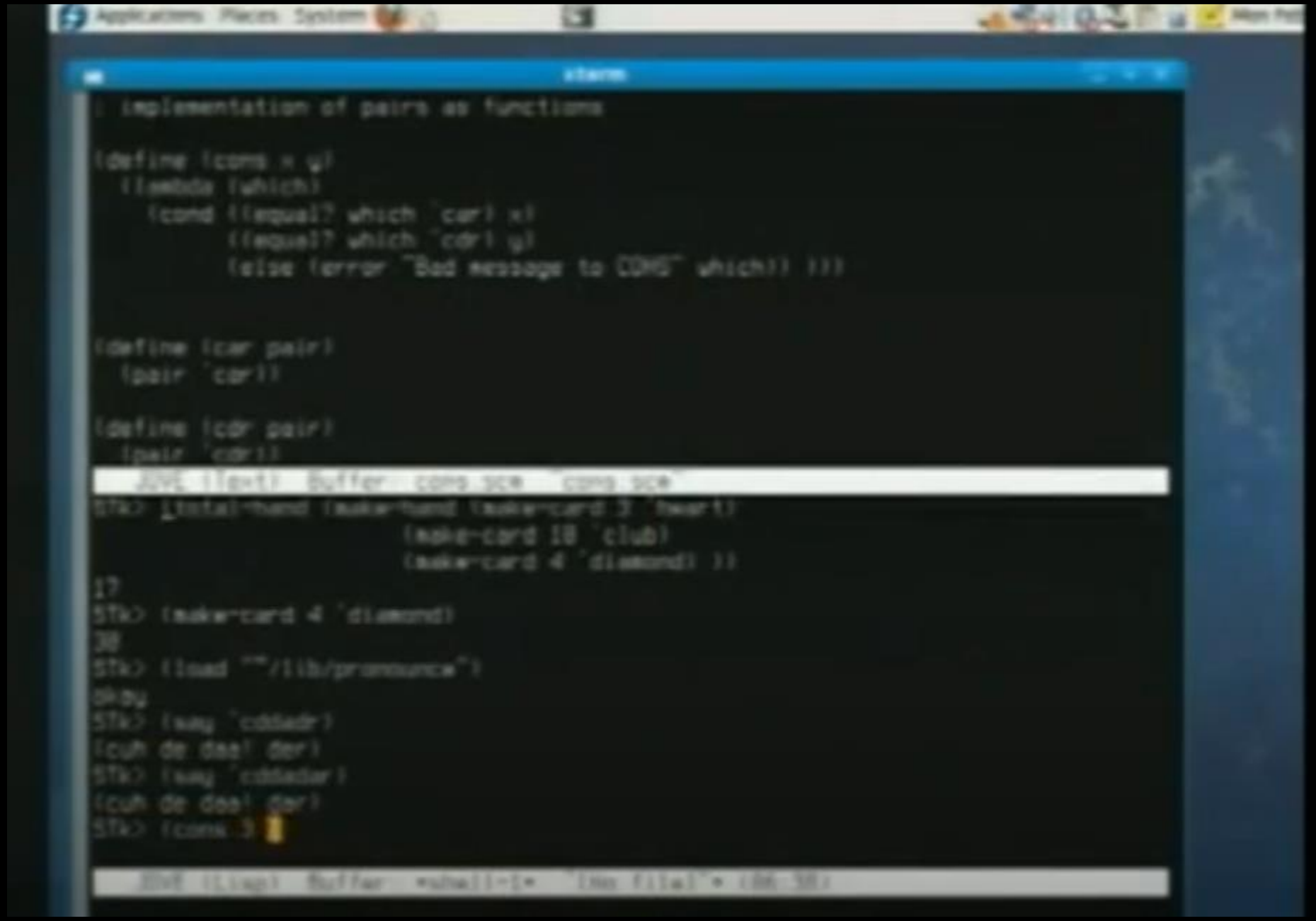

USE
+RAT
*RAT
−RAT
ABSTRACTION LAYER
MAKE-RAT
NUMER
DENOM
PAIRS
REPRESENTATION
DATA ABSTRACTION

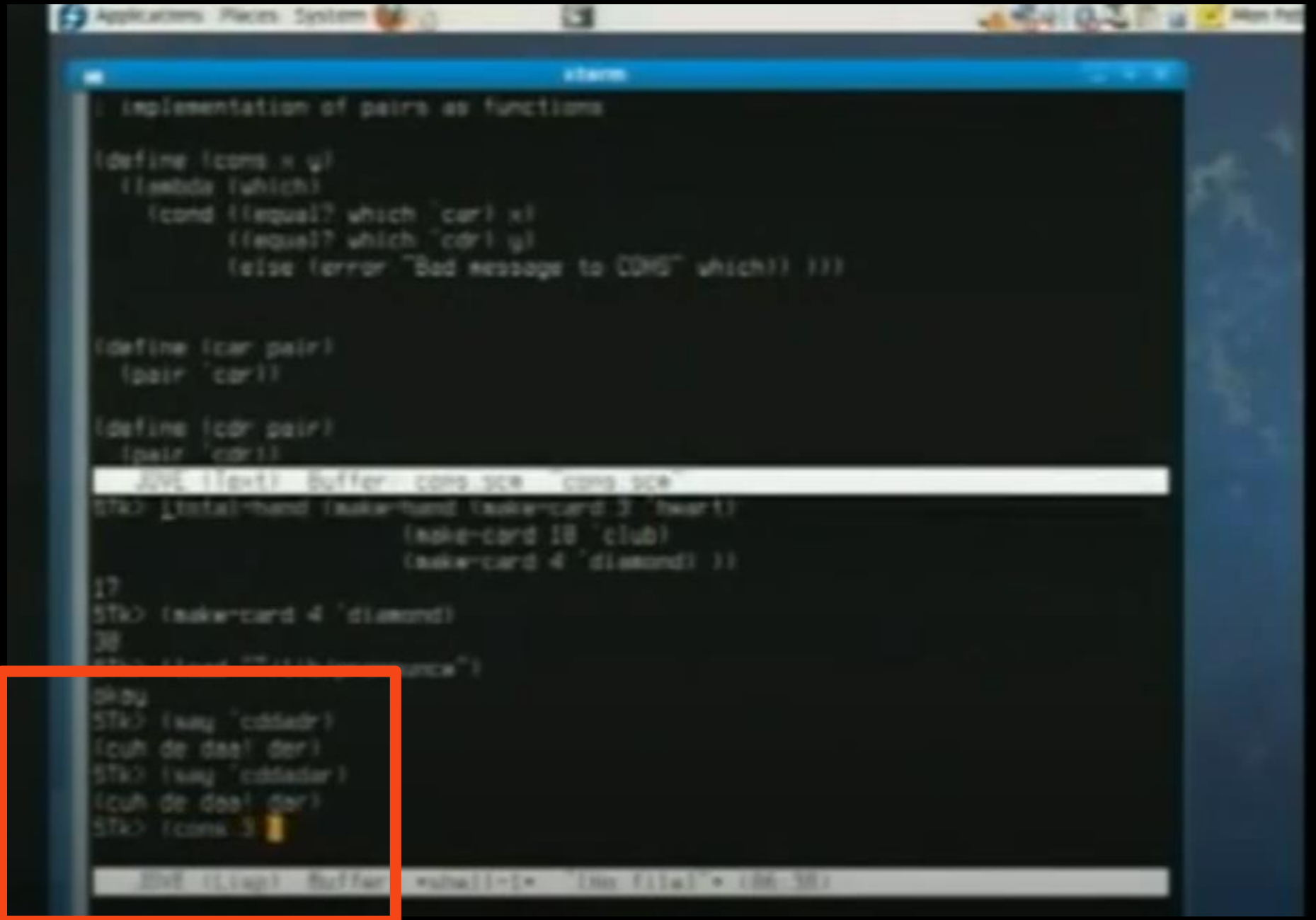
```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))

(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

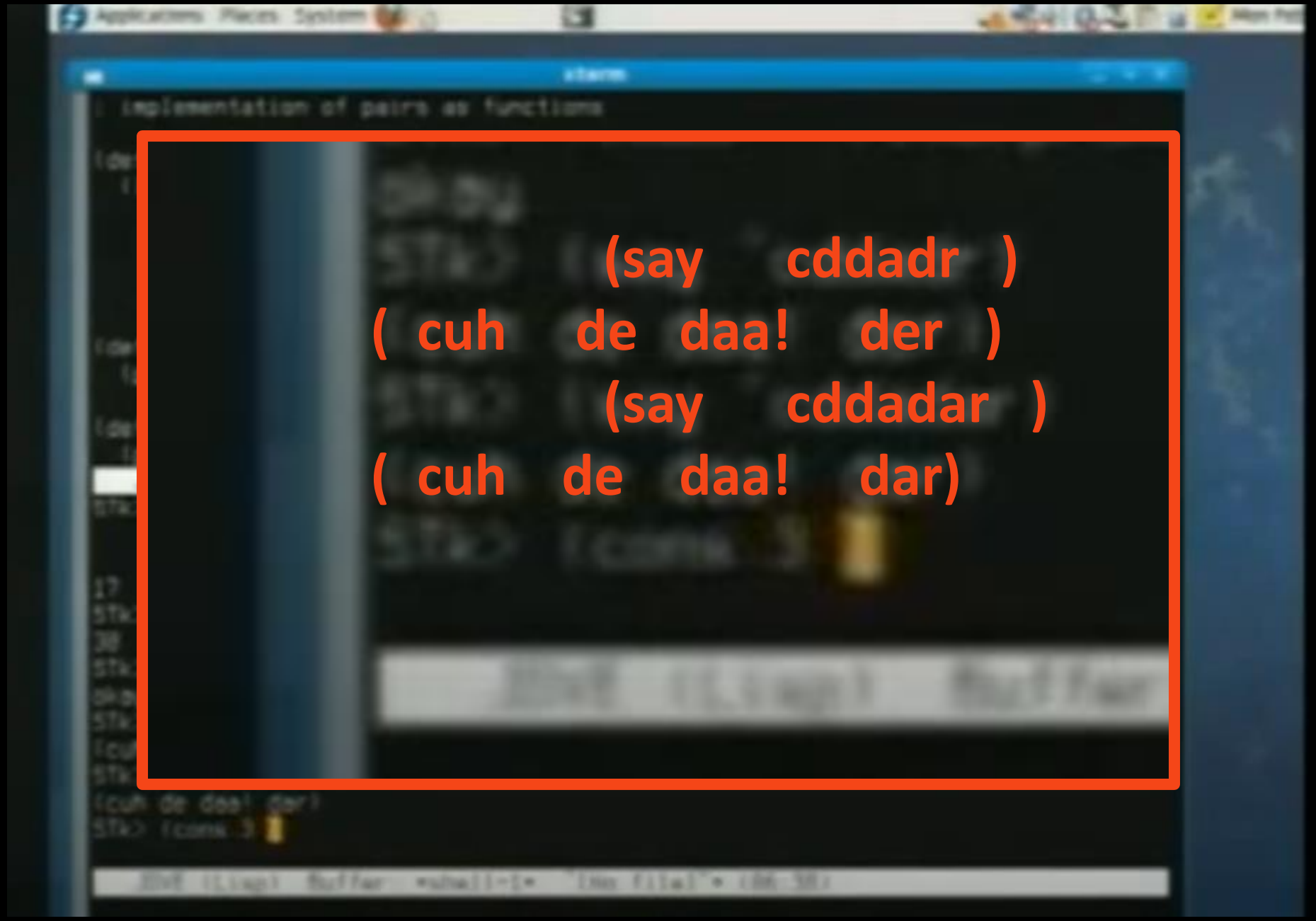
```
(define x (cons 1 2))
(car x)
1
(cdr x)
2
```

[2]The name `cons` stands for "construct." The names `car` and `cdr` derive from the original implementation of Lisp on the IBM 704. That machine had an addressing scheme that allowed one to reference the "address" and "decrement" parts of a memory location. `car` stands for "Contents of Address part of Register" and `cdr` (pronounced "could-er") stands for "Contents of Decrement part of Register."

COMPUTER


Applications Places System
xterm
; implementation of pairs as functions
(define (cons x y)
(lambda (which)
(cond ((equal? which 'car) x)
((equal? which 'cdr) y)
(else (error "Bad message to CONS" which)))))
(define (car pair)
(pair 'car))
(define (cdr pair)
(pair 'cdr))
-UUU:---F1 Buffer: cons.scm (cons.scm)
STk> (total-hand (make-hand (make-card 3 'heart)
(make-card 10 'club)
(make-card 4 'diamond)))
17
STk> (make-card 4 'diamond)
30
STk> (load ""/lib/pronounce")
okay
STk> (say 'cddadr)
(cuh de daa! der)
STk> (say 'cddadar)
(cuh de daa! dar)
STk> (cons 3 4)
-UUU:**-F1 (Lisp) Buffer: *shell*<1> "(no file)" 0% (88,38)


Applications Places System
xterm
; implementation of pairs as functions
(define (cons x y)
(lambda (which)
(cond ((equal? which 'car) x)
((equal? which 'cdr) y)
(else (error "Bad message to CONS" which)))))
(define (car pair)
(pair 'car))
(define (cdr pair)
(pair 'cdr))
STk> (total-hand (make-hand (make-card 3 'heart)
(make-card 10 'club)
(make-card 4 'diamond)))
17
STk> (make-card 4 'diamond)
30
okay
STk> (say 'caddadr)
(oh de daa! der)
STk> (say 'caddaddar)
(oh de daa! der)
STk> (cons 3


(say cddadr)
(cuh de daa! der)
(say cddadar)
(cuh de daa! dar)

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

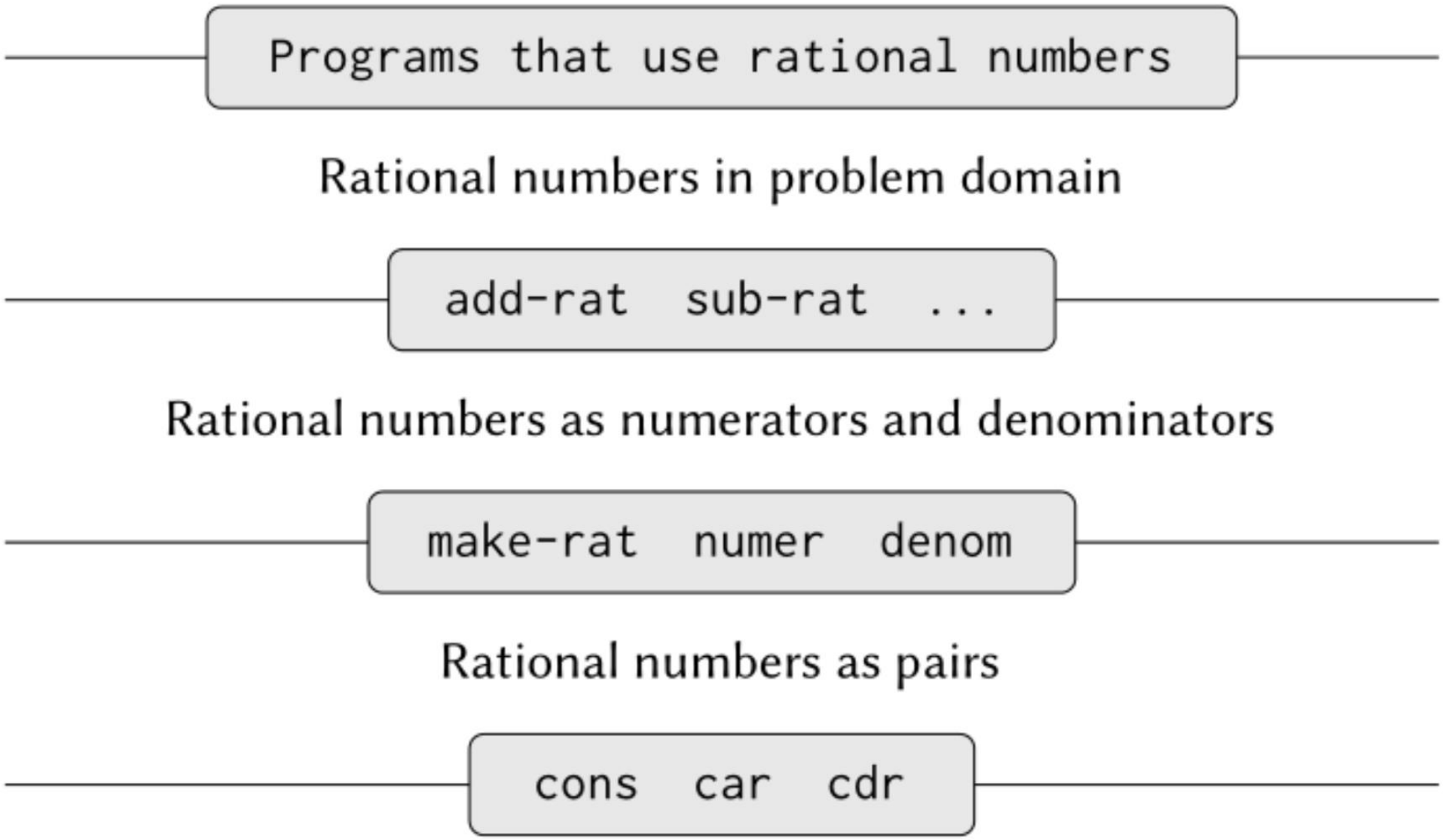
```
(define make-rat cons)
(define numer car)
(define denom cdr)
```

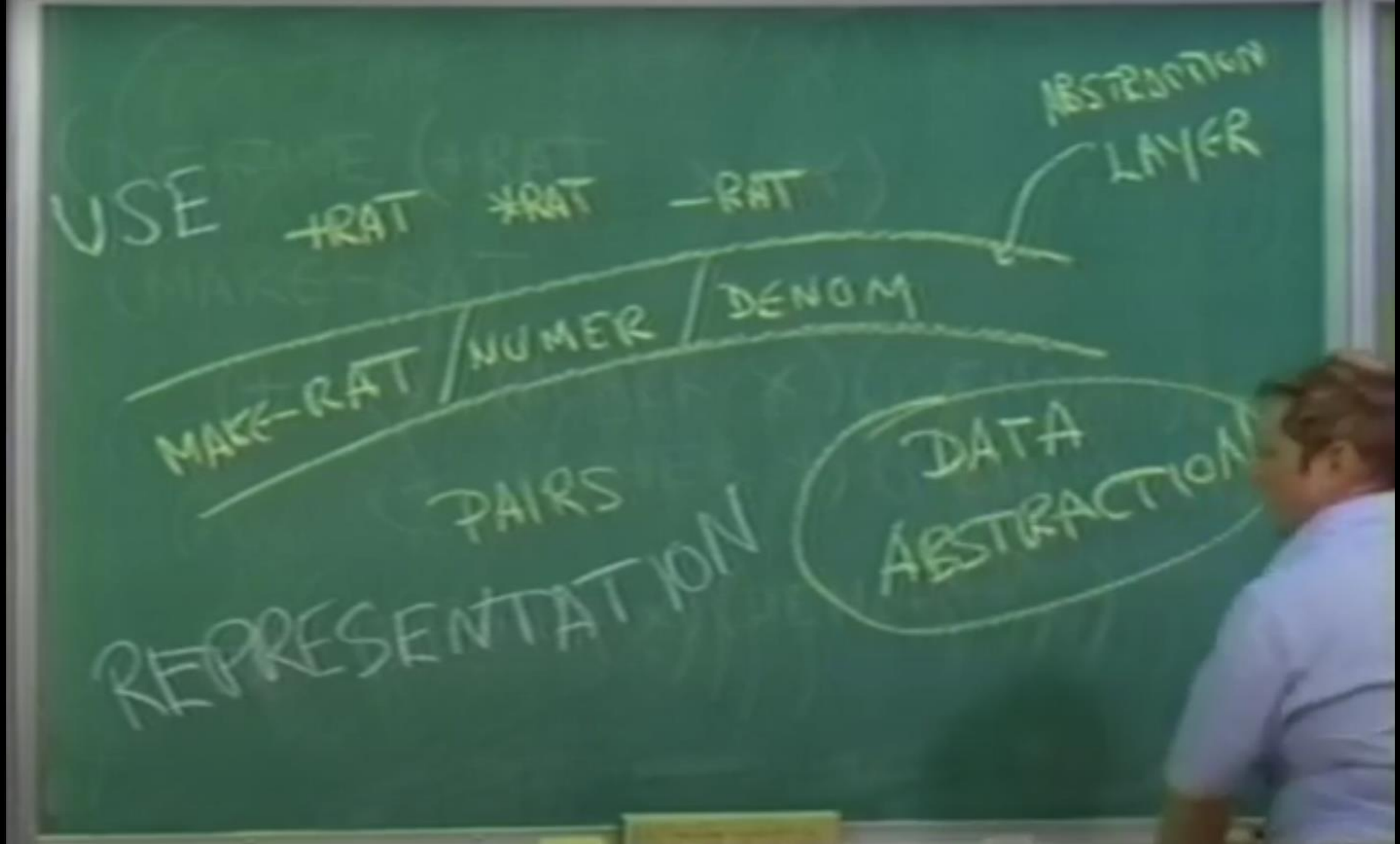
```
(define one-half (make-rat 1 2))
(print-rat one-half)
1/2
(define one-third (make-rat 1 3))
(print-rat (add-rat one-half one-third))
5/6
(print-rat (mul-rat one-half one-third))
1/6
(print-rat (add-rat one-third one-third))
6/9
```

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

Exercise 2.1: Define a better version of `make-rat` that handles both positive and negative arguments. `make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

```
;; Exercise 2.1 (page 118)


(define (make-rat n d)
  (let ((g ((if (< d 0) - +) (abs (gcd n d)))))
    (cons (/ n g) (/ d g))))
```


Programs that use rational numbers
Rational numbers in problem domain
add-rat sub-rat ...
Rational numbers as numerators and denominators
make-rat numer denom
Rational numbers as pairs
cons car cdr
However pairs are implemented


USE
+RAT
*RAT
−RAT
ABSTRACTION LAYER
MAKE-RAT
NUMER
DENOM
PAIRS
REPRESENTATION
DATA ABSTRACTION

```
(define (make-rat n d) (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

Exercise 2.2: Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints).

```
;; Exercise 2.2 (page 121/2)

(define make-segment cons)
(define start-segment car)
(define end-segment cdr)

(define make-point cons)
(define x-point car)
(define y-point cdr)

(define (segment-midpoint segment)
  (let ((x1 (x-point (start-segment segment)))
        (x2 (x-point (end-segment segment)))
        (y1 (y-point (start-segment segment)))
        (y2 (y-point (end-segment segment))))
  (make-point (/ (+ x1 x2) 2.0)
              (/ (+ y1 y2) 2.0))))
```

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))

;; > (define a (make-point -1 -1))
;; > (define b (make-point 5 2))
;; > (define line (make-segment a b))
;; > (print-point (segment-midpoint line))
;;
;; (2.0,0.5)
```

Exercise 2.3: Implement a representation for rectangles in a plane. (Hint: You may want to make use of Exercise 2.2.) In terms of your constructors and selectors, create procedures that compute the perimeter and the area of a given rectangle.

```
;; Exercise 2.3 (page 122)

;; Note: technically you can implement make-rectangle with only 3 points but
;;       for the sake of simplicity I will use 4 (and assume counter-clockwise)

(define (make-rectangle a b c d)
  (list a b c d))

(define first-point first)                                  ; car
(define (second-point rect) (first (rest rect)))            ; cadr
(define (third-point rect) (first (rest (rest rect))))      ; caddr
```

```
(define (sq x) (* x x))

(define (segment-length segment)
  (let* ((x1 (x-point (start-segment segment)))
         (x2 (x-point (end-segment segment)))
         (y1 (y-point (start-segment segment)))
         (y2 (y-point (end-segment segment)))
         (xdiff (- x1 x2))
         (ydiff (- y1 y2)))
    (cond ((= xdiff 0) (abs ydiff))
          ((= ydiff 0) (abs xdiff))
          (else (sqrt (+ (sq xdiff)
                         (sq ydiff)))))))
```

```
(define (rectangle-width rect)
  (segment-length
   (make-segment (first-point rect)
                 (second-point rect))))

(define (rectangle-height rect)
  (segment-length
   (make-segment (second-point rect)
                 (third-point rect))))

(define (rectangle-area rect)
  (* (rectangle-width rect)
     (rectangle-height rect)))

(define (rectangle-perimeter rect)
  (* 2 (+ (rectangle-width rect)
          (rectangle-height rect))))
```

```
(define a (make-point 0 0))
(define b (make-point 0 3))
(define c (make-point 4 3))
(define d (make-point 4 0))
(define r (make-rectangle a b c d))

(define aa (make-point 0 0))
(define bb (make-point -1 1))
(define cc (make-point 0 2))
(define dd (make-point 1 1))
(define rr (make-rectangle aa bb cc dd))

;; > (rectangle-area r)
;; 12
;; > (rectangle-perimeter r)
;; 14
;; > (rectangle-area rr)
;; 2.0000000000000004
;; > (rectangle-perimeter rr)
;; 5.656854249492381 ;; verify 4 * sqrt 2 = 5.65685424949
```

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1: CONS" m))))
  dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```

Exercise 2.4: Here is an alternative procedural representation of pairs. For this representation, verify that `(car (cons x y))` yields `x` for any objects `x` and `y`.

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
```

What is the corresponding definition of `cdr`? (Hint: To verify that this works, make use of the substitution model of Section 1.1.5.)

```
;; Exercise 2.4 (page 125)

(car (cons x y))
(car (lambda (m) (x y)))
(lambda (lambda (p q) p) (x y))
(lambda (x y) x)
(x) ;; :)

(define (car z)
  (z (lambda (p q) q)))
```

Exercise 2.5: Show that we can represent pairs of nonnegative integers using only numbers and arithmetic operations if we represent the pair a and b as the integer that is the product $2^a 3^b$. Give the corresponding definitions of the procedures `cons`, `car`, and `cdr`.

```
;; Exercise 2.5 (page 125)

(require algorithms) ;; repeat, product
(require threading)

(define (positive-pow base exp)
  (~>> (repeat exp base)
       (product)))

(define (make-pair a b)
  (* (positive-pow 2 a)
     (positive-pow 3 b)))

(define (factor-out n factor)
  (define (iter N acc)
    (if (< 0 (remainder N factor))
        acc
        (iter (/ N factor) (+ 1 acc))))
  (iter n 0))
```

```
(define (pair-fst pair) (factor-out pair 2))
(define (pair-snd pair) (factor-out pair 3))

;; > (define p (make-pair 4 6))
;; > (pair-fst p)
;; 4
;; > (pair-snd p)
;; 6
```

Exercise 2.6: In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as *Church numerals*, after its inventor, Alonzo Church, the logician who invented the λ-calculus.

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure `+` (not in terms of repeated application of `add-1`).

Conor Hoekstra
@code_report

@KevlinHenney is back with another awesome talk from @beautyincode! With references to 20+ different #programminglanguages including #LISP #SmallTalk #Ruby #JavaScript #Clojure #Haskell #cpp #Scala #Python #Perl #Scheme #Algol #Simula67 and #PowerShell youtu.be/0igQL-zrx-U 😆

Beauty in Code 2020, 4 of 6 — Kevlin Henney: "Lambda? You ...
Beauty in Code 2020 was a single-track full day IT-conference organized by Living IT, featuring five amazing speakers from ...
youtube.com

9:02 AM · Mar 23, 2020 · Twitter Web App

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By Alonzo Church.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

BEAUTYINCODE

Lambda? You Keep Using That Letter

Kevlin Henney

livingIT

$$0 \rightarrow \lambda f \cdot \lambda x \cdot x$$
$$1 \rightarrow \lambda f \cdot \lambda x \cdot f(x)$$
$$2 \rightarrow \lambda f \cdot \lambda x \cdot f(f(x))$$
$$3 \rightarrow \lambda f \cdot \lambda x \cdot f(f(f(x)))$$
$$4 \rightarrow \lambda f \cdot \lambda x \cdot f(f(f(f(x))))$$
$$5 \rightarrow \lambda f \cdot \lambda x \cdot f(f(f(f(f(x)))))$$
$$6 \rightarrow \lambda f \cdot \lambda x \cdot f(f(f(f(f(f(x))))))$$

BEAUTY IN CODE

Lambda? You Keep Using That Letter

Kevlin Henney

Exercise 2.6: In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as *Church numerals*, after its inventor, Alonzo Church, the logician who invented the λ-calculus.

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure `+` (not in terms of repeated application of `add-1`).

```
;; Exercise 2.6

(define one (lambda (f) (lambda (x) (f x))))
(define two (lambda (f) (lambda (x) (f (f x)))))
```

```
;; taken from http://community.schemewiki.org/?sicp-ex-2.6
 (define (add a b)
   (lambda (f)
     (lambda (x)
       ((a f) ((b f) x)))))
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

“They say that computer science is a lot like **magic** … [but] there is a bad part of computer science that is a lot like **religion**.

Gerald Sussman

MIT Lecture 2B: Data, SICP

5

L05 User Interface | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

6

L06 User Interface | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

7

L07 Orders of Growth | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

8

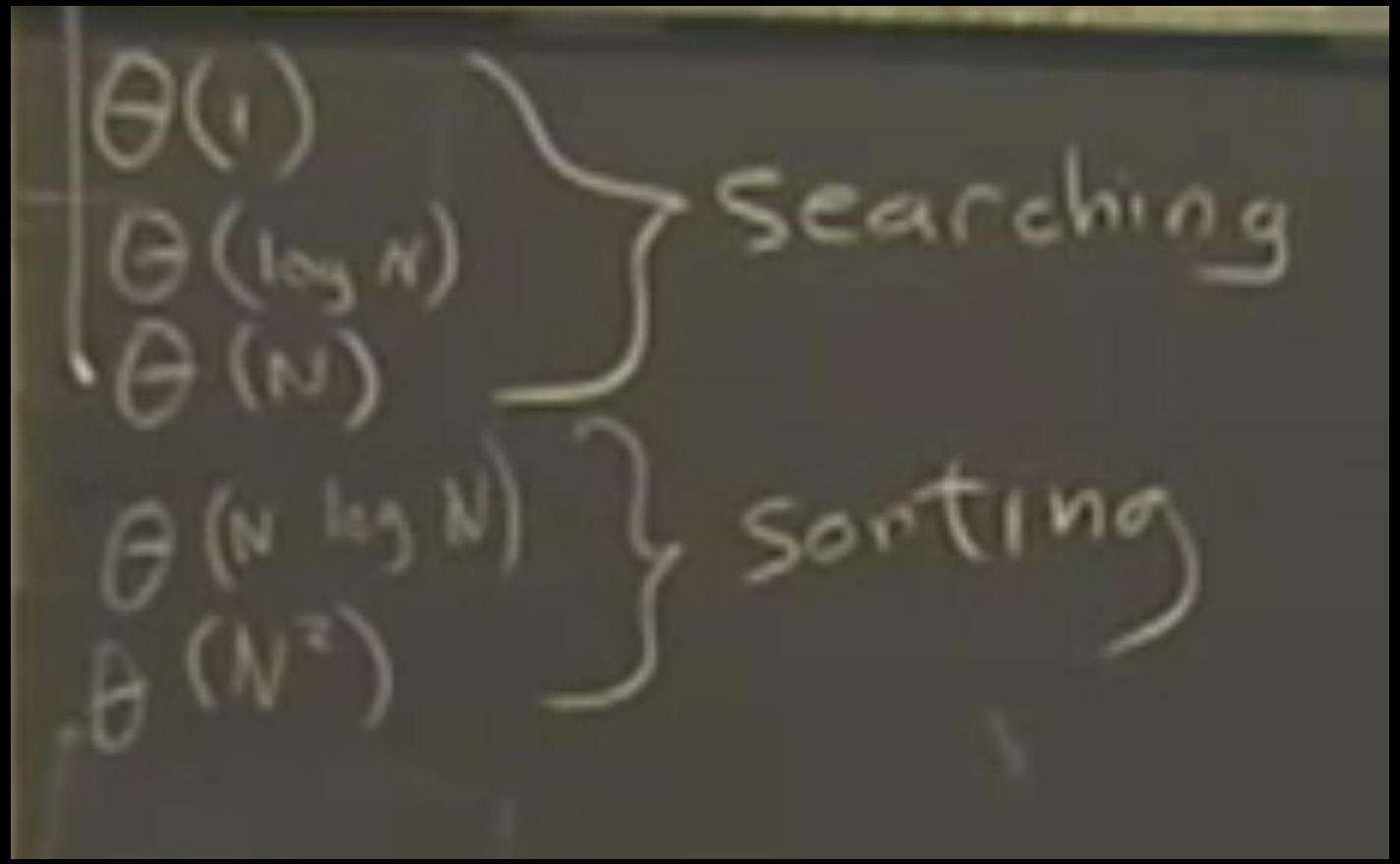
L08 Recursion and Iteration | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

9

L09 Data Abstraction | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina


$\Theta(1)$
$\Theta(\log N)$
$\Theta(N)$
Searching
$\Theta(N \log N)$
$\Theta(N^2)$
Sorting

5

L05 User Interface | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

6

L06 User Interface | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

7

L07 Orders of Growth | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

8

L08 Recursion and Iteration | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

9

L09 Data Abstraction | UC Berkeley CS 61A, Spring 2010

Satyakiran Duggina

“He is one of the **giants** in the history of modern computing. He arguably **invented every aspect** of the **modern computing environment**.”

Brian Harvey

L05 User Interface, SICP

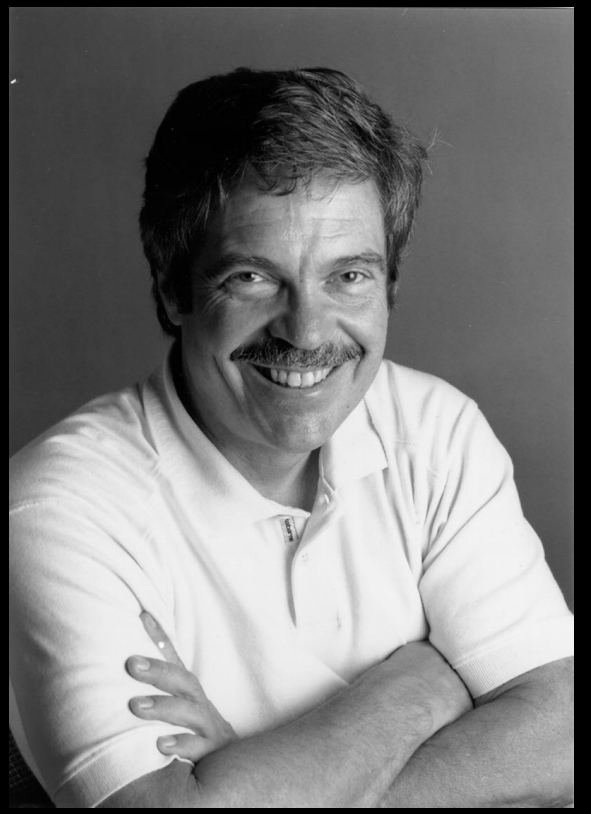
The Inner Game of Tennis

The ultimate guide to the mental side of peak performance

W. Timothy Gallwey

Who is Alan Kay?

- Creator of SmallTalk
- Won the 2003 Turing Award
- One of the father's of OOP
- Worked at Xerox PARC in 1970s (and developed many of the technologies Apple would "steal" for the Lisa and Macintosh)
- Was Chief Scientist at Atari from 1981-84
- Also later worked at Walt Disney and Apple
- "I'm sorry that I long ago coined the term "**objects**" for this topic because it gets many people to focus on the lesser idea. The big idea is "**messaging**"



parc
A Xerox Company

ATARI

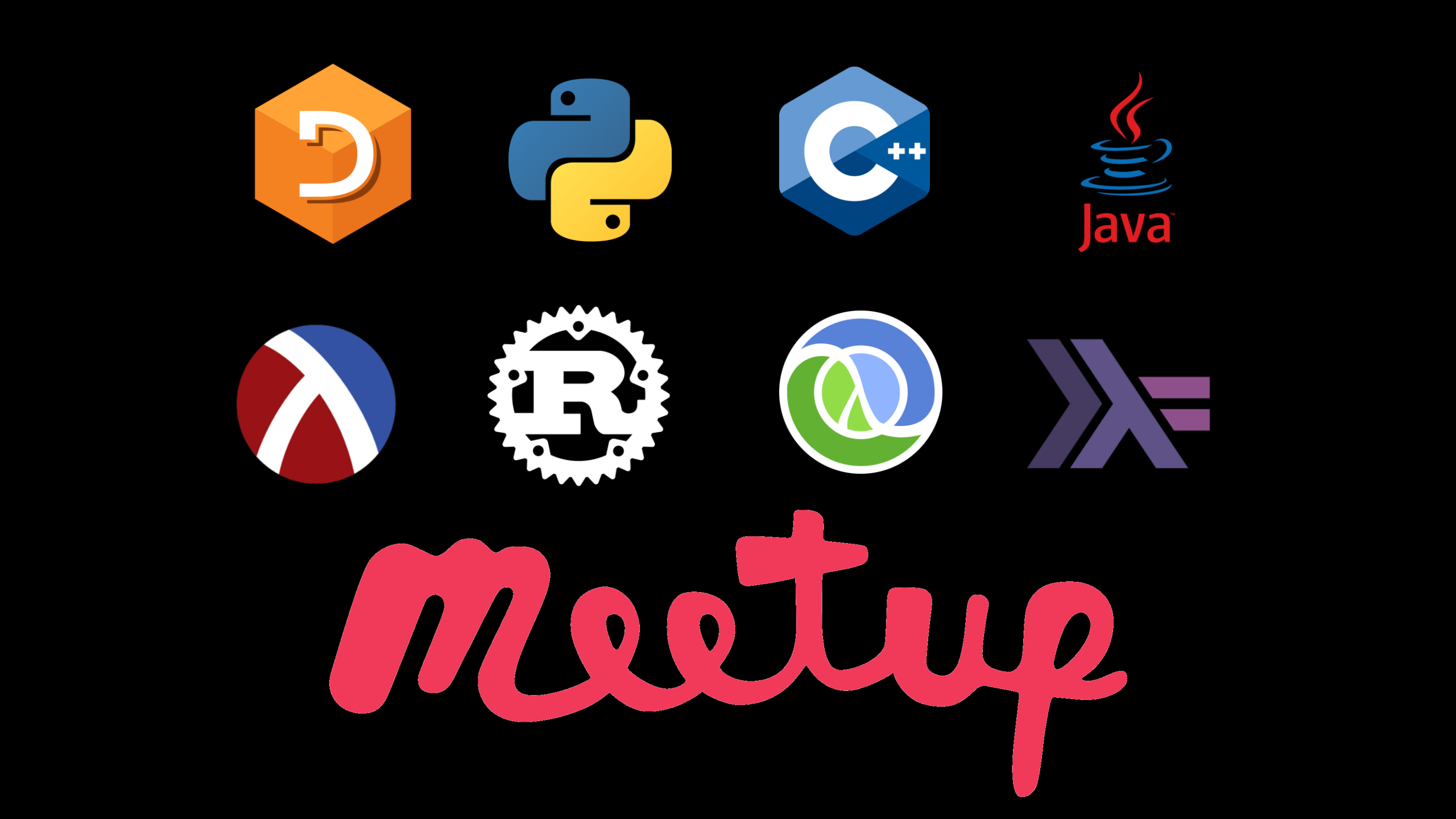
Walt Disney

README.md

SICP-2020

This is the material (code and presentation slide decks) that correspond to the Programming Languages Virtual Meetup course that covered the following:

- **Textbook:** Structure and Iterpretation of Computer Programs
- **MIT Lectures:** YoutTube Playlist | MIT Open CourseWare
- **UC Berkeley Lectures:** YouTube Playlist
- **Meetup Pre-recordings:** YouTube Playlist


Java
meetup