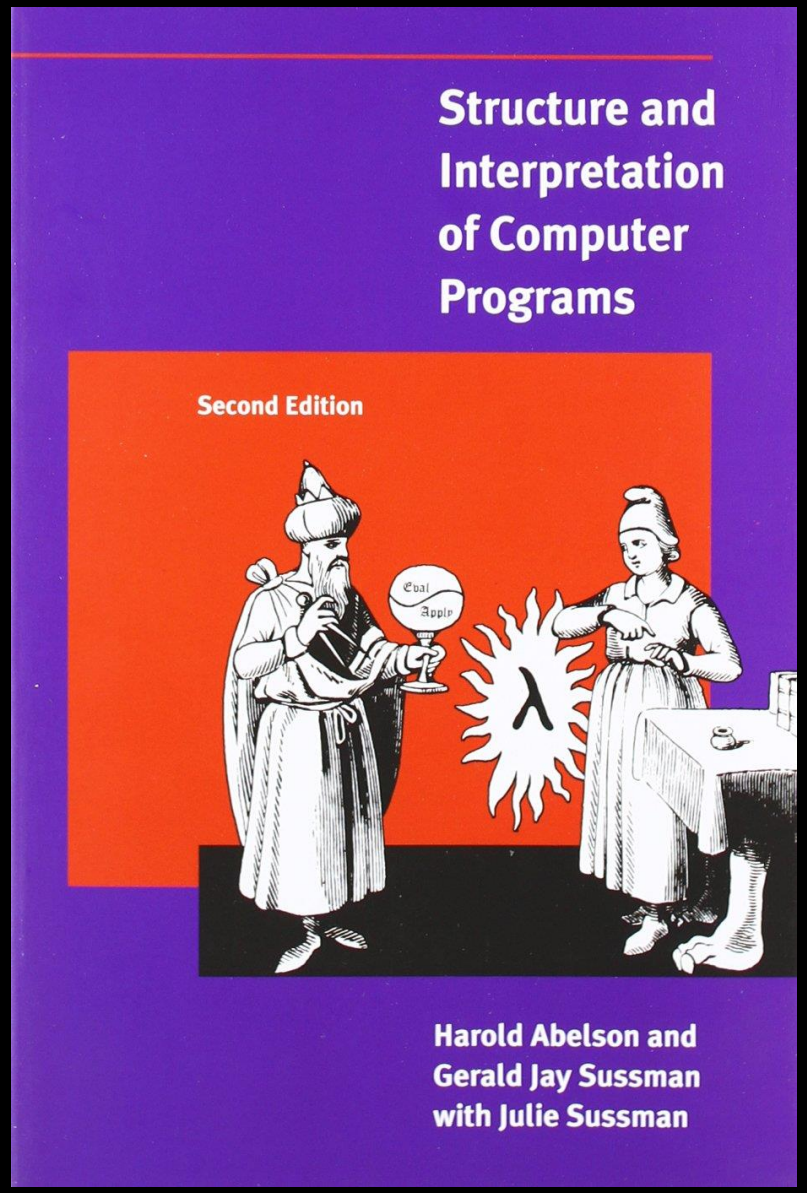
C++

Java

meetup



Structure and Interpretation of Computer Programs

Chaper 1.2

Friendly Environment Policy

Berlin Code of Conduct

Any first time attendees?

That didn’t watch pre-recording of Chapter 1.1?

Goal of PLVM

- Work through books on programming languages together
- Grow knowledge on PLs and PL:
  - principles
  - design
  - implementation
- This ultimately will lead to ability to write **code** that is more:
  - readable & expressive
  - maintainable & scalable
  - beautiful & idiomatic

Format of PLVM

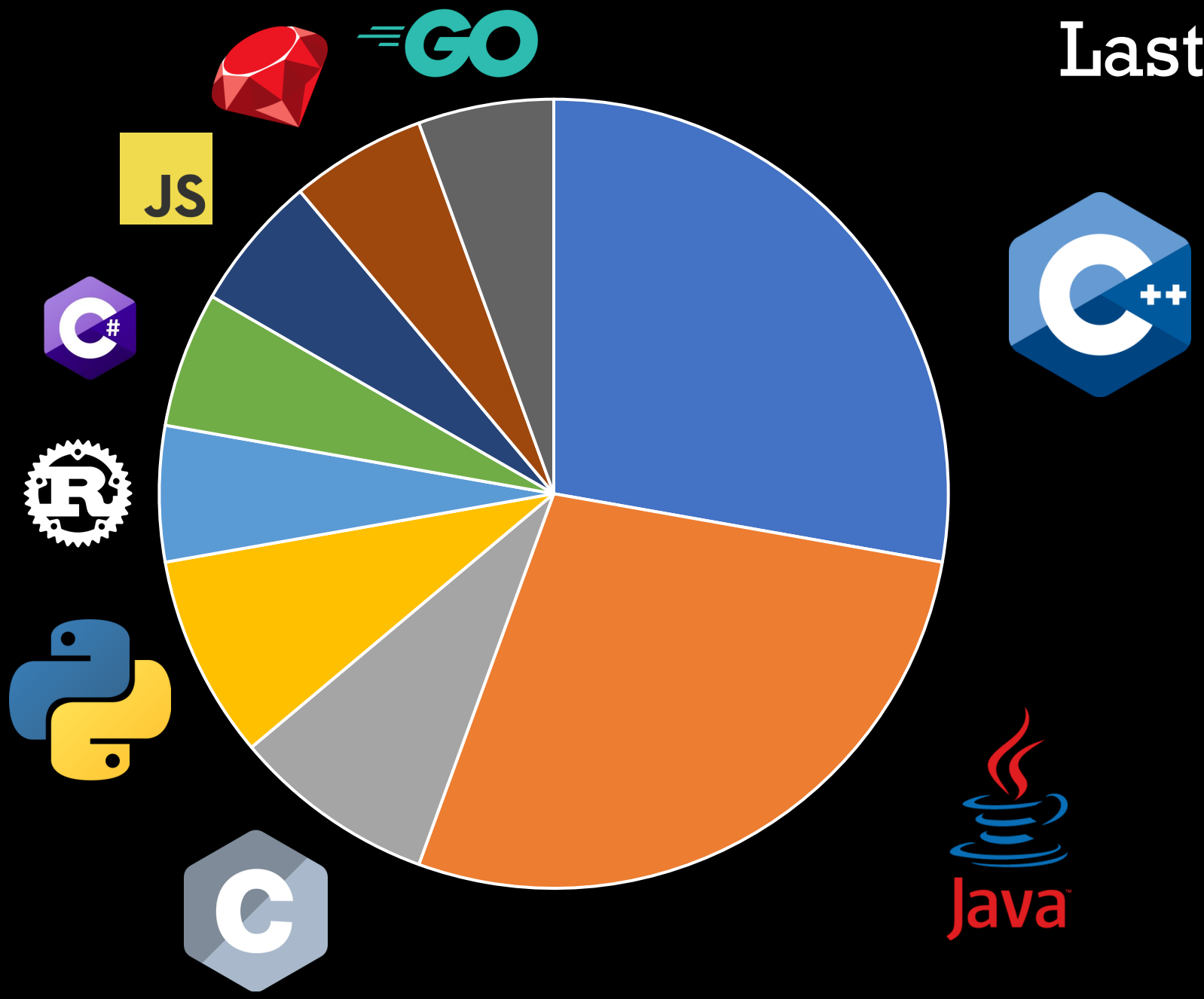
- One hour meeting once a week
- Presentation at beginning
- Discussion afterwards (and during)

- I will pre-record presentation and upload to YouTube for those unable to attend

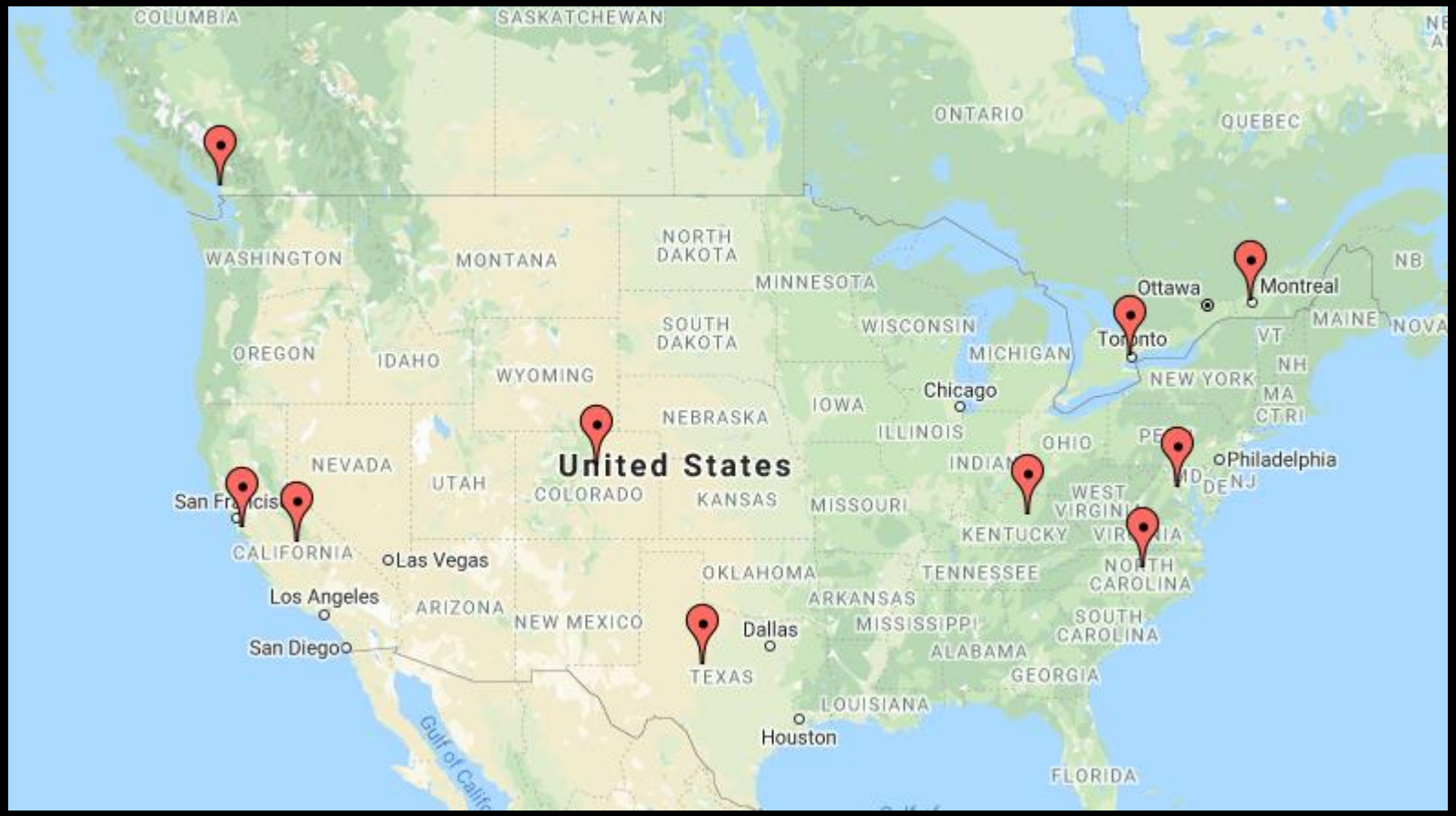
Some things from **last week**

Last week

Last week



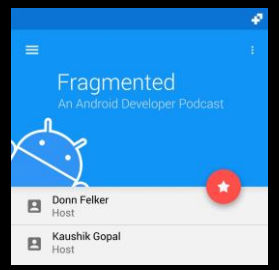
Last week


COLUMBIA
SASKATCHEWAN
ONTARIO
QUEBEC
WASHINGTON
MONTANA
NORTH DAKOTA
MINNESOTA
SOUTH DAKOTA
WISCONSIN
MICHIGAN
OREGON
IDAHO
WYOMING
NEBRASKA
IOWA
ILLINOIS
NEVADA
UTAH
COLORADO
KANSAS
MISSOURI
CALIFORNIA
ARIZONA
NEW MEXICO
OKLAHOMA
ARKANSAS
TEXAS
LOUISIANA
MISSISSIPPI
ALABAMA
GEORGIA
FLORIDA
KENTUCKY
TENNESSEE
OHIO
NEW YORK
VT
NH
MAINE
MA
CT
RI
NJ
DE
MD
NB
Ottawa
Montreal
Toronto
Chicago
Philadelphia
Las Vegas
Los Angeles
San Diego
Dallas
Houston
United States

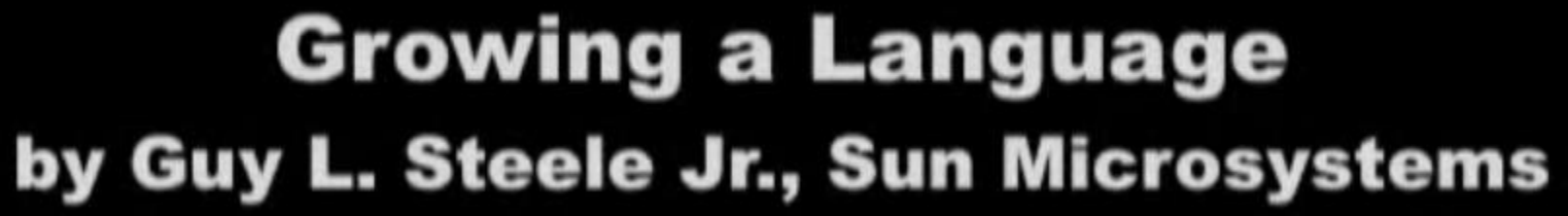
North Sea
United Kingdom
Denmark
Ireland
London
Berlin
Germany
Belgium
Paris
Austria

Mindanao
Basilan Island
Malaysia
Kuala Lumpur
Celebes Sea
Singapore
Jakarta
Java Sea
Indonesia
Banda Sea

The“tech” podcasts I listen to:



Guy Steele gave a great talk (OOPSLA 98 keynote)
"Growing a Language"
https://www.youtube.com/watch?v=_ahvzDzKdB0



Hasan

LISP = lots of irritating stupid parentheses

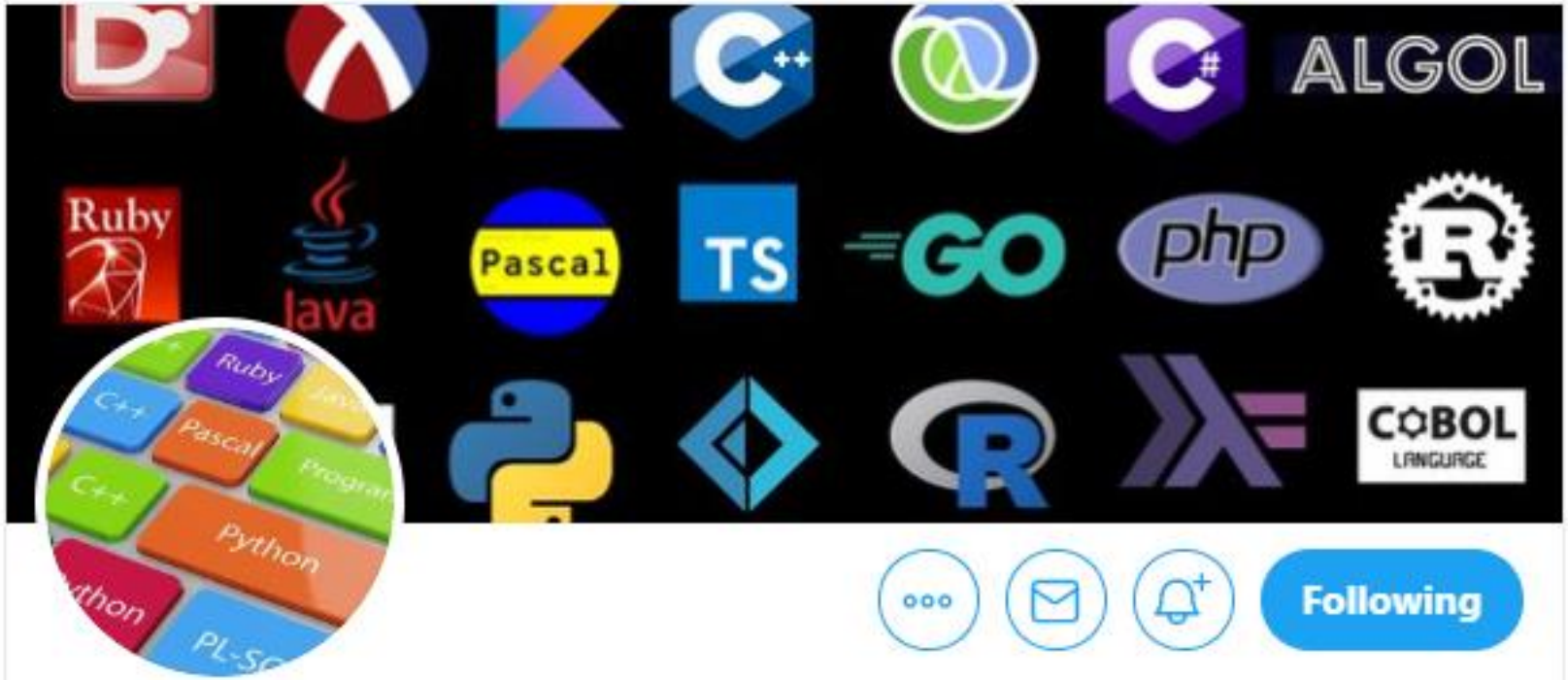
Hasan

LISP = **L**ots of **I**rritating **S**tupid **P**arentheses

DISCORD

Programming Languages Virtual Meetup

1 Tweet



Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

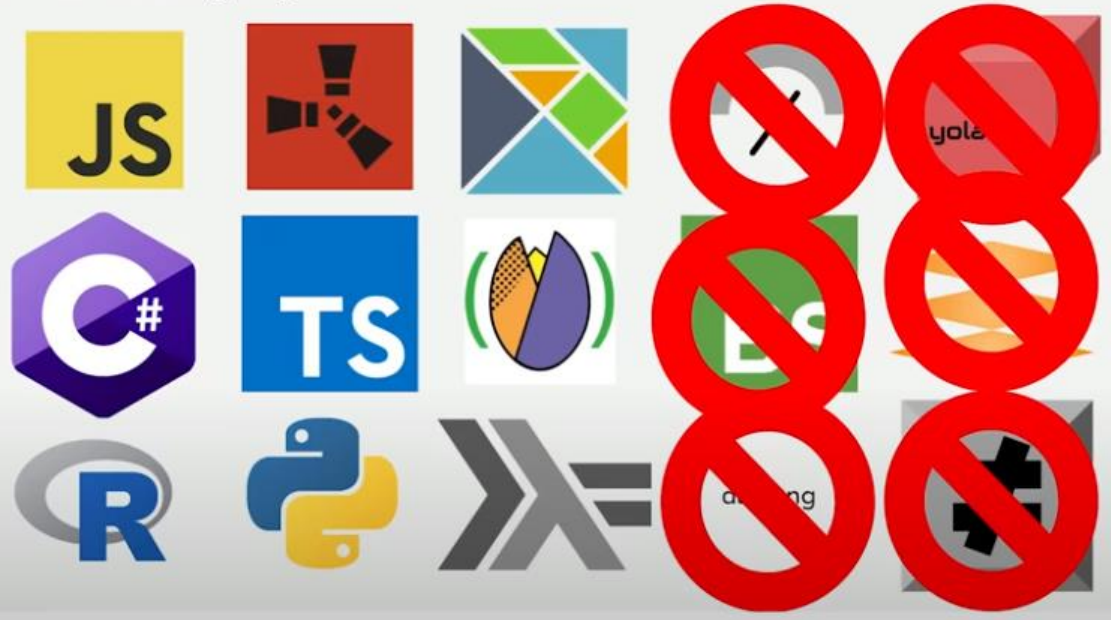
Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

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

Who is Alan. J Perlis?

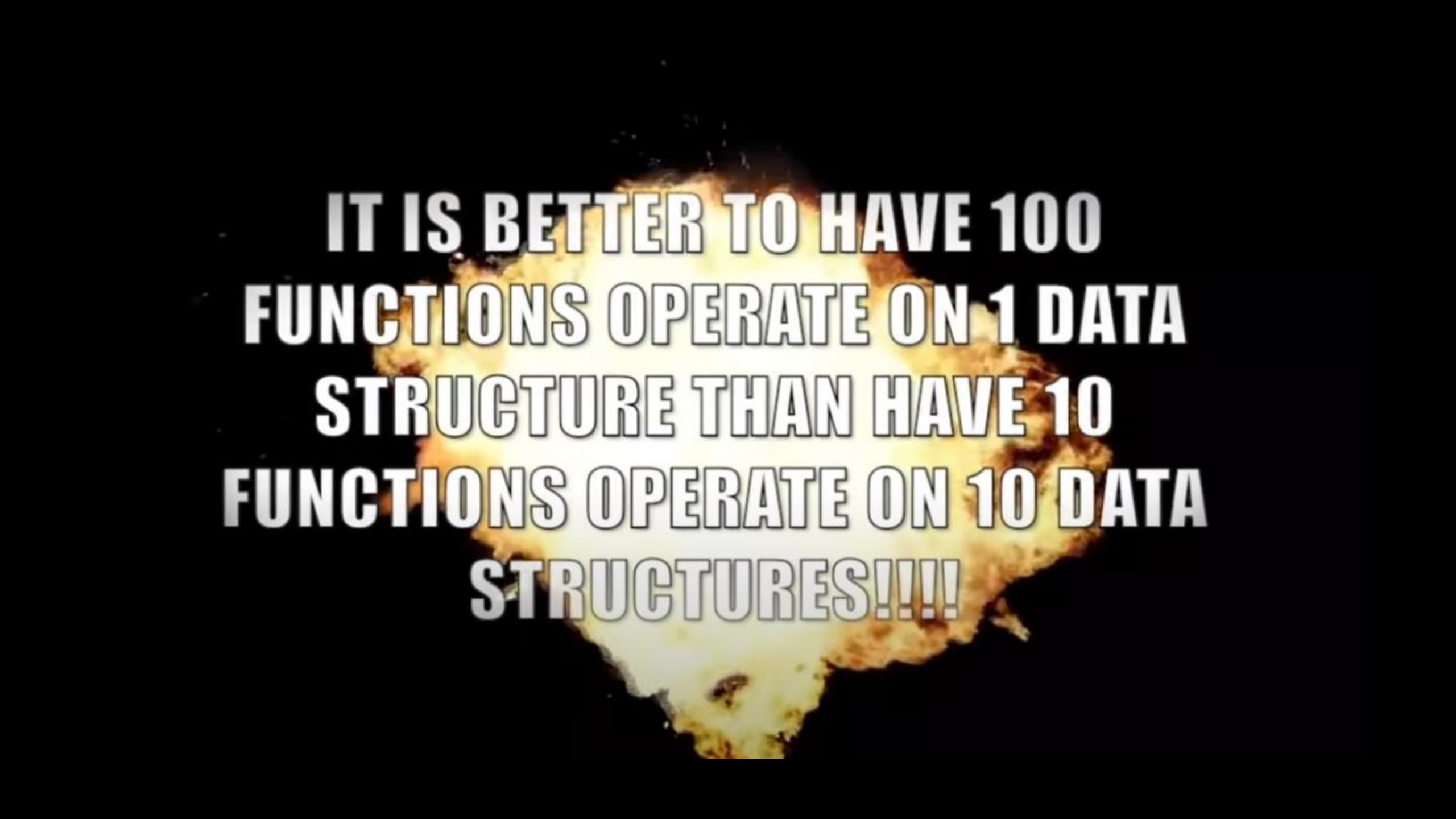
- First recipient of the **Turing Award**
- Member of the team that developed (one of the most influential PLs ever)
- First president of the ACM
- Professor at Purdue, CMU and Yale
- Wrote "*Epigrams on Programming*"
  - "A LISP programmer knows the value of everything, but the cost of nothing."
  - "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."
- *In Praise of APL: A Language for Lyrical Programming*

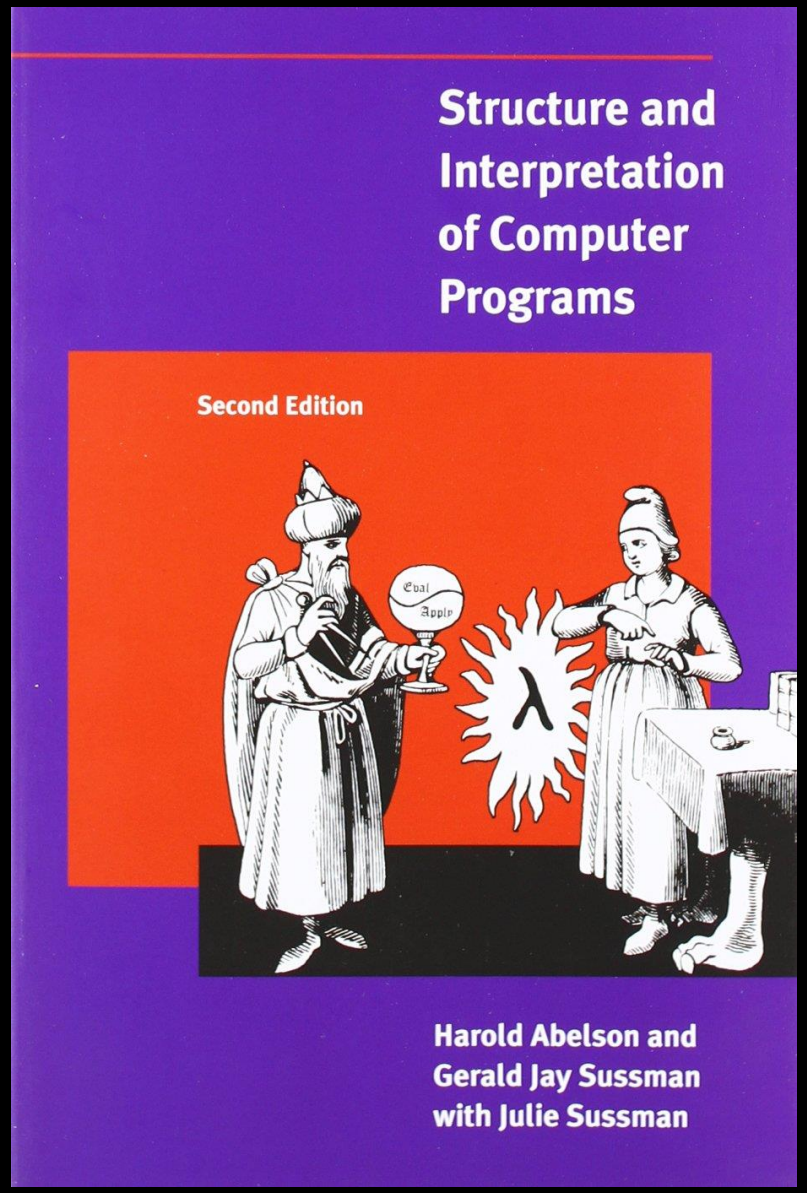
Even more languages



8:22 / 27:36

Every Clojure Talk Ever - Alex Engelberg and Derek Slager


IT IS BETTER TO HAVE 100
FUNCTIONS OPERATE ON 1 DATA
STRUCTURE THAN HAVE 10
FUNCTIONS OPERATE ON 10 DATA
STRUCTURES!!!!



Structure and Interpretation of Computer Programs

Chaper 1.2

1.2 Procedures and the Processes They Generate 40
1.2.1 Linear Recursion and Iteration 41
1.2.2 Tree Recursion 47
1.2.3 Orders of Growth 54
1.2.4 Exponentiation 57
1.2.5 Greatest Common Divisors 62
1.2.6 Example: Testing for Primality 65

Chapter 1.2 by Example

```
; Recursive solution

(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
; Iterative

(define (fact n)
  (define (fact-iter acc n)
    (if (= n 0)
        acc
        (fact-iter (* n acc) (- n 1))))
  (fact-iter 1 n))
```

```
; Algorithms + Threading

(require threading)

(define (fact n)
  (~> (range 1 (+ n 1))
      (foldl * 1 _)))
```

```
auto fact(int n) -> int {
    std::vector<int> v(n);
    std::iota(v.begin(), v.end(), 1);
    return std::accumulate(v.cbegin(), v.cend(), 1,
        [](auto acc, auto n) { return acc * n; });
}
```

```java
public static int fact(int val) {
    return IntStream.rangeClosed(1, val)
                    .reduce(1, (acc, n) -> acc * n);
}
```

```rust
fn fact(n: i32) -> i32 {
    return (1..n+1).fold(1, |acc, n| acc * n);
}
```

```
; Recursive

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

```
; Iterative

(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

```cpp
auto fib(int n) -> int {
    std::vector<int> v(n-2);
    return std::accumulate(v.cbegin(), v.cend(),
        std::pair{1, 1},
        [](auto pair, auto unused) {
            auto [a, b] = pair;
            return std::pair{b, a + b};
        }).second;
}
```

```rust
fn fib(n: i32) -> i32 {
    return (1..n-1).fold((1, 1), |(a, b), _unused| (b, a + b)).1;
}
```

Exercise 1.11: A function f is defined by the rule that

$$f(n) = \begin{cases} n & \text{if} \quad n < 3, \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{if} \quad n \geq 3. \end{cases}$$

Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

```
; Exercise 1.11 (page 53)

; Recursive solution

(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1))
         (* 2 (f (- n 2)))
         (* 3 (f (- n 3))))))

; > (map f (range 10))
; '(0 1 2 4 11 25 59 142 335 796)
```

```
; Iterative solution

(define (f n)
  (f-iter 0 1 2 n))

(define (f-iter a b c count)
  (cond ((= 0 count) a)
        (else (f-iter b
                      c
                      (+ c
                         (* 2 b)
                         (* 3 a))
                      (- count 1)))))

; > (map f (range 10))
; '(0 1 2 4 11 25 59 142 335 796)
```

Exercise 1.12: The following pattern of numbers is called *Pascal's triangle.*

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
      .  .  .
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.[35] Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

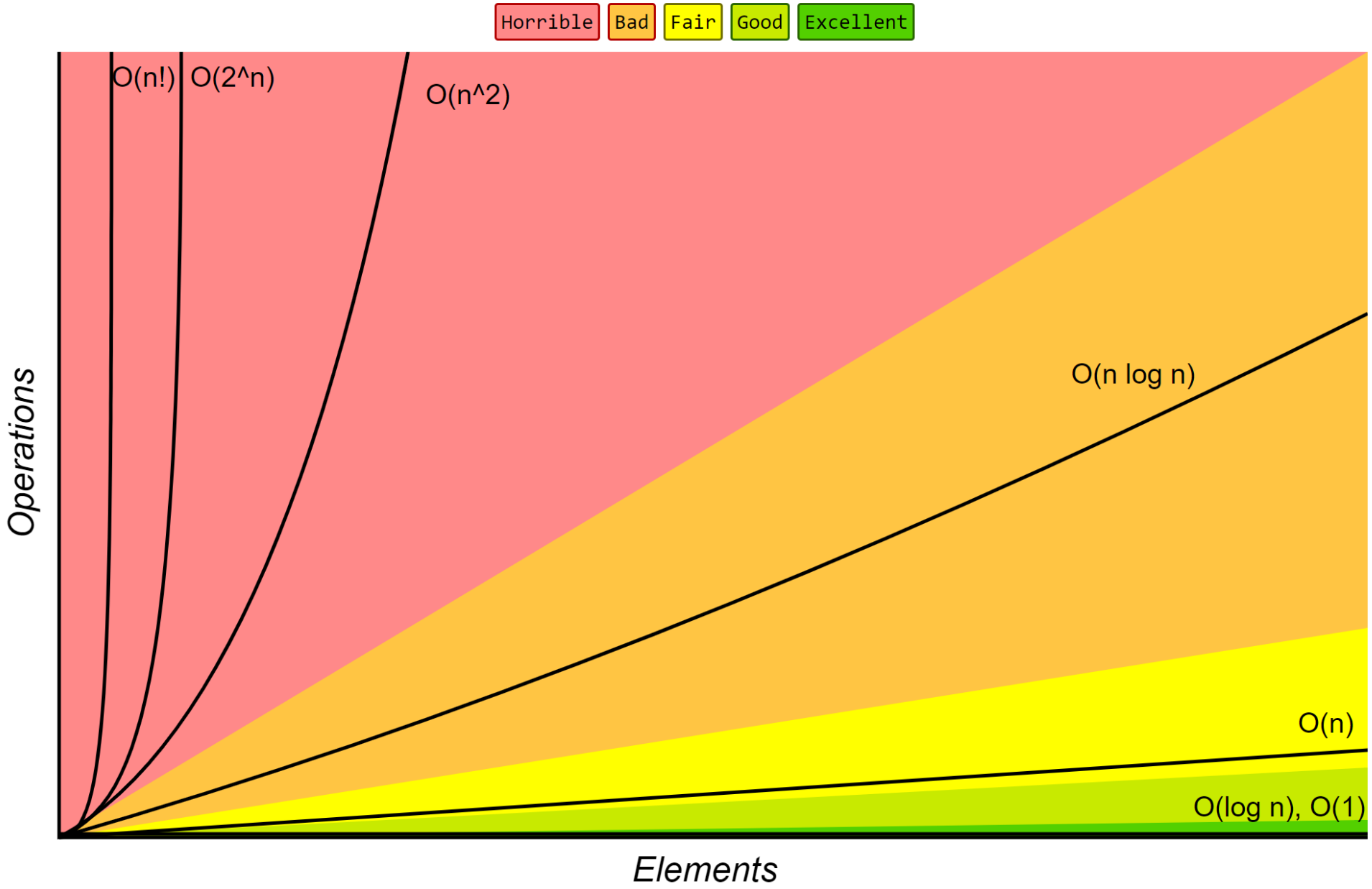
```
; Exercise 1.12

(require threading)
(require algorithms)

(define (pascals-
  (if (= 1 n)
      '(1)
      (pt-iter '(

(define (pt-iter
  (if (= 0 n)
      (reverse ac
      (pt-iter (c

(define (pt-next-row row)
  (~> row
      (sliding 2)
      (map (λ (x) (foldl + 0 x)) _)
      (append '(1) _ '(1))))
```

```
(define (pt-next-row row)
  (~> row
      (sliding 2)
      (map (λ (x) (foldl + 0 x)) _)
      (append '(1) _ '(1))))
```

```
; > (pascals-triangle 10)
; '((1)
;                    )
;                    1)
;                    3 1)
;                    6 4 1)
;                    10 10 5 1)
;                    15 20 15 6 1)
;                    21 35 35 21 7 1)
;   (1 8 28 56 70 56 28 8 1)
;   (1 9 36 84 126 126 84 36 9 1))
```

1.2 Procedures and the Processes They Generate 40
✓ 1.2.1 Linear Recursion and Iteration 41
✓ 1.2.2 Tree Recursion 47
→ 1.2.3 Orders of Growth 54
1.2.4 Exponentiation 57
1.2.5 Greatest Common Divisors 62
1.2.6 Example: Testing for Primality 65

Big-O Complexity Chart



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

1.2 Procedures and the Processes They Generate 40
✓ 1.2.1 Linear Recursion and Iteration 41
✓ 1.2.2 Tree Recursion 47
✓ 1.2.3 Orders of Growth 54
→ 1.2.4 Exponentiation 57
1.2.5 Greatest Common Divisors 62
1.2.6 Example: Testing for Primality 65

```
; Recursive

(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

```
; Iterative

(define (expt b n)
  (define (expt-iter acc base count)
    (if (= count 0)
        acc
        (expt-iter (* acc base) base (- count 1))))
  (expt-iter 1 b n))
```

```
; Algorithms + Threading

(require threading)
(require algorithms)

(define (expt b n)
  (~> (repeat n b)
      (foldl * 1 _)))
```

```
; Fast exponentiation

; Recursive

(define (square x) (* x x))

(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

Exercise 1.16: Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast-expt`. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an *invariant quantity* that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

```
; Iterative

(define (fast-expt b n)
  (define (iter squares extra N)
    (cond ((= N 1) (* squares extra))
          ((even? N) (iter (square squares) extra (/ N 2)))
          (else (iter squares (* extra squares) (- N 1)))))
  (iter b 1 n))
```

Exercise 1.17: The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the `expt` procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in `b`. Now suppose we include, together with addition, operations `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to `fast-expt` that uses a logarithmic number of steps.

```
; Exercise 1.17

; Recursive

; corrected code
(define (multiply a b)
  (cond ((= a 0) 0)
        ((even? a) (multiply (/ a 2) (double b)))
        (else (+ b (multiply (- a 1) b)))))
```

Exercise 1.18: Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.[40]

```
; Exercise 1.18

; this is totally broken
(define (multiply a b)
  (define (iter A B acc)
    (cond ((= B 0) acc)
          ((even? B) (iter A (/ B 2) (double A)))
          (else (iter A (- B 1) (+ acc A)))))
  (iter a b 0))
```

```scheme
; corrected code (after looking at http://community.schemewiki.org/?sicp-ex-1.18)
(define (multiply a b)
  (define (iter A B acc)
    (cond ((= B 0) acc)
          ((even? B) (iter (double A) (/ B 2) acc)) ; very interesting that
          (else (iter A (- B 1) (+ acc A)))))         acc isn't modified in this iteration
  (iter a b 0))
```

1.2 Procedures and the Processes They Generate 40
✓ 1.2.1 Linear Recursion and Iteration 41
✓ 1.2.2 Tree Recursion 47
✓ 1.2.3 Orders of Growth 54
✓ 1.2.4 Exponentiation 57
→ 1.2.5 Greatest Common Divisors 62
1.2.6 Example: Testing for Primality 65

```
; GCD

(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

```
; Exercise 1.22 (page 70/71)

(define (smallest-divisor n)
  (define (iter i)
    (cond ((= (remainder n i) 0) i)
          ((> (* i i) n) n)
          (else (iter (+ i 1)))))
  (iter 2))

(define (prime? n)
  (= (smallest-divisor n) n))

(define (timed-prime-test n)
  (start-prime-test n (current-milliseconds)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (current-milliseconds) start-time) n)
      (display "")))

(define (report-prime elapsed-time n)
  (display n)
  (display " *** ")
  (display elapsed-time)
  (newline))

; > (timed-prime-test 10000000000037)
; 10000000000037 *** 116
```

```
; As of 2008, computers have become too fast to appreciate the time
; required to test the primality of such small numbers.
; To get meaningful results, we should perform the test with numbers
; greater by, say, a factor 1e6.
(newline)
(search-for-primes 1000000000 1000000021)       ; 1e9
(search-for-primes 10000000000 10000000061)     ; 1e10
(search-for-primes 100000000000 100000000057)   ; 1e11
(search-for-primes 1000000000000 1000000000063) ; 1e12
```

```
(define (search-for-primes lower upper)
  (define (iter n)
    (cond ((<= n upper) (timed-prime-test n) (iter (+ n 2)))))
  (iter (if (odd? lower) lower (+ lower 1))))
```

Exercise 1.27: Demonstrate that the Carmichael numbers listed in Footnote 1.47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

[47] Numbers that fool the Fermat test are called *Carmichael numbers*, and little is known about them other than that they are extremely rare. There are 255 Carmichael numbers below 100,000,000. The smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

> **Fermat's Little Theorem:** If n is a prime number and a is any positive integer less than n, then a raised to the nth power is congruent to a modulo n.

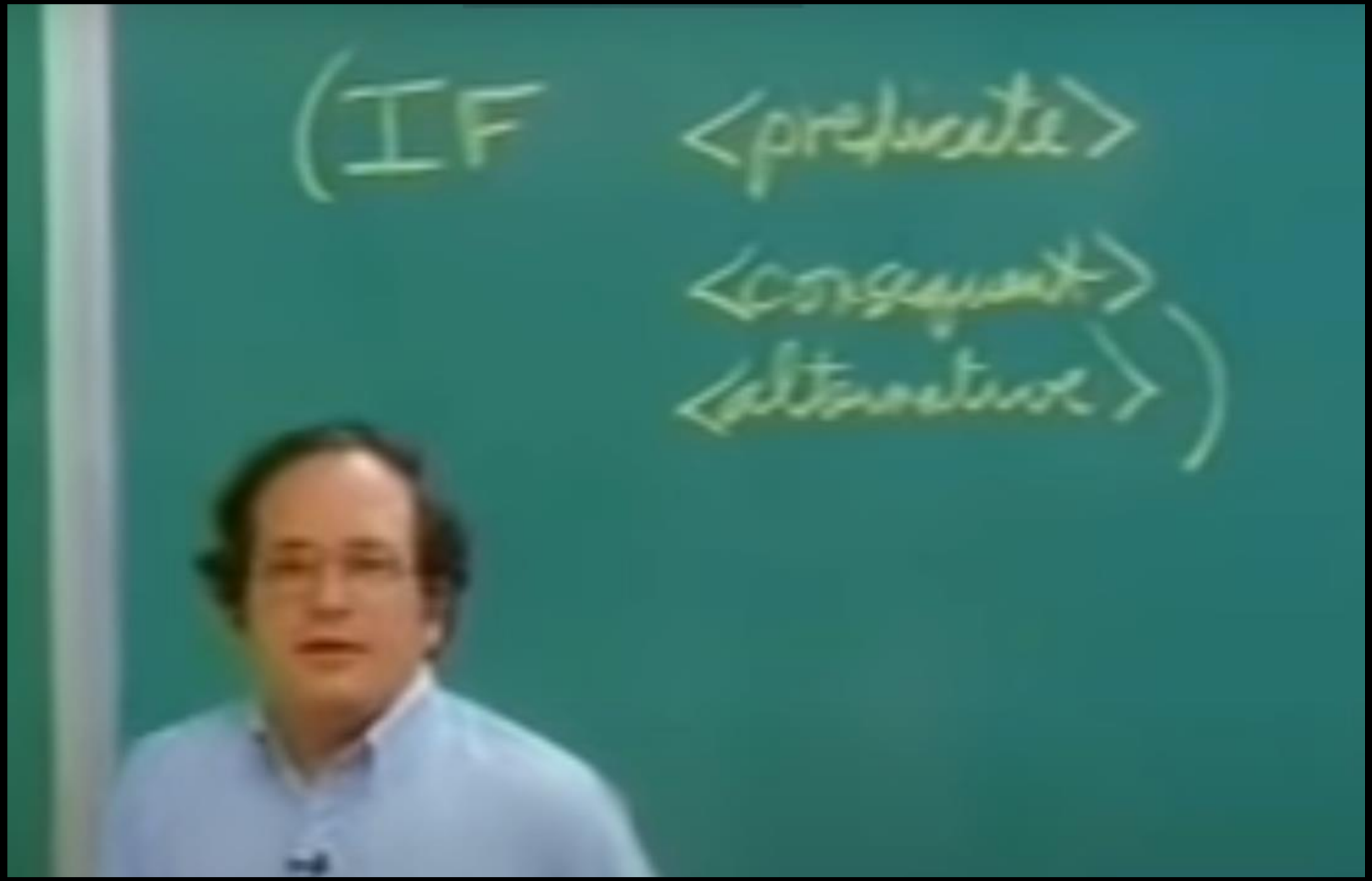
(Two numbers are said to be *congruent modulo n* if they both have the same remainder when divided by n. The remainder of a number a when divided by n is also referred to as the *remainder of a modulo n*, or simply as *a modulo n*.)

If n is not prime, then, in general, most of the numbers $a < n$ will not satisfy the above relation. This leads to the following algorithm for testing primality: Given a number n, pick a random number $a < n$ and compute the remainder of a^n modulo n. If the result is not equal to a, then n is certainly not prime. If it is a, then chances are good that n is prime.

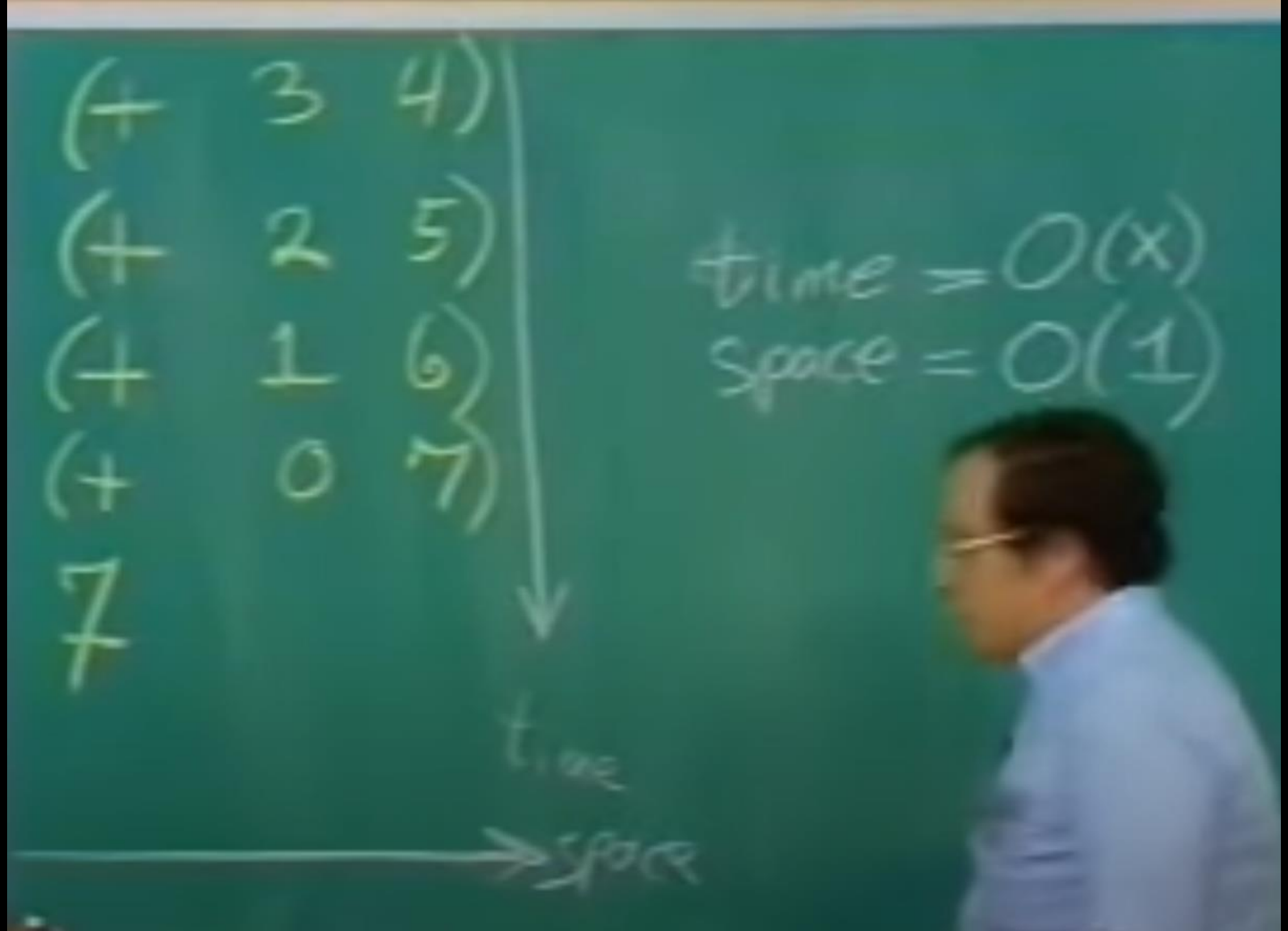
```
; Exercise 1.27 (page 73)

; Copied from text

; prime? from above

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (square (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base (expmod base (- exp 1) m))
          m))))

(define (square x) (* x x))
```

```
; Modified

; Don't want a random number (pass it in)
(define (fermat-test n a)
  (= (expmod a n n) a))

; My code

(define (is-carmichael-number? n)
  (define (fermat-test-all n)
    (~> (range 1 n)
        (andmap (λ (a) (fermat-test n a)) _)))
  (and (fermat-test-all n) (not (prime? n))))

; > (map is-carmichael-number? '(561 1105 1729 2465 2821 6601))
; '(#t #t #t #t #t #t)
```

Structure & Interpretation
of Computer Programs
Harold
Abelson
Gerald Jay
Sussman
EVAL
APPLY


(IF
<predicate>
<consequent>
<alternative>
)

“It's important, by the way, to get **names** for the parts of things, or the parts of expressions. One of the things that every sorcerer will tell you is if you have the **name** of a spirit, you have power over it. So you have to learn these **names** so that we can discuss these things.”


(+ 3 4)
(+ 2 5)
(+ 1 6)
(+ 0 7)
7
time
space
time = O(x)
space = O(1)

```
(+ 3 4)
(+ (+ 2 4))
(+ (+ (+ 1 4)))
(+ (+ (+ (+ 0 4))))
(+ (+ (+ 4)))
(+ (+ 5))
(+ 6)
7
```

time = O(k)

space = O(k)

C++

Java

meetup