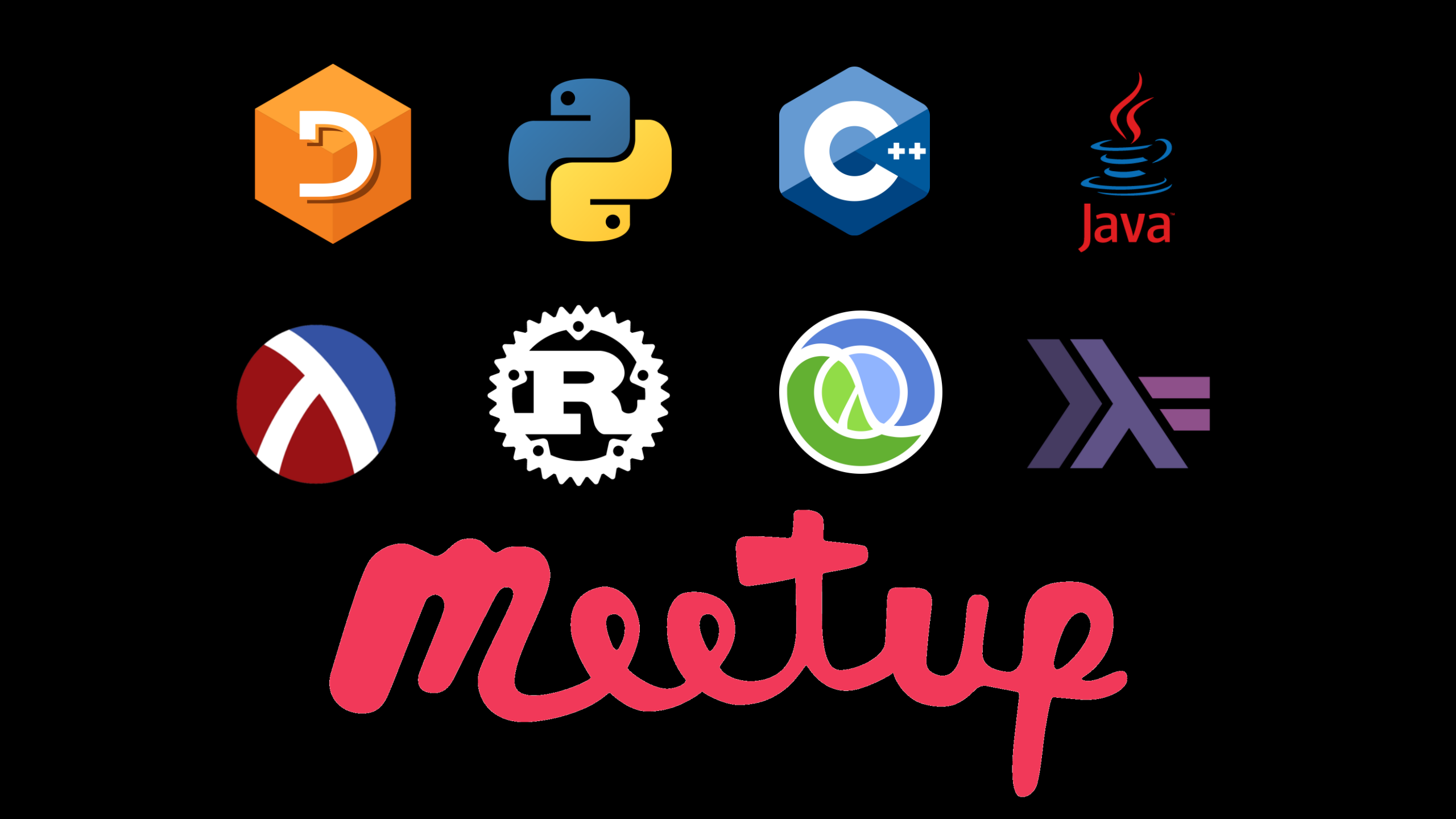
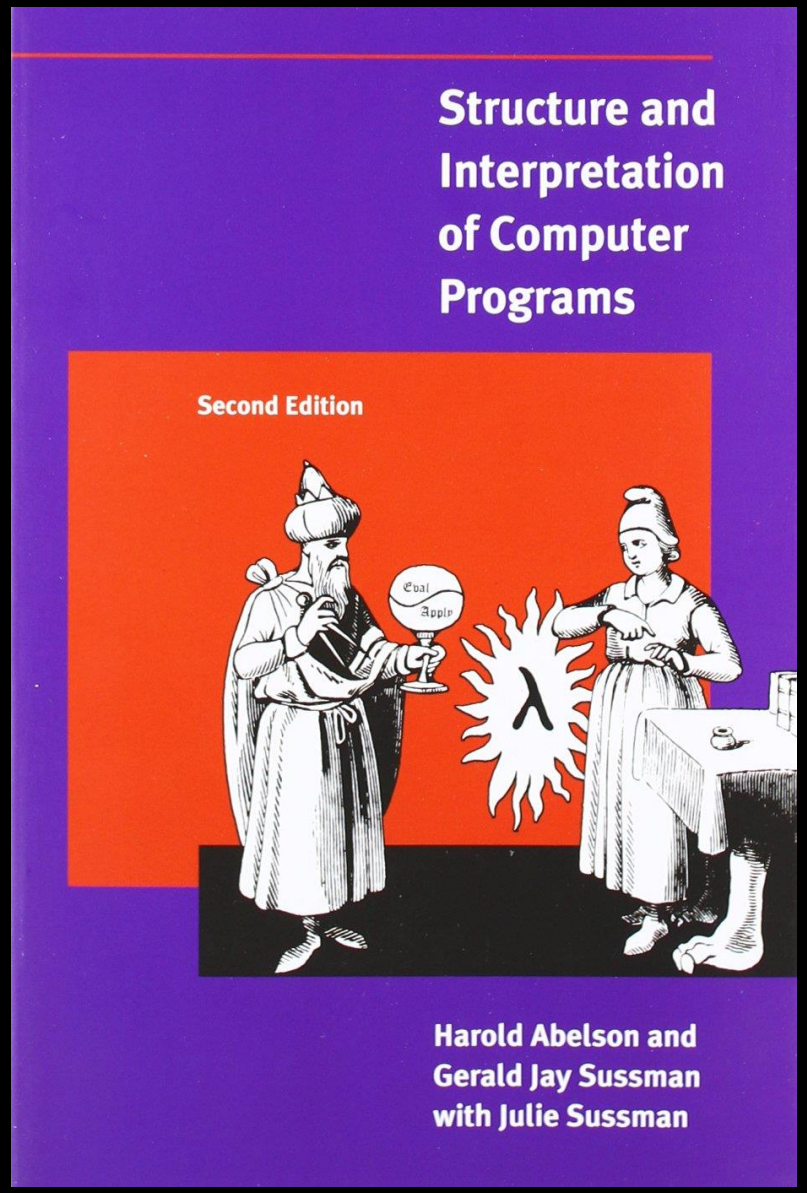

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 1.3

Friendly Environment Policy

Berlin Code of Conduct

Some things from **last week**

Last week

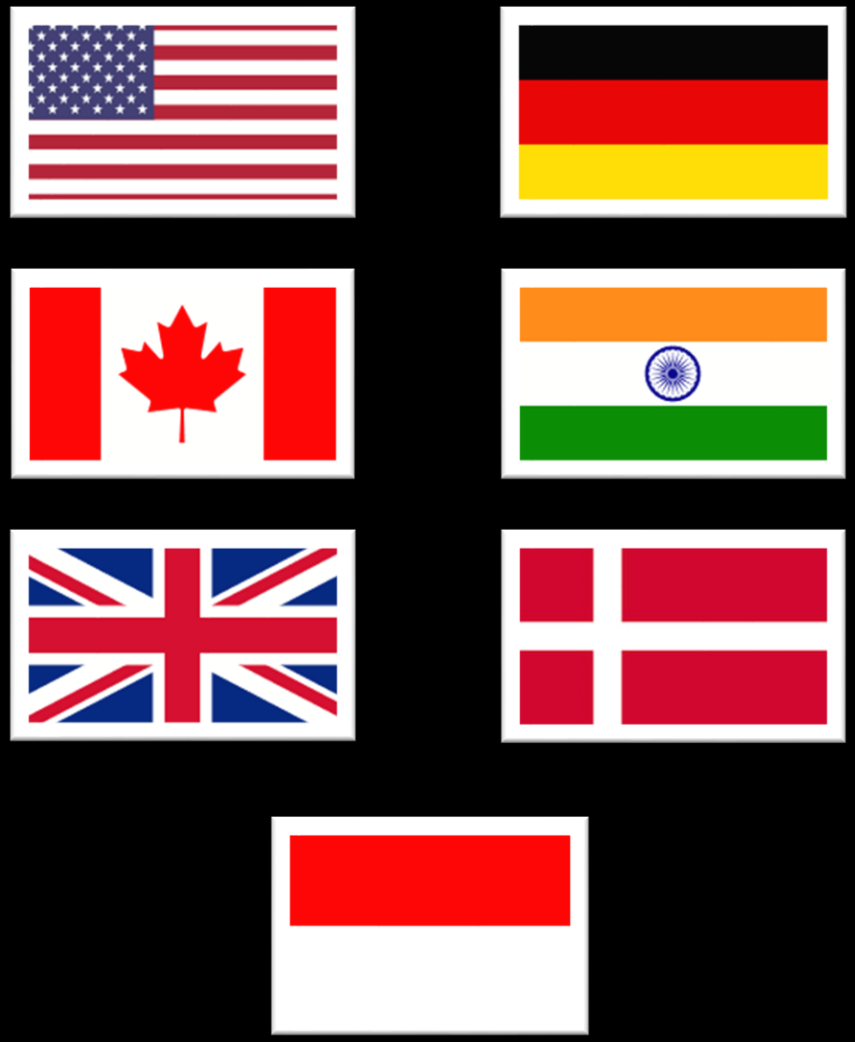









In aggregate




DISCORD

Programming Languages Virtual Meetup

1 Tweet

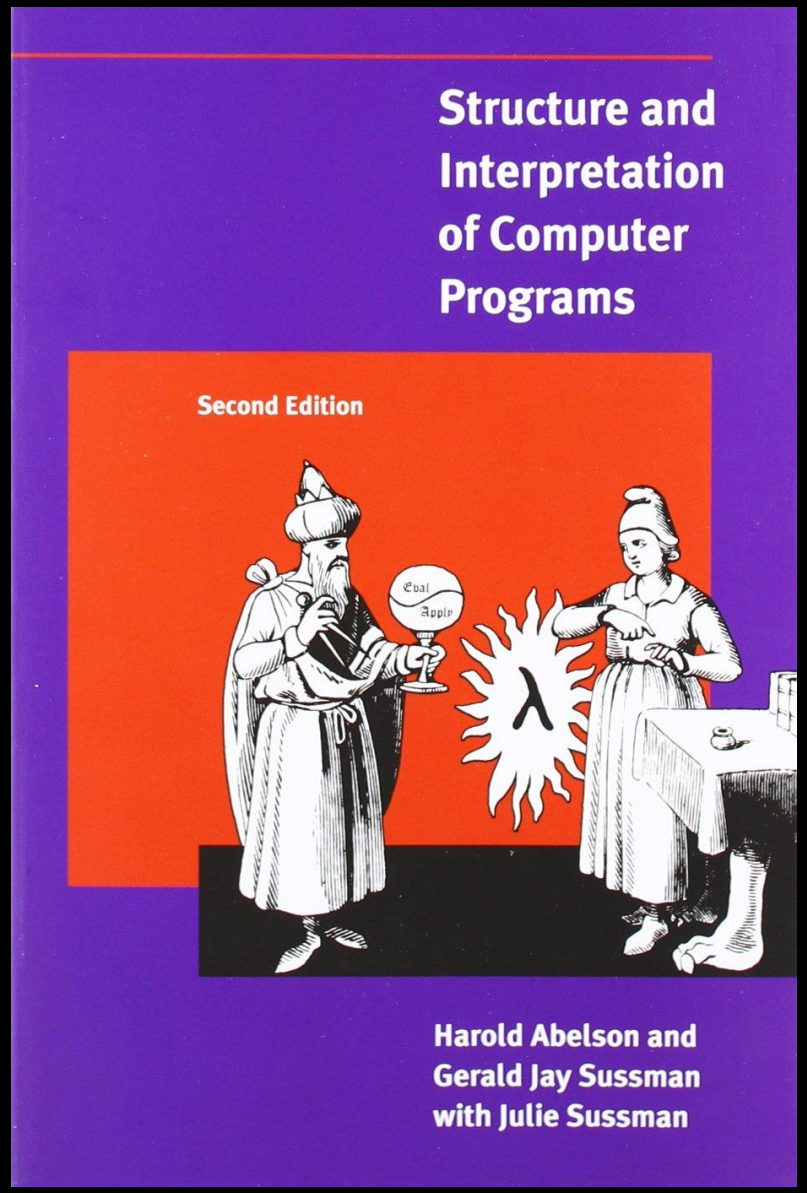
Following

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



Structure and Interpretation of Computer Programs

Chaper 1.3

Chapter 1.3 by Example

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
```

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

```
(define (inc n) (+ n 1))
(define (sum-cubes a b)
  (sum cube a inc b))

(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b))
```

Exercise 1.30: The sum procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if ⟨??⟩
        ⟨??⟩
        (iter ⟨??⟩ ⟨??⟩)))
  (iter ⟨??⟩ ⟨??⟩))
```

```scheme
;; Exercise 1.30 (page 80)

(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))

(define (inc n) (+ n 1))
(define (identity n) n)

;; > (sum identity 0 inc 10)
;; 55
```

Exercise 1.31:

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures.[51] Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`.

```
;; Exercise 1.31 a) (page 80/81)

(define (product term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1))

;; > (product identity 1 inc 10)
;; 3628800
```

Exercise 1.32:

a. Show that `sum` and `product` (Exercise 1.31) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

```
;; Exercise 1.32 a) (page 81/82)

(define (accumulate combiner init term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (term a)))))
  (iter a init))

(define (product term a next b) (accumulate * 1 term a next b))
(define (sum     term a next b) (accumulate + 0 term a next b))
```

π

```
(~>> (first-n-odds 10000)
     (chunks-of 2)
     (map product)
     (map (λ (x) (/ 8.0 x)))
     (sum))
```

```clojure
;; Original

(~>> (range 1000)
     (map (λ (x) (+ 1 (* 2 x))))
     (chunks-of _ 2)
     (map (λ (x) (foldl * 1 x)))
     (map (λ (x) (/ 1.0 x)))
     (foldl + 0)
     (* 8))

;; 3.1405926538397897
```

```
;; Cleaned up

(require algorithms)

(define (first-n-odds n)
  (~>> (range n)
       (map (λ (x) (+ 1 (* 2 x))))))

(~>> (first-n-odds 10000)
     (chunks-of _ 2)
     (map product)
     (map (λ (x) (/ 8.0 x)))
     (sum))

;; 3.1414926535900367
```

```
import Data.List.Split (chunksOf)

let firstNOdds n = map (((-)1) . (*2)) $ [1..n]

let pi = sum
       . map ((8/) . product)
       . chunksOf 2
       . firstNOdds
```

Exercise 1.29: Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n),$$

where $h = (b - a)/n$, for some even integer n, and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f, a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate cube between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

```
;; Exercise 1.29 (page 80)

(require threading)
(require algorithms)

(define (cube x) (* x x x))

(define (simpsons-integral f a b n)
  (let* ((h (/ (+ b a) n))
         (k (- (/ n 2) 1))
         (coefficients (flatten
                        (append '(1)
                                (make-list k '(4 2))
                                '(4 1)))))
    (~>> (range a (+ b h) h)
         (map f)
         (zip-with * coefficients)
         (sum)
         (* (/ h 3.0)))))
```

```
(define (plus4 x) (+ x 4))
```

is equivalent to

```
(define plus4 (lambda (x) (+ x 4)))
```

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

Exercise 1.34: Suppose we define the procedure

```
(define (f g) (g 2))
```

Then we have

```
(f square)
4
(f (lambda (z) (* z (+ z 1))))
6
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

```
;; Exercise 1.34 (page 88)

(define (f proc) (proc 2))

(f f)
(f 2)
(2 2) ; <- 2 is not a procedure
```

Finding fixed points of functions

A number x is called a *fixed point* of a function f if x satisfies the equation $f(x) = x$. For some functions f we can locate a fixed point by beginning with an initial guess and applying f repeatedly,

$$f(x), \quad f(f(x)), \quad f(f(f(x))), \quad \ldots,$$

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Exercise 1.35: Show that the golden ratio φ (Section 1.2.2) is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute φ by means of the `fixed-point` procedure.

```
;; Exercise 1.35 (page 94)

(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

;; > (fixed-point (λ (x) (+ 1 (/ 1 x))) 1.0)
;; 1.6180327868852458

;; Golden Ratio (from Google) = 1.61803398875
```

Exercise 1.36: Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log(1000)/\log(x)$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log(1) = 0$.)

```
;; Exercise 1.36 (page 94)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (newline)
      (display next)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

```
;; > (fixed-point (λ (x) (+ 1 (/ 1 x))) 1.0)

;; 2.0
;; 1.5
;; 1.6666666666666665
;; 1.6
;; 1.625
;; 1.6153846153846154
;; 1.619047619047619
;; 1.6176470588235294
;; 1.6181818181818182
;; 1.6179775280898876
;; 1.6180555555555556
;; 1.6180257510729614
;; 1.6180371352785146
;; 1.6180327868852458
```

```
;; > (fixed-point (λ (x) (/ (log 1000) (log x))) 2.0)

;; 9.965784284662087
;; 3.004472209841214
;; 6.279195757507157
;; 3.759850702401539
;; 5.215843784925895
;; 4.182207192401397
;; 4.8277650983445906
;; 4.387593384662677
;; 4.671250085763899
;; 4.481403616895052
;; 4.6053657460929
;; 4.5230849678718865
;; 4.577114682047341
;; 4.541382480151454
;; 4.564903245230833
;; 4.549372679303342
;; 4.559606491913287
;; 4.552853875788271
;; 4.557305529748263
;; 4.554369064436181
;; 4.556305311532999
;; 4.555028263573554
;; 4.555870396702851
;; 4.555315001192079
;; 4.5556812635433275
;; 4.555439715736846
;; 4.555599009998291
;; 4.555493957531389
;; 4.555563237292884
;; 4.555517548417651
;; 4.555547679306398
;; 4.555527808516254
;; 4.555540912917957
;; 4.555532270803653
```

Exercise 1.37:

a. An infinite *continued fraction* is an expression of the form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \ldots}}}.$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\varphi$, where φ is the golden ratio (described in Section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called *k-term finite continued fraction*—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ddots + \cfrac{N_k}{D_k}}}.$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\varphi$ using

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

b. If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

```
;; Exercise 1.37 a) (page 94-96)

; Note: this recursive solution on works when n and d ignore k :p
(define (cont-frac n d k)
  (if (= k 1)
      (/ (n k) (d k))
      (/ (n k) (+ (d k) (cont-frac n d (- k 1))))))

;; > (/ 1 (cont-frac (λ (i) 1.0)
;;                   (λ (i) 1.0)
;;                   13))
;; 1.6180257510729614
```

```
;; Exercise 1.37 b) (page 96)

;; Iterative solution

(define (cont-frac n d k)
  (define (cf-iter i)
    (if (= i k)
        (/ (n i) (d i))
        (/ (n i) (+ (d i) (cf-iter (+ i 1))))))
  (cf-iter 1))

;; > (/ 1 (cont-frac (λ (i) 1.0)
;;                   (λ (i) 1.0)
;;                   13))
;; 1.6180257510729614
```

Exercise 1.38: In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ... Write a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

```
;; Exercise 1.38 (page 96)

(+ 2 (cont-frac (λ (i) 1.0)
                (λ (i) (if (= (remainder i 3) 2)
                           (* (/ (+ i 1) 3) 2)
                           1))
                13))

;; 2.718281828735696
```

Newton's method

When we first introduced the square-root procedure, in Section 1.1.7, we mentioned that this was a special case of *Newton's method*. If $x \mapsto g(x)$ is a differentiable function, then a solution of the equation $g(x) = 0$ is a fixed point of the function $x \mapsto f(x)$, where

$$f(x) = x - \frac{g(x)}{Dg(x)}$$

```
(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

Exercise 1.40: Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

```
;; Exercise 1.40 (page 103)

(define (cubic a b c)
  (λ (x) (+ (* x x x)
            (* a x x)
            (* b x)
            c)))

(define (solve a b c)
  (newtons-method (cubic a b c) 1.0))
```

Exercise 1.41: Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double  inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

```
;; Exercise 1.41 (page 103)

(define (double f)
  (λ (x) (f (f x))))

;; > (((double (double double)) inc) 5)
;; 21
```

Exercise 1.42: Let f and g be two one-argument functions. The *composition* f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

```
((compose square inc) 6)
49
```

```
;; Exercise 1.42 (page 103)

(define (compose f g)
  (λ (x) (f (g x))))

;; > ((compose (λ (x) (* x x))
;;             (λ (x) (+ 1 x)))
;;    6)
;; 49
```

Exercise 1.43: If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if f is the function $x \mapsto x + 1$, then the n^{th} repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n-th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

Hint: You may find it convenient to use `compose` from Exercise 1.42.

```
;; Exercise 1.43 (page 104)

(define (repeated f n)
  (if (= n 0)
      (λ (x) x)
      (compose f (repeated f (- n 1)))))
```

Exercise 1.44: The idea of *smoothing* a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x-dx)$, $f(x)$, and $f(x+dx)$. Write a procedure `smooth` that takes as input a procedure that computes f and returns a procedure that computes the smoothed f. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the *n-fold smoothed function*. Show how to generate the n-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

```
;; Exercise 1.44 (page 104)

(define (smoothed-f f dx)
  (λ (x) (~>> (list x (+ x dx) (- x dx))
              (map f)
              (average))))

;; > (map (smoothed-f (λ (x) (* x x)) 1.0) (range 10))
;; '(0.6666666666666666
;;   1.6666666666666667
;;   4.666666666666667
;;   9.666666666666666
;;   16.666666666666668
;;   25.666666666666668
;;   36.666666666666664
;;   49.666666666666664
;;   64.66666666666667
;;   81.66666666666667)
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

The rights and privileges of first-class citizens

To be named by variables.

To be passed as arguments to procedures.

To be returned as values of procedures.

To be incorporated into data structures.

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the "rights and privileges" of first-class elements are:[64]

- They may be named by variables.
- They may be passed as arguments to procedures.
- They may be returned as the results of procedures.
- They may be included in data structures.[65]

Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation,

[64] The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916-1975).

[65] We'll see examples of this after we introduce data structures in Chapter 2.

Language		Higher-order functions		Nested functions		Non-local variables		Notes
		Arguments	Results	Named	Anonymous	Closures	Partial application	
Algol family	ALGOL 60	Yes	No	Yes	No	Downwards	No	Have function types.
	ALGOL 68	Yes	Yes[8]	Yes	Yes	Downwards[9]	No	
	Pascal	Yes	No	Yes	No	Downwards	No	
	Ada	Yes	No	Yes	No	Downwards	No	
	Oberon	Yes	Non-nested only	Yes	No	Downwards	No	
	Delphi	Yes	Yes	Yes	2009	2009	No	
C family	C	Yes	Yes	No	No	No	No	Has function pointers.
	C++	Yes	Yes	C++11[10]	C++11[11]	C++11[11]	C++11	Has function pointers, function objects. (Also, see below.) Explicit partial application possible with `std::bind`.
	C#	Yes	Yes	7	2.0 / 3.0	2.0	3.0	Has delegates (2.0) and lambda expressions (3.0).
	Objective-C	Yes	Yes	Using anonymous	2.0 + Blocks[12]	2.0 + Blocks	No	Has function pointers.
	Java	Partial	Partial	Using anonymous	Java 8	Java 8	No	Has anonymous inner classes.
	Go	Yes	Yes	Using anonymous	Yes	Yes	Yes[13]	
	Limbo	Yes	Yes	Yes	Yes	Yes	No	
	Newsqueak	Yes	Yes	Yes	Yes	Yes	No	
	Rust	Yes	Yes	Yes	Yes	Yes	No	
Functional languages	Lisp	Syntax	Syntax	Yes	Yes	Common Lisp	No	(see below)
	Scheme	Yes	Yes	Yes	Yes	Yes	SRFI 26[14]	
	Julia	Yes	Yes	Yes	Yes	Yes	Yes	
	Clojure	Yes	Yes	Yes	Yes	Yes	Yes	
	ML	Yes	Yes	Yes	Yes	Yes	Yes	
	Haskell	Yes	Yes	Yes	Yes	Yes	Yes	
	Scala	Yes	Yes	Yes	Yes	Yes	Yes	
	F#	Yes	Yes	Yes	Yes	Yes	Yes	
	OCaml	Yes	Yes	Yes	Yes	Yes	Yes	

In general, programming languages impose restrictions on the ways in which computational elements can be manipulated. Elements with the fewest restrictions are said to have *first-class* status. Some of the "rights and privileges" of first-class elements are:[64]

- They may be named by variables.
- They may be passed as arguments to procedures.

[64]The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916-1975).

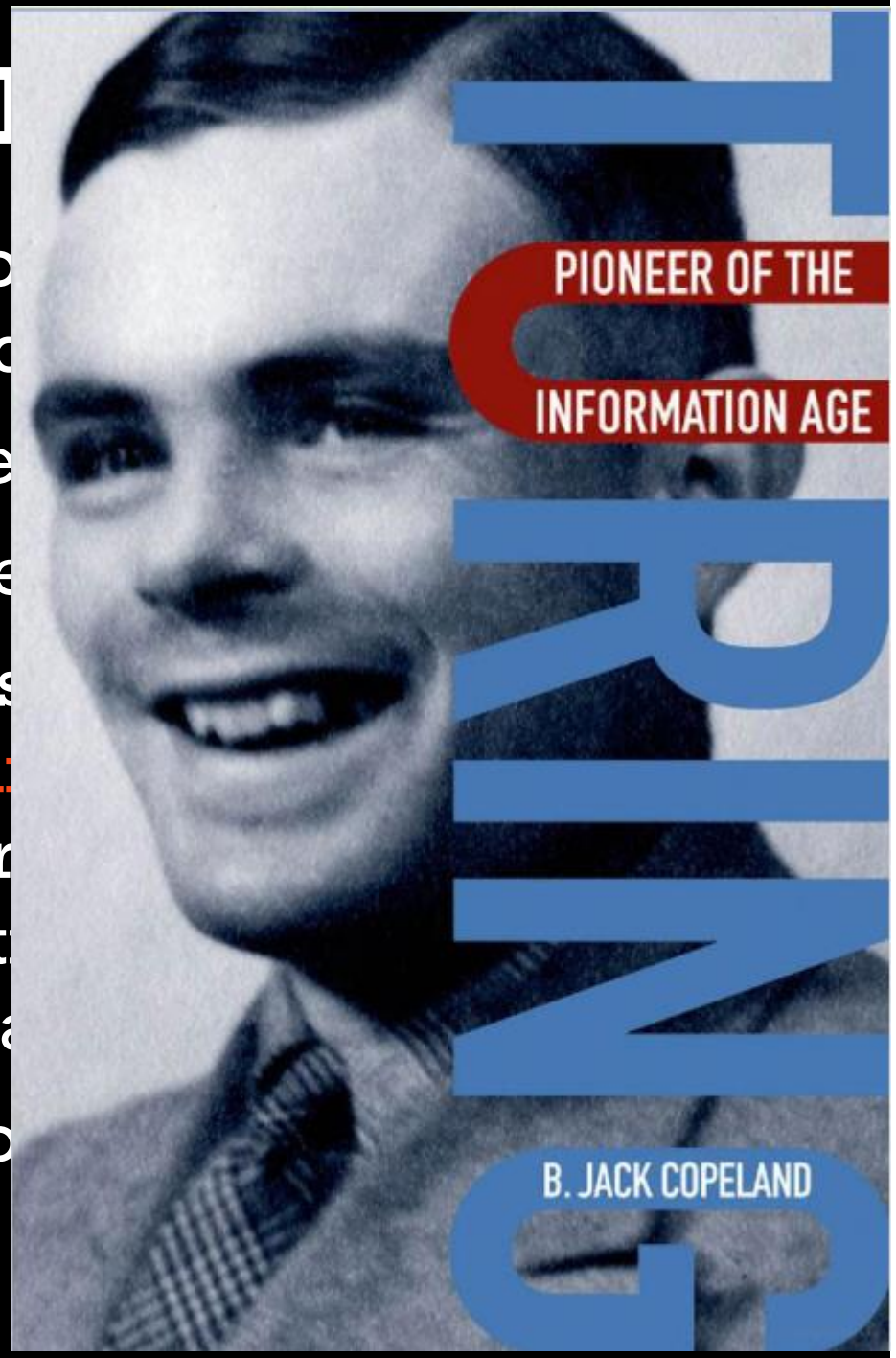
Lisp, unlike other common programming languages, awards procedures full first-class status. This poses challenges for efficient implementation,

[64]The notion of first-class status of programming-language elements is due to the British computer scientist Christopher Strachey (1916-1975).

[65]We'll see examples of this after we introduce data structures in Chapter 2.

Wh

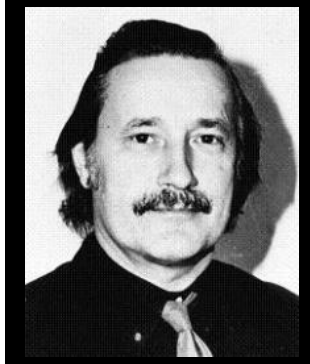
- Co
  (Sc
- De
- We
- His
  Fu
  for
  int
  "pa
- Co



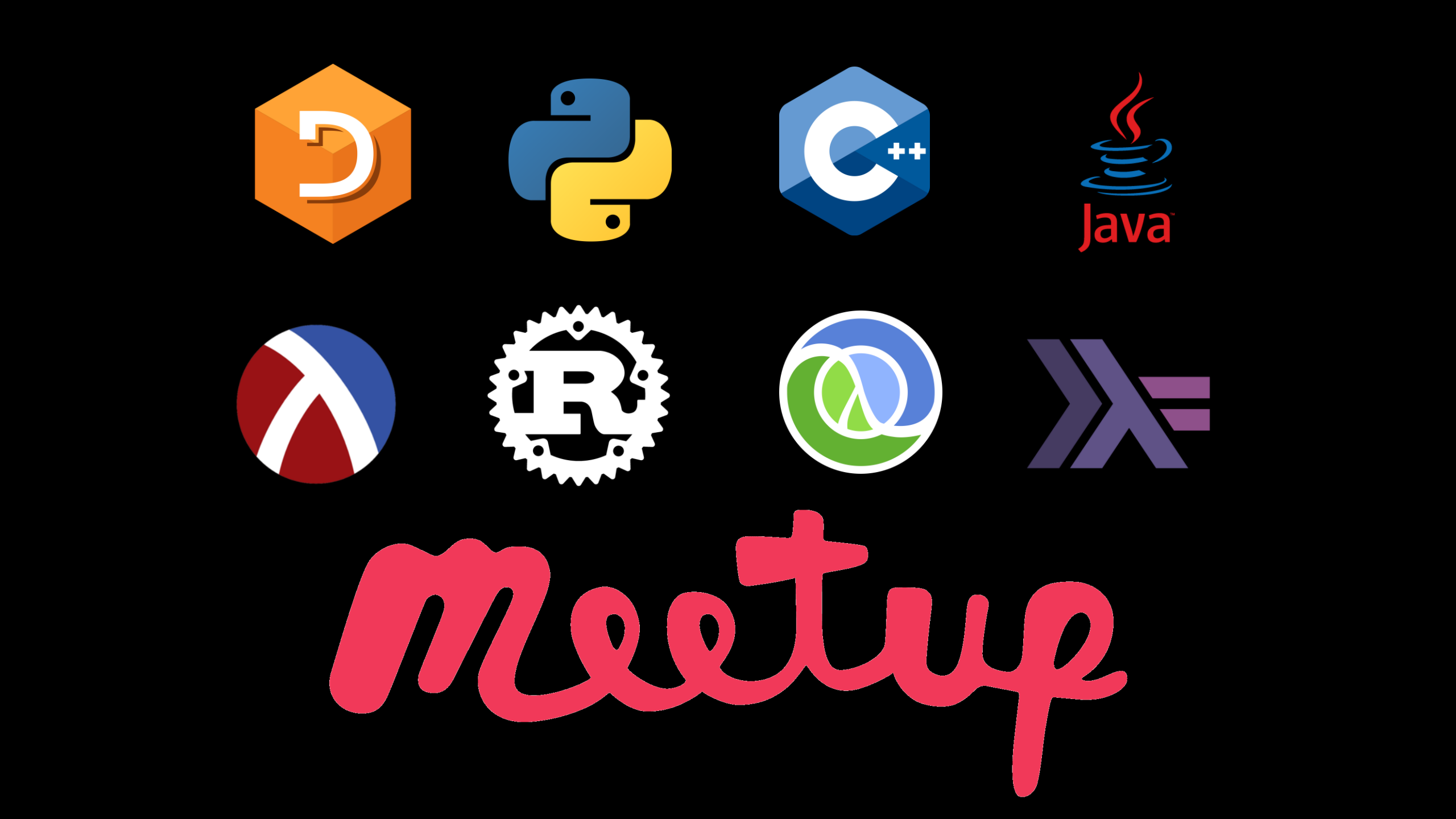
Strachey hard on his heels. Hackers, Steven Levy wrote in his 1984 classic of the same name, are 'computer programmers and designers who regard computing as the most important thing in the world'.[102] Nowadays the term is generally used for programmers who break into other people's computer systems, but real hackers know the true meaning of the word. When Strachey showed up in Manchester, Turing decided to drop him in at the deep end, and suggested he try writing a program to make the computer simulate itself. At that time this was a devilishly difficult task—a bit like running the four-minute mile, as Roger Bannister would soon do for the first time in recorded history. When Strachey left the laboratory, Turing turned to Robin Gandy and said impishly, 'That will keep *him* busy!'[103]

It did keep him busy. Strachey read Turing's programming manual assiduously. This was 'famed in those days for its incomprehensibility', Strachey said.[104] An ardent pianist, he appreciated the potential of Turing's terse directions on how to program musical notes. Eventually he returned to Manchester with twenty or so pages covered in lines of programming code; previously the longest program to run on the computer had amounted to no more than about half a page of code.[105] 'Turing came in and gave me a typical high-speed, high-pitched description of how to use the machine,' Strachey recollected.[106] Then he was left alone at the computer's console until the following morning. It was the first of a lifetime of all-night programming sessions. 'I sat in front of this enormous machine,' he said, 'with four or five rows of twenty switches and things, in a room that felt like the control room of a battle-ship.'[107] Turing came back in the morning to find Strachey's simulator program running like a top—and then, without warning, the computer raucously hooted out the British National Anthem, 'God save the King'. Turing was his customary monosyllabic self. 'Good show,' he said in an enthusiastic way.[108]

Strachey had started a craze and more programs were soon written. The BBC heard about the musical computer and a *Children's Hour* radio presenter known as Auntie was despatched with a recording team to cover the story.[109] Besides 'God save the King', the BBC recorded a version of Glen Miller's swinging 'In the Mood'. The American superstar's hair might have curled up with horror if he had been able to hear the



y".


Java
meetup