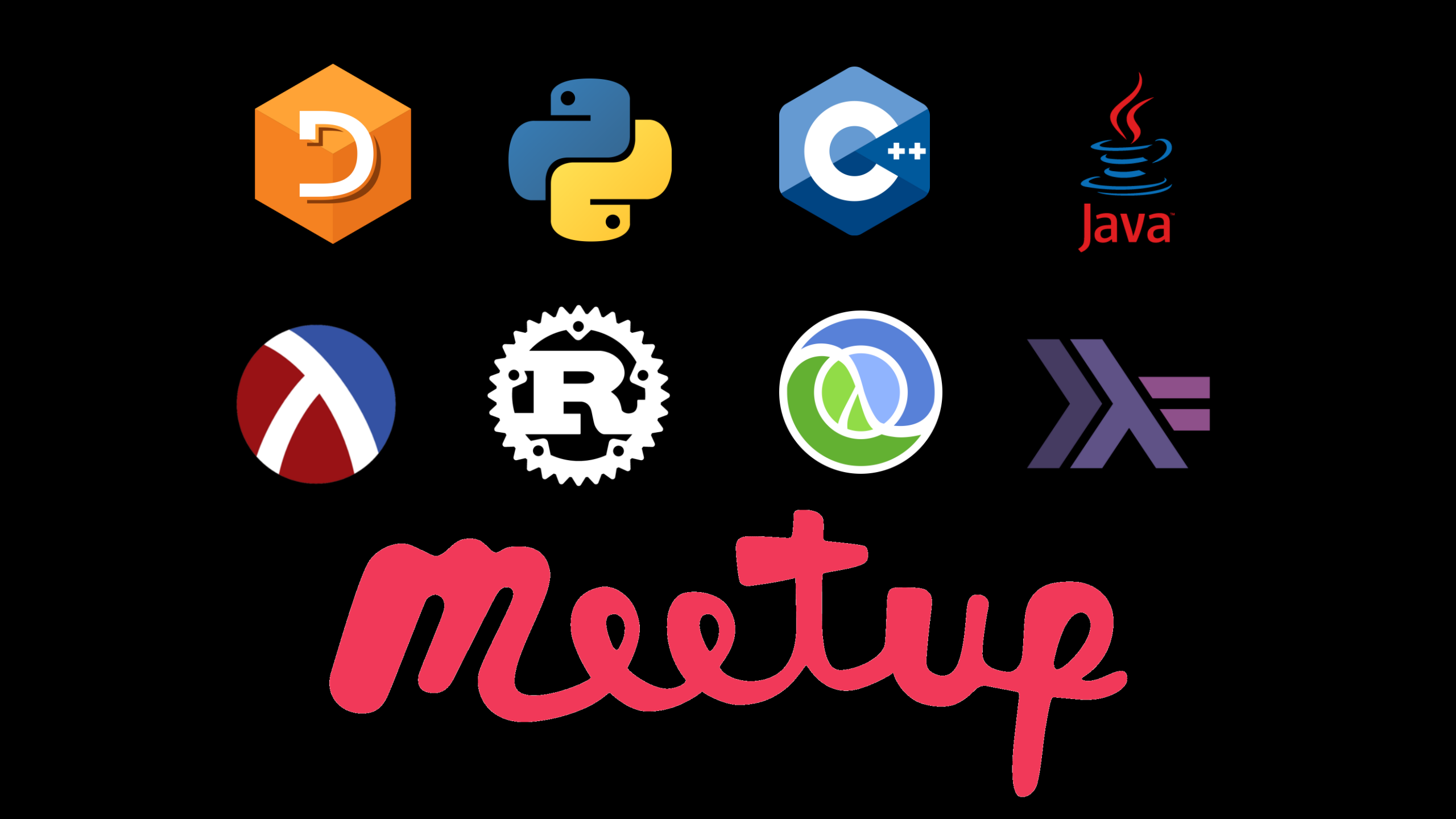
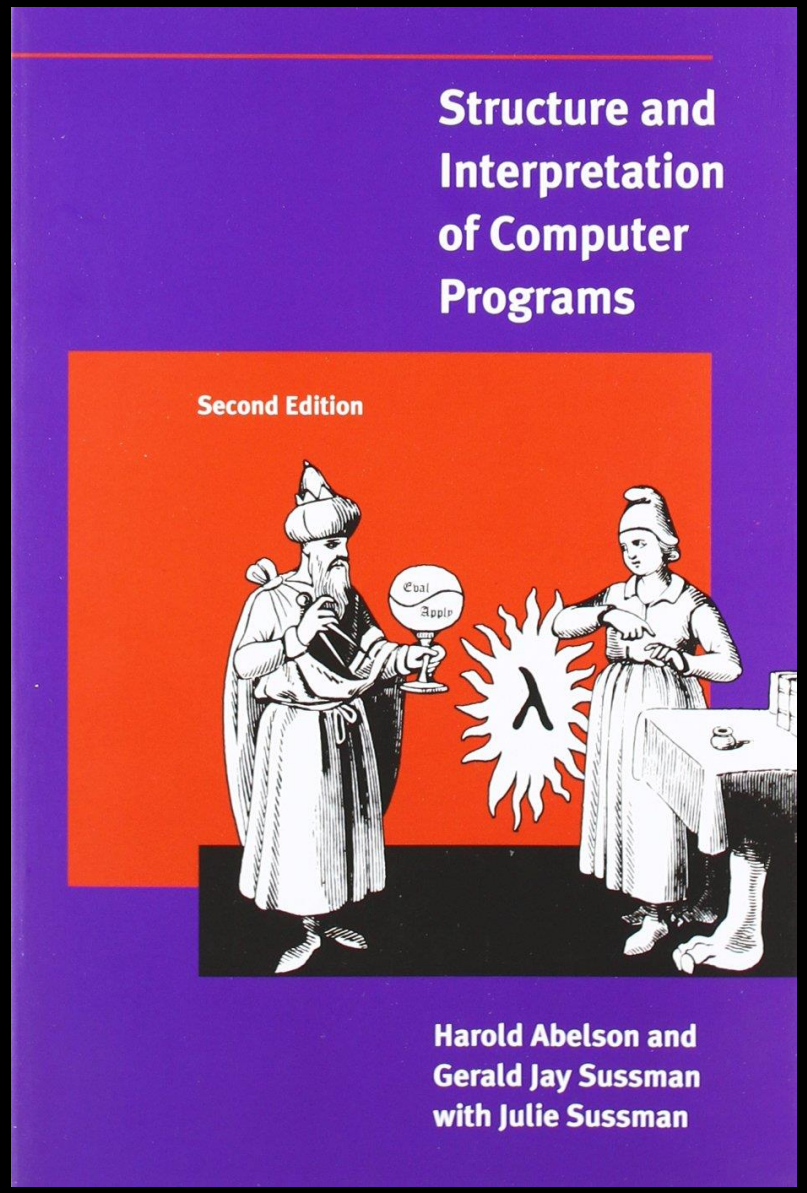

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 2.2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

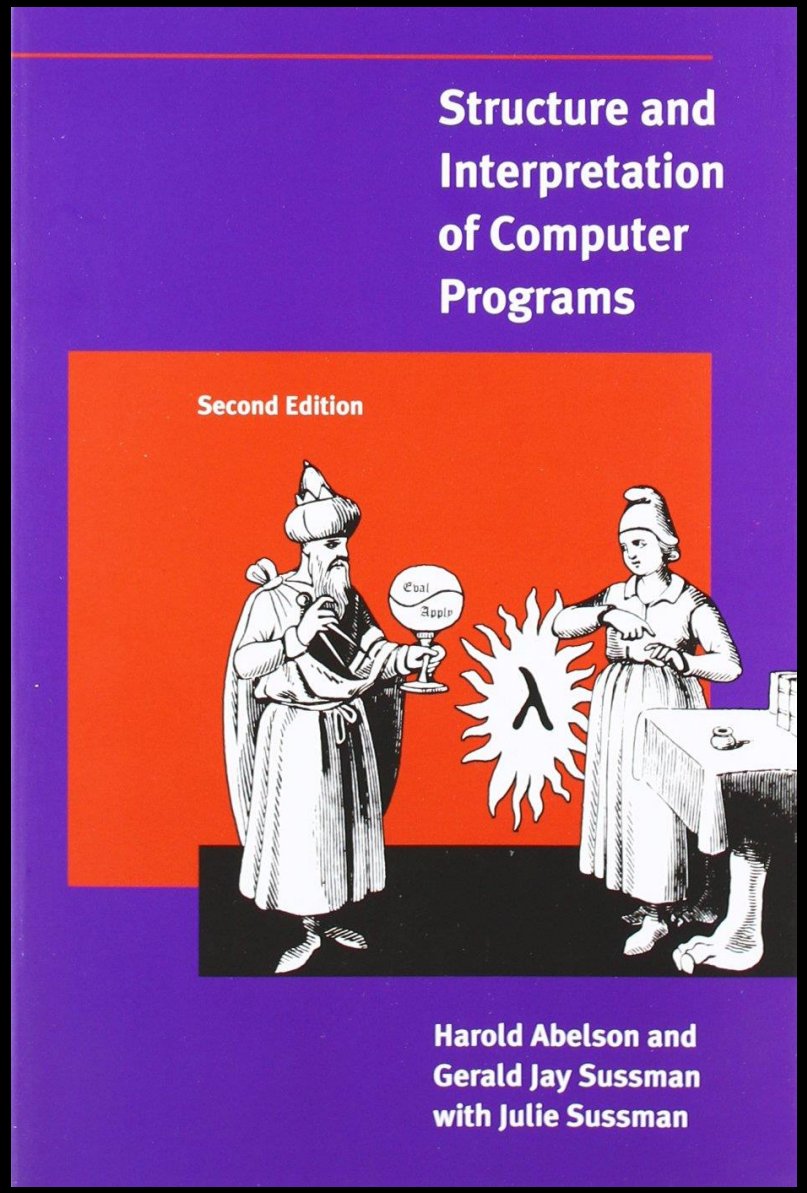
Following

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 2.2

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

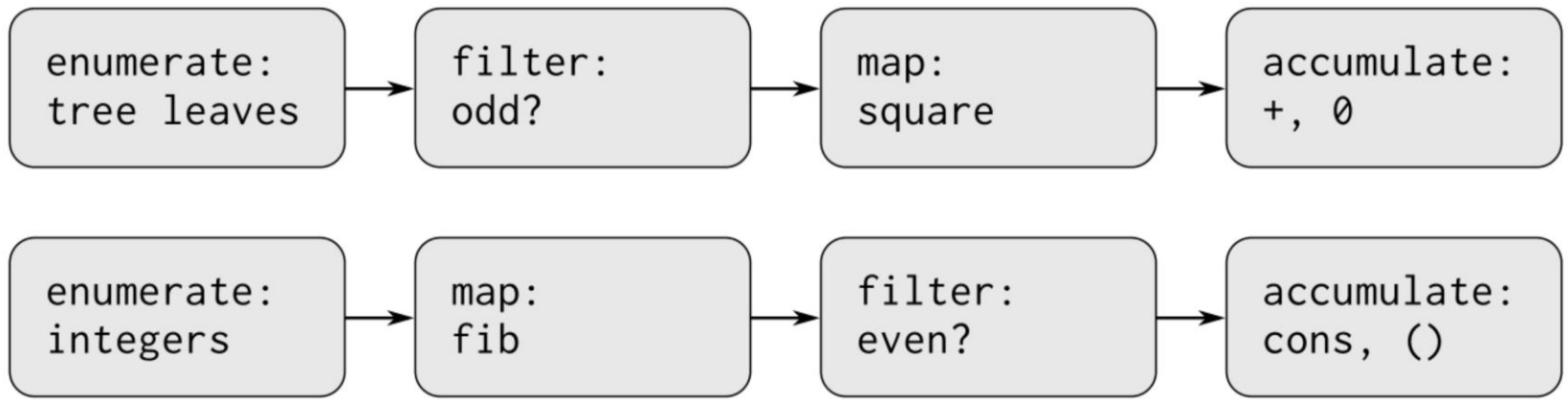


Figure 2.7: The signal-flow plans for the procedures `sum-odd-squares` (top) and `even-fibs` (bottom) reveal the commonality between the two programs.

In `sum-odd-squares`, we begin with an *enumerator*, which generates a "signal" consisting of the leaves of a given tree. This signal is passed through a *filter*, which eliminates all but the odd elements. The resulting signal is in turn passed through a *map*, which is a "transducer" that applies the `square` procedure to each element. The output of the map is then fed to an *accumulator*, which combines the elements using `+`, starting from an initial `0`. The plan for `even-fibs` is analogous.

Our two procedures decompose the computations in a different way, spreading the enumeration over the program and mingling it with the map, the filter, and the accumulation. If we could organize our programs to make the signal-flow structure manifest in the procedures we write, this would increase the conceptual clarity of the resulting code.

```
(define (sum-odd-squares tree)
  (accumulate
   + 0 (map square (filter odd? (enumerate-tree tree)))))
```

```
;; With Threading Macro

(require threading)

(define (sum-odd-squares tree)
  (~>> tree
       (enumerate-tree)
       (filter odd?)
       (map square)
       (accumulate + 0)))
```

[15]Richard Waters (1979) developed a program that automatically analyzes traditional Fortran programs, viewing them in terms of maps, filters, and accumulations. He found that fully 90 percent of the code in the Fortran Scientific Subroutine Package fits neatly into this paradigm. One of the reasons for the success of Lisp as a programming language is that lists provide a standard medium for expressing ordered collections so that they can be manipulated using higher-order operations. The programming language APL owes much of its power and appeal to a similar choice. In APL all data are represented as arrays, and there is a universal and convenient set of generic operators for all sorts of array operations.

Exercise 2.33: Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
(define (map p sequence)
  (accumulate (lambda (x y) ⟨??⟩) nil sequence))
(define (append seq1 seq2)
  (accumulate cons ⟨??⟩ ⟨??⟩))
(define (length sequence)
  (accumulate ⟨??⟩ 0 sequence))
```

```
;; Exercise 2.33 (page 161)

(define (map p sequence)
  (accumulate (λ (x y) (cons (p x) y)) `() sequence))

(define (append seq1 seq2)
  (accumulate cons seq1 seq2))

(define (length sequence)
  (accumulate (λ (_ acc) (+ acc 1)) 0 sequence))
```

Exercise 2.34: Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called *Horner's rule*, which structures the computation as

$$(\ldots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0.$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0.[16]

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) ⟨??⟩)
              0
              coefficient-sequence))
```

```
;; Exercise 2.34 (page 162-3)

(define (horner-eval x coefficient-sequence)
  (accumulate (λ (coeff acc) (+ coeff (* x acc)))
              0
              coefficient-sequence))

;; > (horner-eval 2 (list 1 3 0 5 0 1))
;; 79
```

Exercise 2.36: The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init ⟨??⟩)
            (accumulate-n op init ⟨??⟩))))
```

```
;; Exericse 2.36 (page 163)

(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      '()
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))

;; > (accumulate-n + 0 '((0 1) (1 2) (2 3)))
;; '(3 6)
```

Hoogle Translate

zipwith

Clojure	**map***	core	Doc
Racket	**map***	base	Doc
OCaml	**map2**	List	Doc
F#	**map2**	Seq	Doc
C++	**transform***	<algorithm>	Doc
Haskell	**zipWith**	Prelude	Doc

Exercise 2.37: Suppose we represent vectors $\mathbf{v} = (v_i)$ as sequences of numbers, and matrices $\mathbf{m} = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

`(dot-product v w)`	returns the sum $\Sigma_i v_i w_i$;
`(matrix-*-vector m v)`	returns the vector $\mathbf{t}$, where $t_i = \Sigma_j m_{ij} v_j$;
`(matrix-*-matrix m n)`	returns the matrix $\mathbf{p}$, where $p_{ij} = \Sigma_k m_{ik} n_{kj}$;
`(transpose m)`	returns the matrix $\mathbf{n}$, where $n_{ij} = m_{ji}$.

We can define the dot product as[17]

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in Exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map ⟨??⟩ m))

(define (transpose mat)
  (accumulate-n ⟨??⟩ ⟨??⟩ mat))
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map ⟨??⟩ m)))
```

```
;; Exercise 2.37 (page 163-5)

(define (dot-product v w)
  (accumulate + 0 (map * v w)))

;; > (dot-product (range 4) (range 4))
;; 14
```

```
(define (matrix-*-vector m v)
  (map (λ (row) (dot-product v row)) m))

;; > (define mat '((0 1 2) (1 2 3) (2 3 4)))
;; > (matrix-*-vector mat (range 3))
;; '(5 8 11)
```

```
(define (transpose mat)
  (accumulate-n cons '() mat))
```

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (λ (row) (matrix-*-vector cols row))  m)))

;; > (matrix-*-matrix mat mat2)
;; '((5 5 5) (8 8 8) (11 11 11))
;; > (matrix-*-matrix mat2 mat)
;; '((0 0 0) (3 6 9) (6 12 18))
```

Exercise 2.38: The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

What are the values of

```
(fold-right / 1 (list 1 2 3))
(fold-left / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left list nil (list 1 2 3))
```

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

```
;; Exercise 2.38 (page 165)

;; > (foldr / 1 '(1 2 3))
;; 1 1/2
;; > (foldl / 1 '(1 2 3))
;; 1 1/2
;; > (foldr list '() '(1 2 3))
;; '(1 (2 (3 ())))
;; > (foldl list '() '(1 2 3))
;; '(3 (2 (1 ())))

;; asssociativity / commutative
```

```
(define (prime-sum-pairs n)
  (map make-pair-sum
       (filter prime-sum? (flatmap
                           (lambda (i)
                             (map (lambda (j) (list i j))
                                  (enumerate-interval 1 (- i 1))))
                           (enumerate-interval 1 n)))))
```

```
({<⌿2↑ω∧prime⍣¯1↑ω}⍨t)⌿t←,∘.{α ω,(α+ω)}⍨⍳5
```

```
prime ← (2=0+.=⍳|⊢)
```

Exercise 2.40: Define a procedure `unique-pairs` that, given an integer n, generates the sequence of pairs (i, j) with $1 \leq j < i \leq n$. Use `unique-pairs` to simplify the definition of `prime-sum-pairs` given above.

```
;; Exercise 2.40 (169)

(require algorithms)
(require threading)

(define (unique-pairs n)
  (let ((lst (range 1 (+ n 1))))
       (~>> lst
            (cartesian-product lst)
            (filter increasing?)
            (remove-duplicates))))

;; > (unique-pairs 4)
;; '((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))
```

```
;; Attempt #1 at increasing

(define (increasing? lst)
  (~>> lst
       (reverse)
       (adjacent-map _ -)
       (andmap positive?)))
```

```
;; Attempt #2 at increasing

(define (increasing? lst)
  (~>> lst
       (adjacent-map _ <)
       (andmap identity)))
```

```
;; Attempt #3 at increasing

(define (all? lst)
  (andmap identity lst))

(define (increasing? lst)
  (~>> lst
       (adjacent-map _ <)
       (all?)))
```

(</¨t)/t←,∘.,˜ι3

Exercise 2.41: Write a procedure to find all ordered triples of distinct positive integers i, j, and k less than or equal to a given integer n that sum to a given integer s.

```
(require algorithms)
(require threading)

(define (triplets-sum-k n k)
  (let ((lst (range 1 (+ n 1))))
        (~>> lst
             (cartesian-product lst lst)
             (filter increasing?)
             (filter (λ (t) (= (sum t) k))))))
```

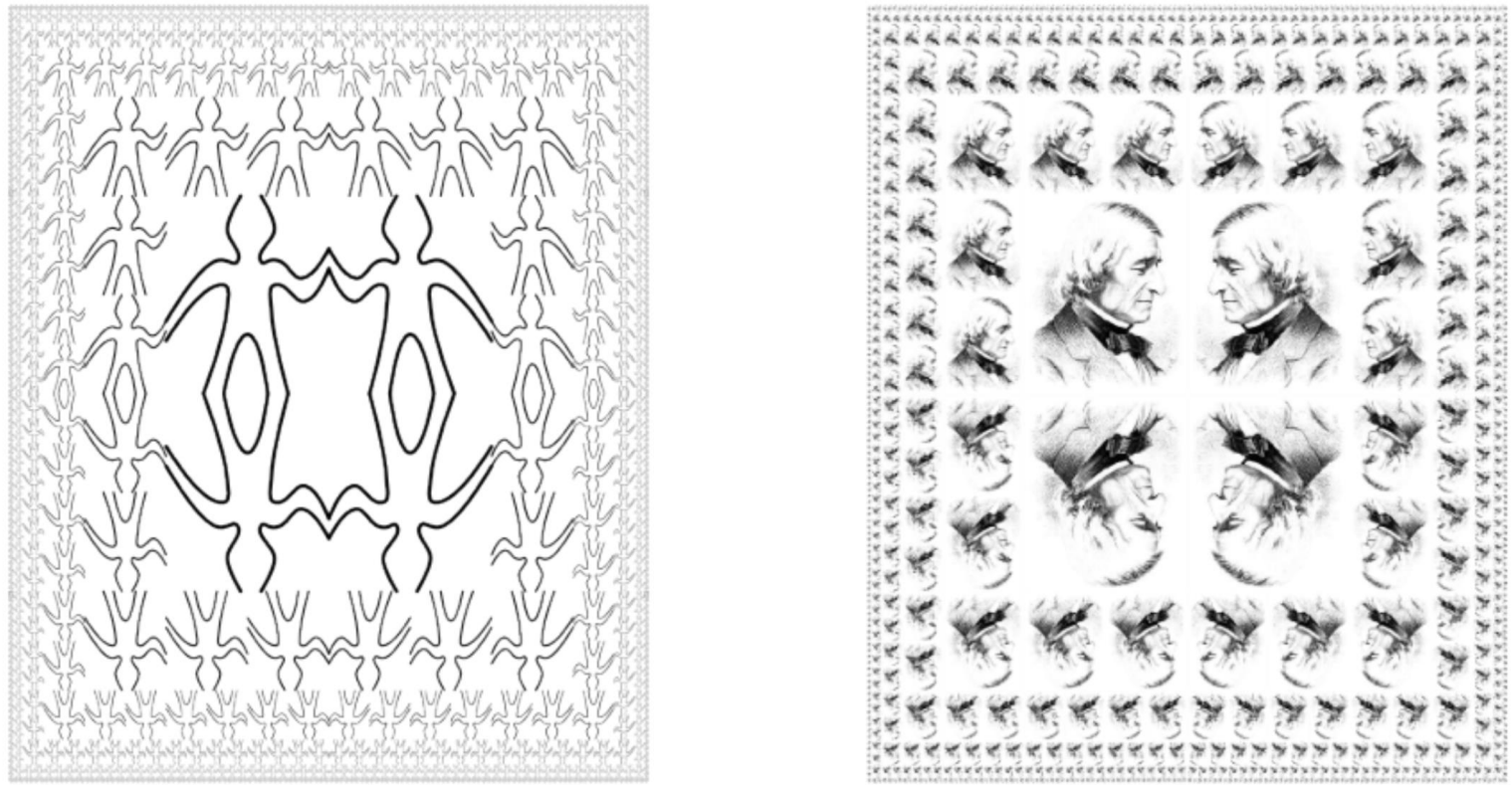


Figure 2.9: Designs generated with the picture language.

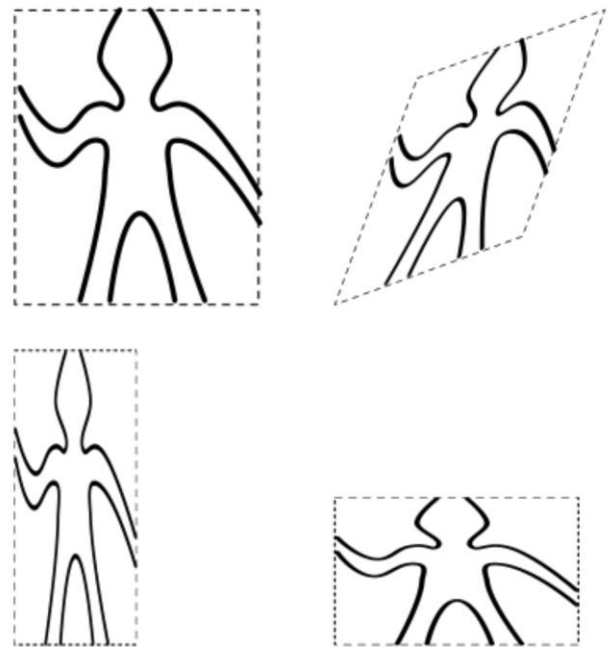


Figure 2.10: Images produced by the `wave` painter, with respect to four different frames. The frames, shown with dotted lines, are not part of the images.

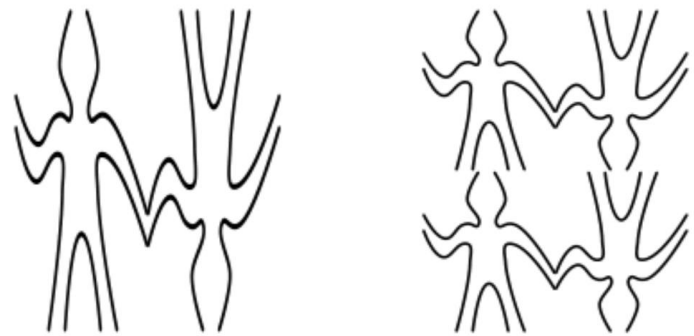


Figure 2.12: Creating a complex figure, starting from the `wave` painter of Figure 2.10.

Exercise 2.44: Define the procedure `up-split` used by `corner-split`. It is similar to `right-split`, except that it switches the roles of `below` and `beside`.

```
define (up-split painter n)
```

```
> (paint (right-split einstein 3))
```

```
> (paint (up-split einstein 3))
```

```
;; Exercise 2.44 (page 179)

(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (up-split painter (- n 1))))
        (below painter (beside smaller smaller)))))

(define (right-split painter n)
  (if (= n 0)
      painter
      (let ((smaller (right-split painter (- n 1))))
        (beside painter (below smaller smaller)))))
```

Exercise 2.45: `right-split` and `up-split` can be expressed as instances of a general splitting operation. Define a procedure `split` with the property that evaluating

```
(define right-split (split beside below))
(define up-split (split below beside))
```

produces procedures `right-split` and `up-split` with the same behaviors as the ones already defined.

```
;; Exercise 2.45 (page 182)

(define (split f g)
  (define (rec painter n)
    (if (= n 0)
        painter
        (let ((smaller (rec painter (- n 1))))
          (f painter (g smaller smaller)))))
  rec)

(define right-split (split beside below))
(define up-split (split below beside))
```

Structure & Interpretation of Computer Programs

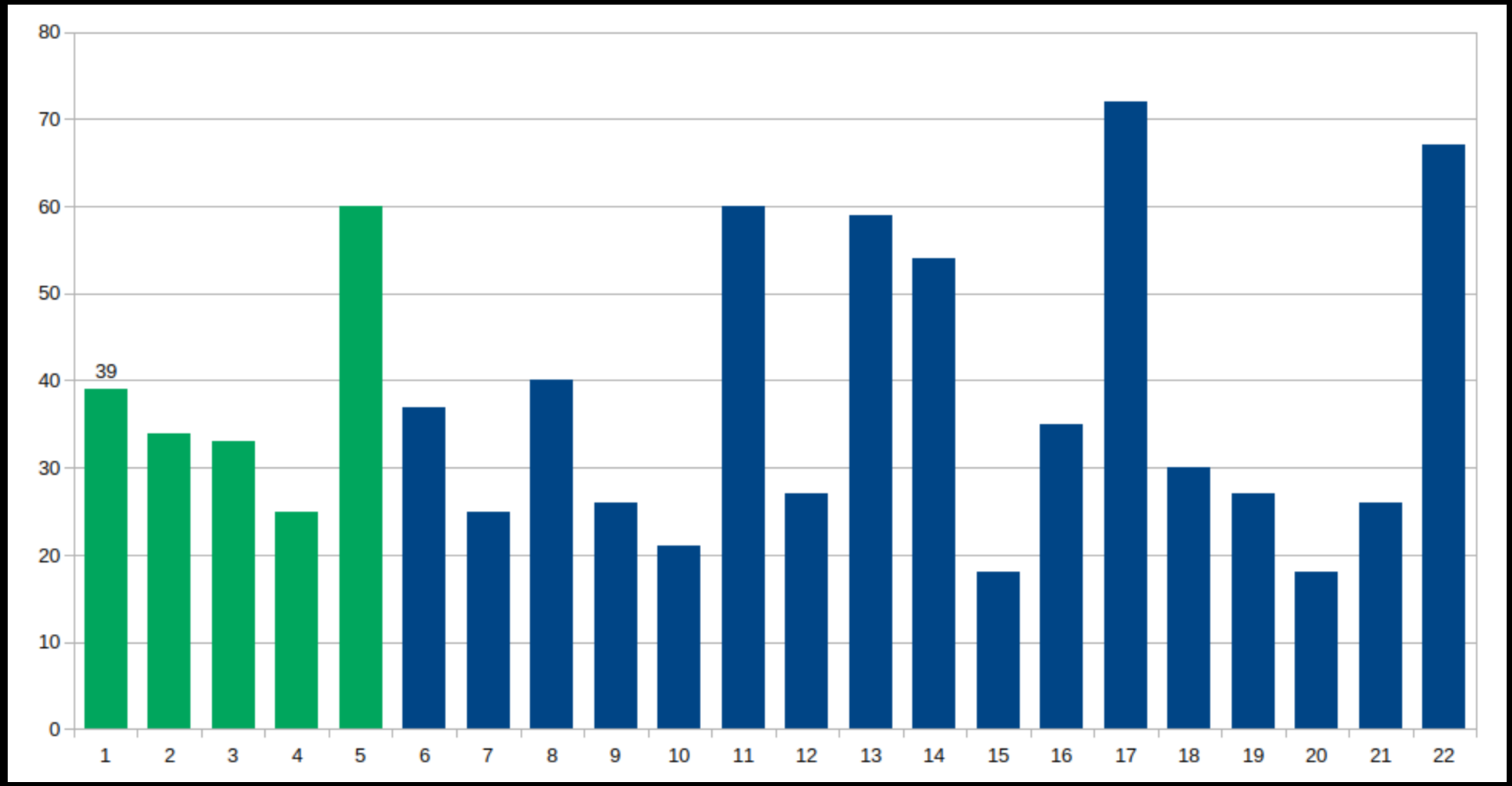
Harold Abelson

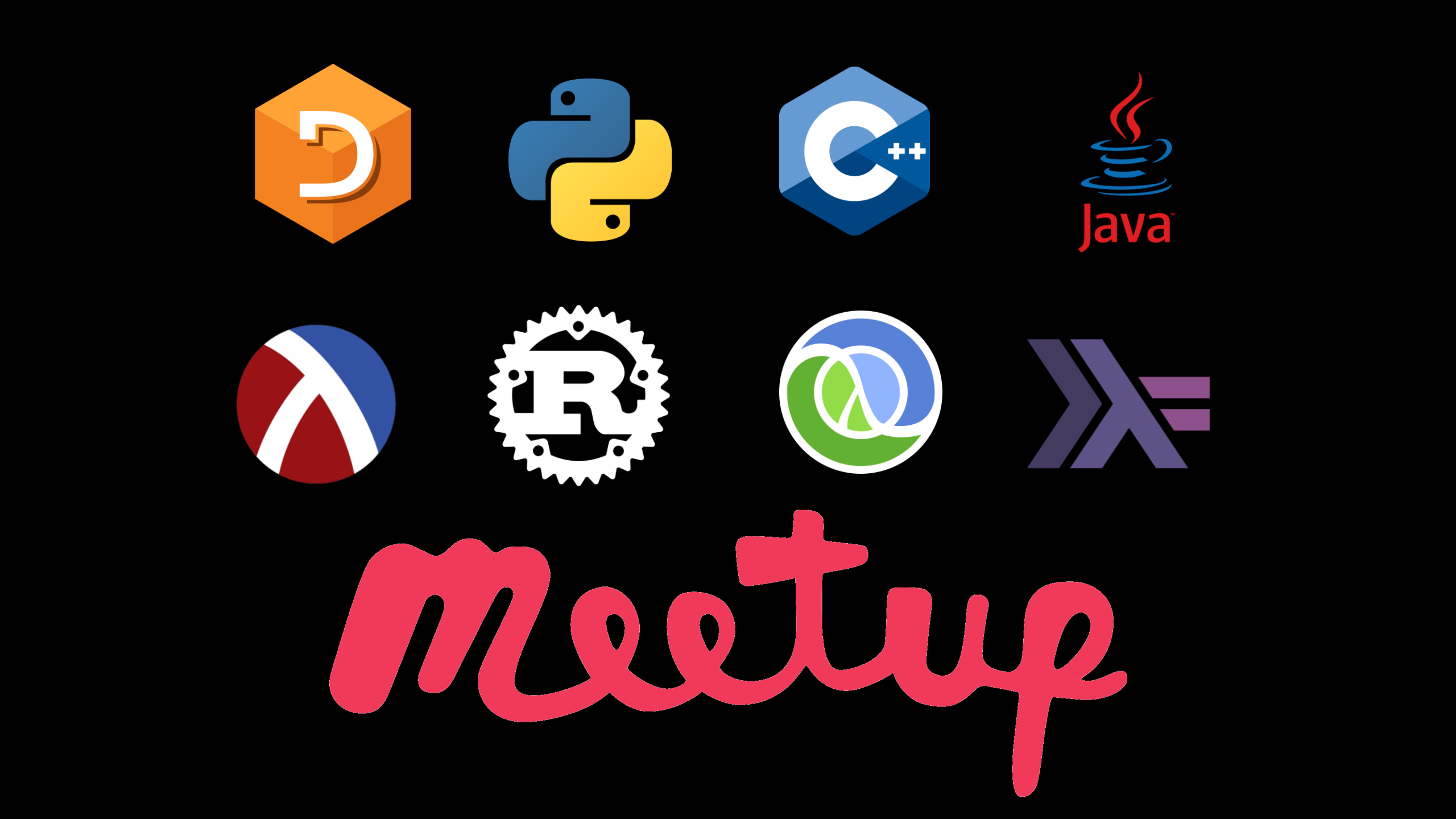
Gerald Jay Sussman

“there is an operation called **rotate**”

Hal Abelson

Lecture 3A, SICP


80
70
60
50
40
30
20
10
0
39
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22


Java
meetup