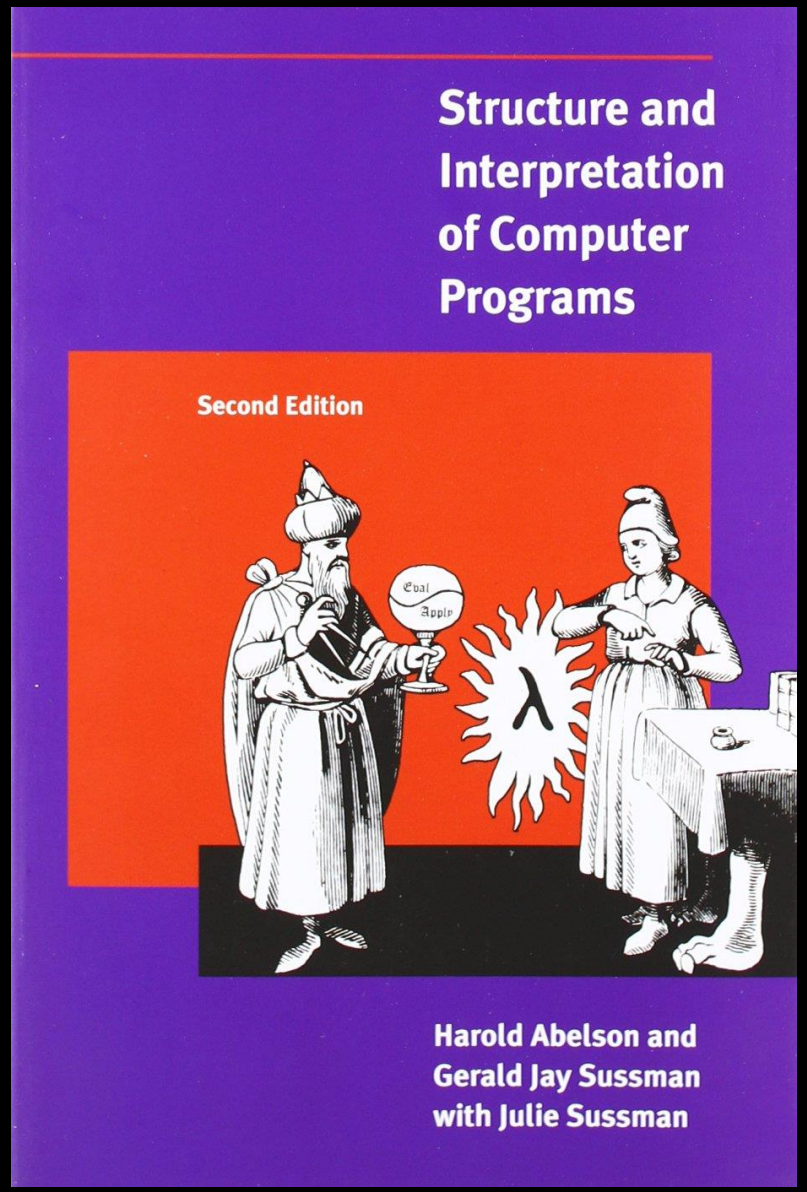


Structure and Interpretation of Computer Programs

Chaper 3.5 2/2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

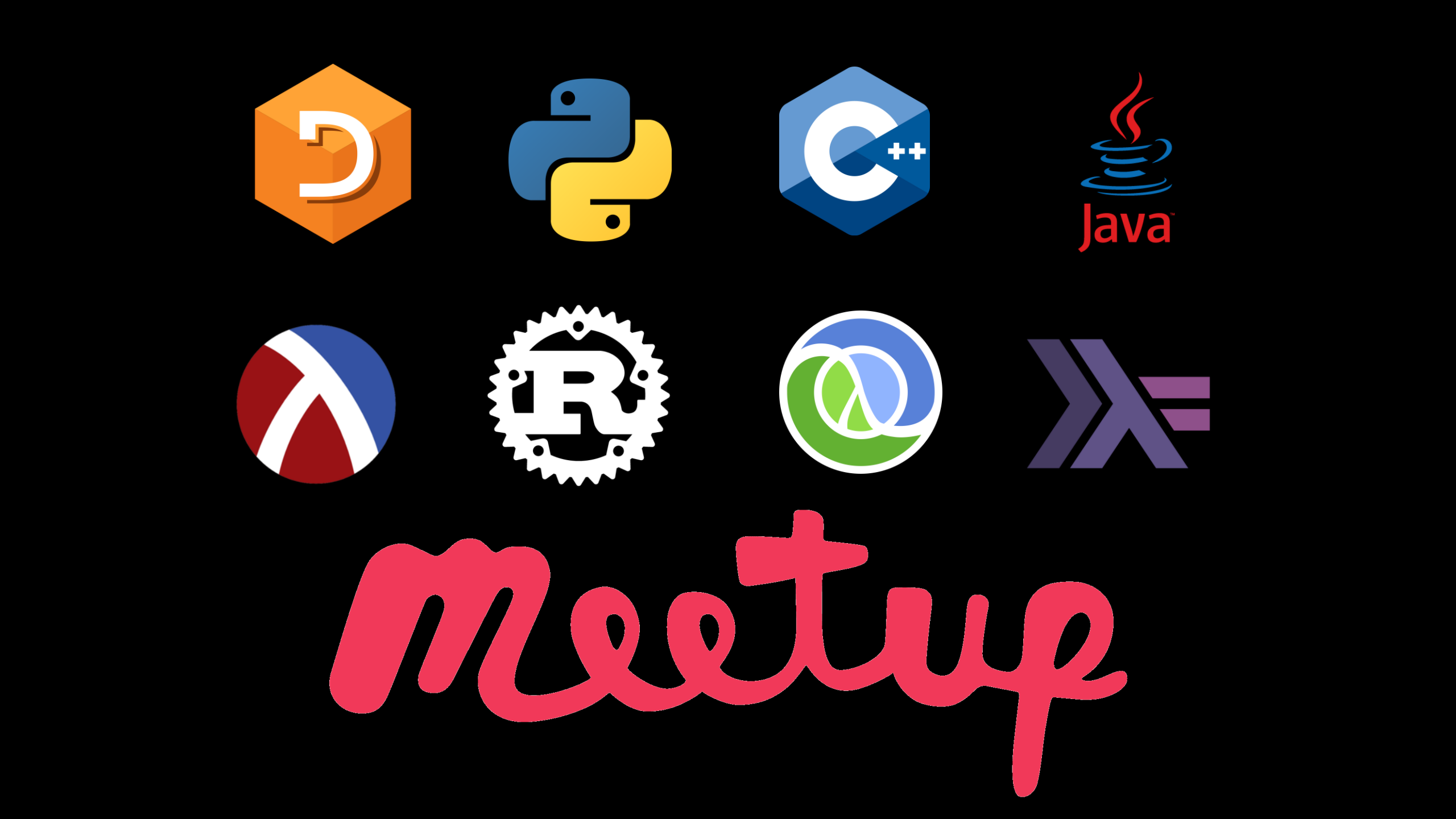
Following

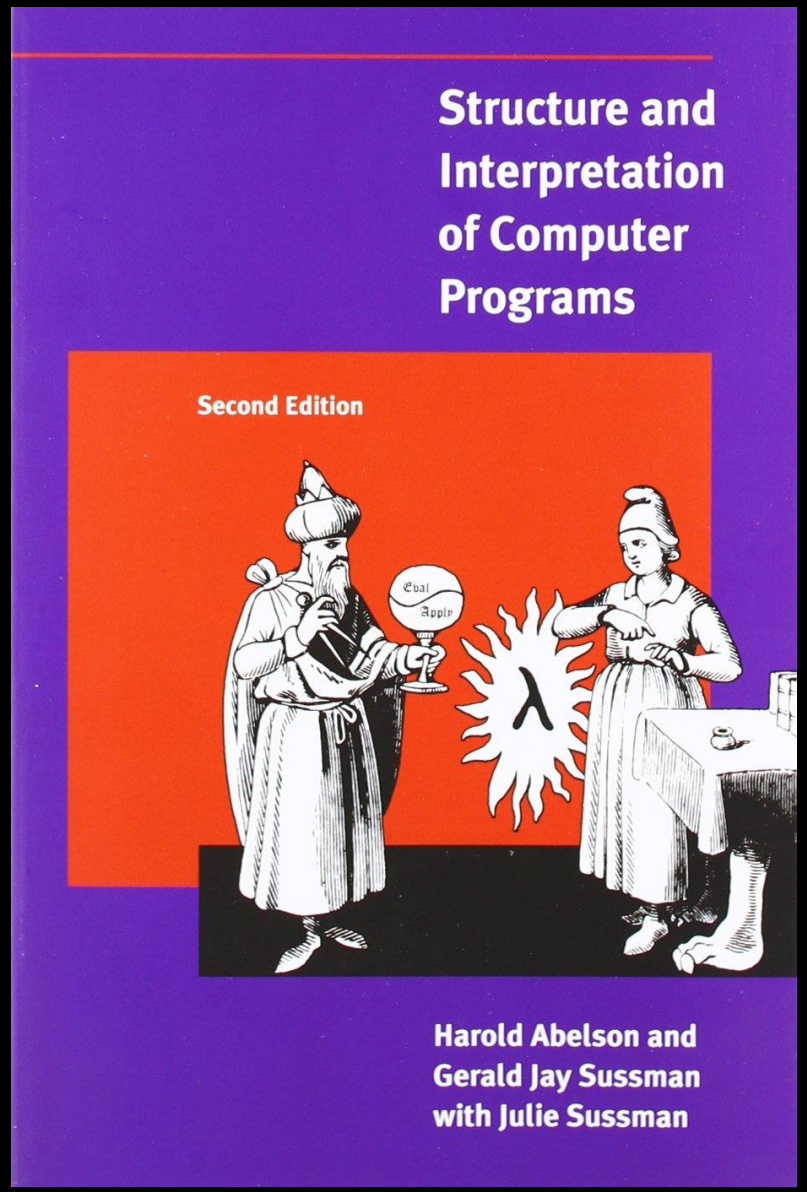
Programming Languages Virtual Meetup

@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


Java
meetup



Structure and Interpretation of Computer Programs

Chaper 3.5 2/2

Streams as signals

We began our discussion of streams by describing them as computational analogs of the "signals" in signal-processing systems. In fact, we can use streams to model signal-processing systems in a very direct way, representing the values of a signal at successive time intervals as consecutive elements of a stream.

Exercise 3.74: Alyssa P. Hacker is designing a system to process signals coming from physical sensors. One important feature she wishes to produce is a signal that describes the *zero crossings* of the input signal. That is, the resulting signal should be +1 whenever the input signal changes from negative to positive, −1 whenever the input signal changes from positive to negative, and 0 otherwise. (Assume that the sign of a 0 input is positive.) For example, a typical input signal with its associated zero-crossing signal would be

```
... 1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4 ...
... 0 0  0  0  0   -1   0  0  0   0   1  0 0 ...
```

In Alyssa's system, the signal from the sensor is represented as a stream `sense-data` and the stream `zero-crossings` is the corresponding stream of zero crossings. Alyssa first writes a procedure `sign-change-detector` that takes two values as arguments and compares the signs of the values to produce an appropriate 0, 1, or - 1. She then constructs her zero-crossing stream as follows:

```
(define (make-zero-crossings input-stream last-value)
  (cons-stream
   (sign-change-detector
    (stream-car input-stream)
    last-value)
   (make-zero-crossings
    (stream-cdr input-stream)
    (stream-car input-stream))))
(define zero-crossings
  (make-zero-crossings sense-data 0))
```

Alyssa's boss, Eva Lu Ator, walks by and suggests that this program is approximately equivalent to the following one, which uses the generalized version of `stream-map` from Exercise 3.50:

```
(define zero-crossings
  (stream-map sign-change-detector
              sense-data
              ⟨expression⟩))
```

Complete the program by supplying the indicated ⟨*expression*⟩.

```
;; Exercise 3.74 (page 467-9)

;(define sense-data '(1 2 1.5 1 0.5 -0.1 -2 -3 -2 -0.5 0.2 3 4))
(define sense-data (cons-stream 1 (cons-stream -1 1)))

(define (sign-change-detector curr prev)
  (cond ((and (> prev 0) (< curr 0)) -1)
        ((and (< prev 0) (> curr 0))  1)
        (else 0)))

(define (make-zero-crossings input-stream last-value)
  (cons-stream
   (sign-change-detector
    (stream-car input-stream)
    last-value)
   (make-zero-crossings
    (stream-cdr input-stream)
    (stream-car input-stream))))
```

```
(define zero-crossings
  (make-zero-crossings sense-data -1))

(stream-ref zero-crossings 0) ;  1
(stream-ref zero-crossings 1) ; -1

(define zero-crossings2
  (stream-map sign-change-detector
              sense-data
              (cons-stream -1 sense-data)))

(stream-ref zero-crossings2 0) ;  1
(stream-ref zero-crossings2 1) ; -1
```

Exercise 3.77: The `integral` procedure used above was analogous to the "implicit" definition of the infinite stream of integers in Section 3.5.2. Alternatively, we can give a definition of `integral` that is more like `integers-starting-from` (also in Section 3.5.2):

```
(define (integral integrand initial-value dt)
  (cons-stream
   initial-value
   (if (stream-null? integrand)
       the-empty-stream
       (integral (stream-cdr integrand)
                 (+ (* dt (stream-car integrand))
                    initial-value)
                 dt))))
```

When used in systems with loops, this procedure has the same problem as does our original version of `integral`. Modify the procedure so that it expects the `integrand` as a delayed argument and hence can be used in the `solve` procedure shown above.

```
;; Exercise 3.77 (page 473)

;; code from the book

(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream
     initial-value
     (let ((integrand (force delayed-integrand)))
       (add-streams (scale-stream integrand dt) int))))
  int)

(define (solve f y0 dt)
  (define y (integral (delay dy) y0 dt))
  (define dy (stream-map f y))
  y)

(stream-ref (solve (lambda (x) x) 1 0.001) 1000)
; y: undefined;
; cannot use before initialization
```

```
;; original
(define (integral integrand initial-value dt)
  (cons-stream
   initial-value
   (if (stream-null? integrand)
       the-empty-stream
       (integral (stream-cdr integrand)
                 (+ (* dt (stream-car integrand))
                    initial-value)
                 dt))))
```

```
;; modified
(define (integral delayed-integrand initial-value dt)
  (cons-stream
   initial-value
   (let ((integrand (force delayed-integrand)))
     (if (stream-null? integrand)
         the-empty-stream
         (integral (delay (stream-cdr integrand))
                   (+ (* dt (stream-car integrand))
                      initial-value)
                   dt)))))
```

Normal-order evaluation

The examples in this section illustrate how the explicit use of `delay` and `force` provides great programming flexibility, but the same examples also show how this can make our programs more complex. Our new `integral` procedure, for instance, gives us the power to model systems with loops, but we must now remember that `integral` should be called with a delayed integrand, and every procedure that uses `integral` must be aware of this. In effect, we have created two classes of procedures: ordinary procedures and procedures that take delayed arguments. In general, creating separate classes of procedures forces us to create separate classes of higher-order procedures as well.[72]

[72]This is a small reflection, in Lisp, of the difficulties that conventional strongly typed languages such as Pascal have in coping with higher-order procedures. In such languages, the programmer must specify the data types of the arguments and the result of each procedure: number, logical value, sequence, and so on. Consequently, we could not express an abstraction such as "map a given procedure `proc` over all the elements in a sequence" by a single higher-order procedure such as `stream-map`. Rather, we would need a different mapping procedure for each different combination of argument and result data types that might be specified for a `proc`. Maintaining a practical notion of "data type" in the presence of higher-order procedures raises many difficult issues. One way of dealing with this problem is illustrated by the language ML (Gordon et al. 1979), whose "polymorphic data types" include templates for higher-order transformations between data types. Moreover, data types for most procedures in ML are never explicitly declared by the programmer. Instead, ML includes a *type-inferencing* mechanism that uses information in the environment to deduce the data types for newly defined procedures.

One way to avoid the need for two different classes of procedures is to make all procedures take delayed arguments. We could adopt a model of evaluation in which all arguments to procedures are automatically delayed and arguments are forced only when they are actually needed (for example, when they are required by a primitive operation). This would transform our language to use normal-order evaluation, which we first described when we introduced the substitution model for evaluation in Section 1.1.5. Converting to normal-order evaluation provides a uniform and elegant way to simplify the use of delayed evaluation, and this would be a natural strategy to adopt if we were concerned only with stream processing. In Section 4.2, after we have studied the evaluator, we will see how to transform our language in just this way. Unfortunately, including delays in procedure calls wreaks havoc with our ability to design programs that depend on the order of events, such as programs that use assignment, mutate data, or perform input or output.

Exercise 3.81: Exercise 3.6 discussed generalizing the random-number generator to allow one to reset the random-number sequence so as to produce repeatable sequences of "random" numbers. Produce a stream formulation of this same generator that operates on an input stream of requests to `generate` a new random number or to `reset` the sequence to a specified value and that produces the desired stream of random numbers. Don't use assignment in your solution.

```
;; Exercise 3.81 (page 481)

(define random
  (let ((a 69069)
        (c 1.0)
        (m (expt 2 32.0))
        (seed 19380110))
    (lambda (args)
      (if (list? args)
          (set! seed (cadr args))
          (set! seed (modulo (+ (* seed a) c) m)))
      (/ seed m))))

(define (random-stream input-stream)
  (stream-map random input-stream))
```

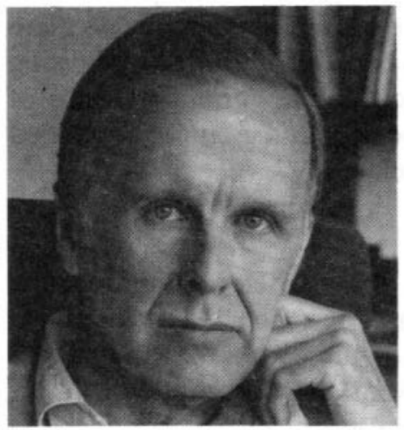
```
(define temp (cons-stream '(reset 19380110) (cons-stream 'new temp)))
(define r (random-stream temp))

;; Test
(map (lambda (x) (stream-ref r x))
     '(0 1 2 3 4 5 6 7 8 9 10))

; (0.004512283485382795
;  0.6589080521371216
;  0.004512283485382795
;  0.6589080521371216
;  0.004512283485382795
;  0.6589080521371216
;  0.004512283485382795
;  0.6589080521371216
;  0.004512283485382795
;  0.6589080521371216
;  0.004512283485382795)
```

Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of *functional programming languages*, which do not include any provision for assignment or mutable data. In such a language, all procedures implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems.[74]

[74]John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington et al. 1982.

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

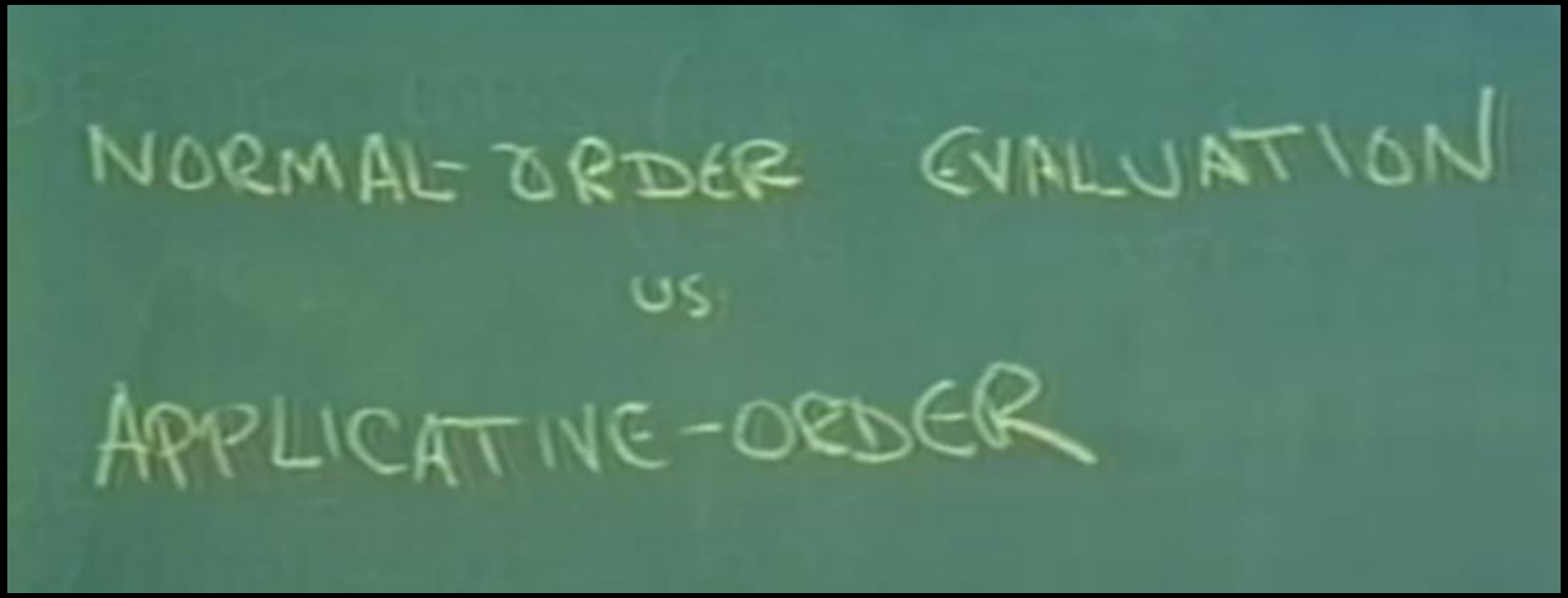
We began this chapter with the goal of building computational models whose structure matches our perception of the real world we are trying to model. We can model the world as a collection of separate, time-bound, interacting objects with state, or we can model the world as a single, timeless, stateless unity. Each view has powerful advantages, but neither view alone is completely satisfactory. A grand unification has yet to emerge.[76]

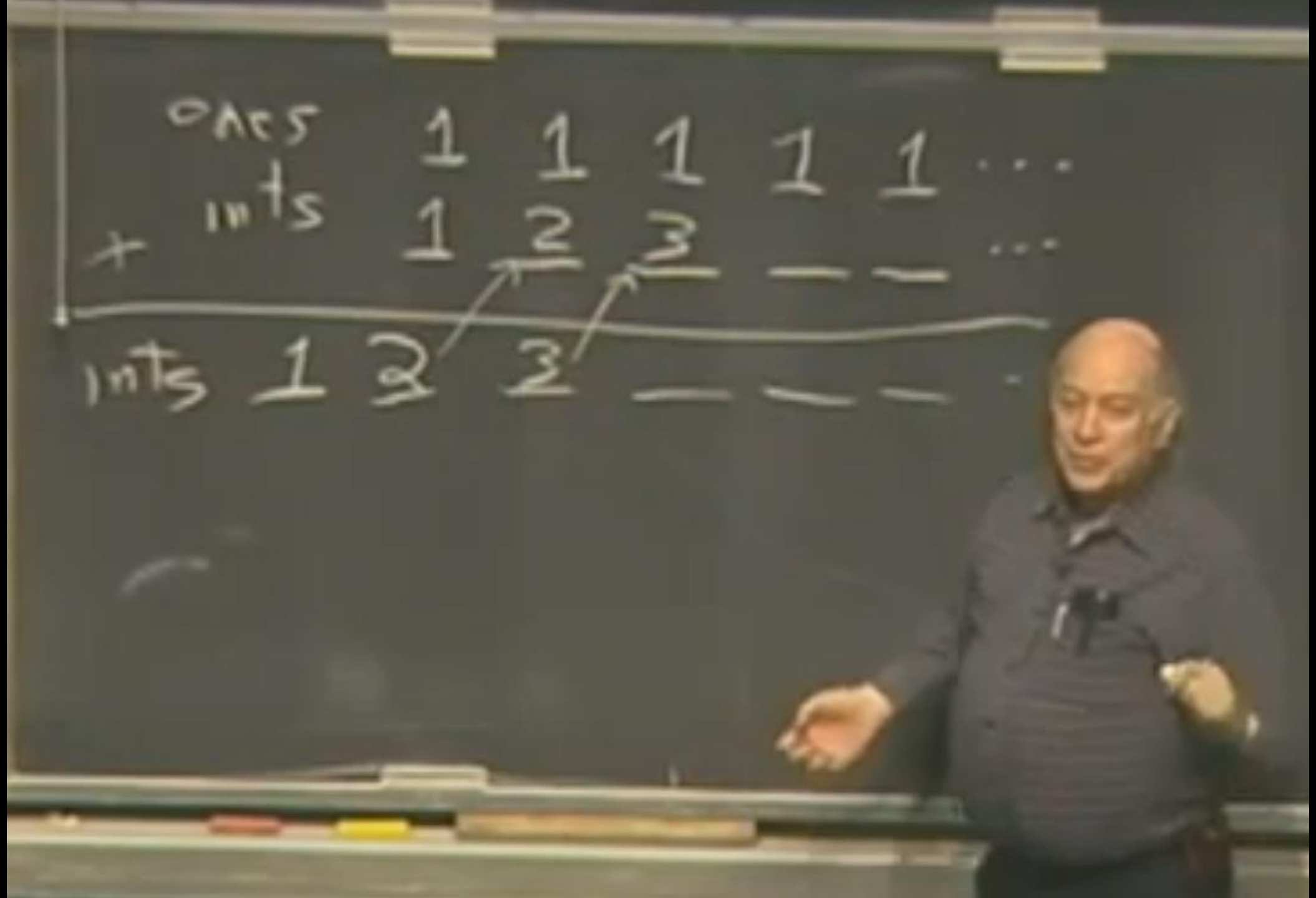
Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


Miranda™
A Non-strict Polymorphic Functional Language


NORMAL-ORDER EVALUATION
VS
APPLICATIVE-ORDER


0Arcs 1 1 1 1 1 ...
+ ints 1 2 3 _ _ ...
ints 1 2 3 _ _ _ .

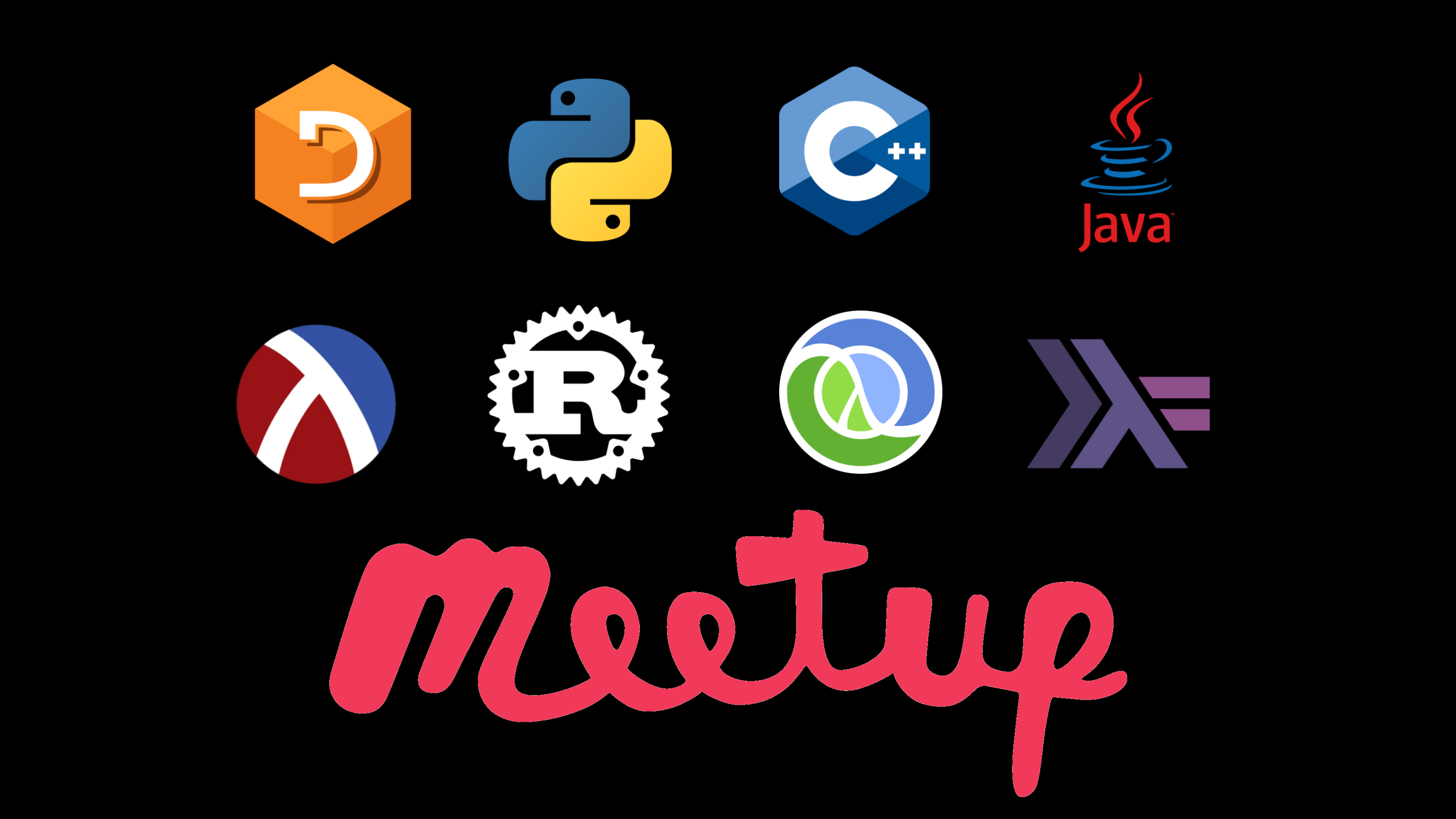
(define b
  (cons-stream 7 (interleave a b)))

a a_1 a_2 a_3

b a_1

b 7 a_1 a_1 a_2

Fermi Estimates: https://en.wikipedia.org/wiki/Fermi_estimate
Stepanov on Sieve of Eratosthenes: https://youtu.be/6tBKk0Nj7I0?t=1532
Currying: https://en.wikipedia.org/wiki/Currying
Ben on Deane Currying: https://www.youtube.com/watch?v=ojZbFIQSdI8&t=908s
Partial Application: https://en.wikipedia.org/wiki/Partial_application
Bartosz Milewski seems like he covered curry:
https://bartoszmilewski.com/2019/07/03/programming-with-universal-constructions/
Recommended Euler history lecture: https://www.youtube.com/watch?v=fEWj93XjON0
UPenn Haskell course -- compare this week's SICP with:
https://www.cis.upenn.edu/~cis194/spring13/hw/06-laziness.pdf
Taxicab numbers: https://oeis.org/A011541 https://oeis.org/A001235


Java
meetup