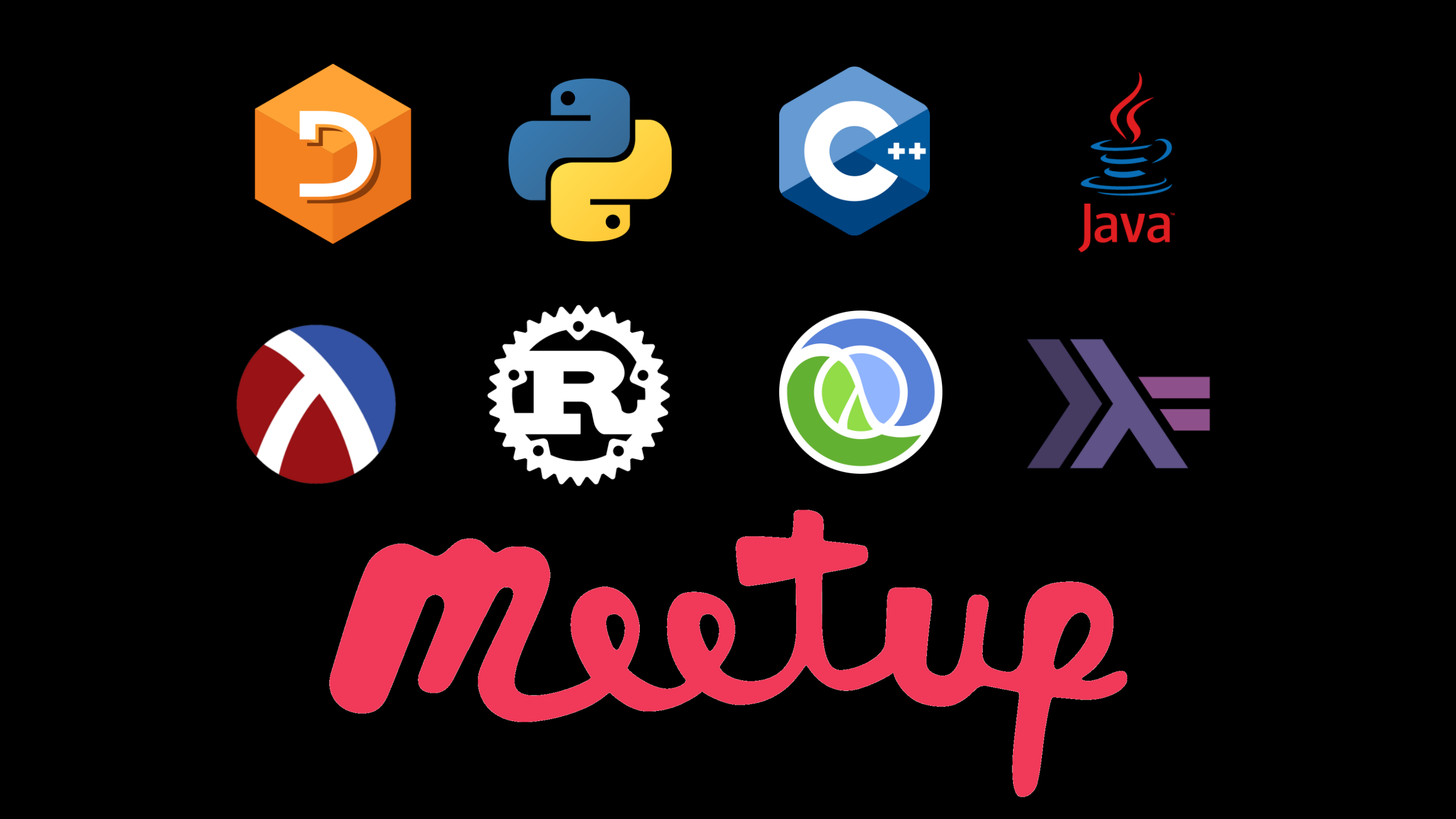
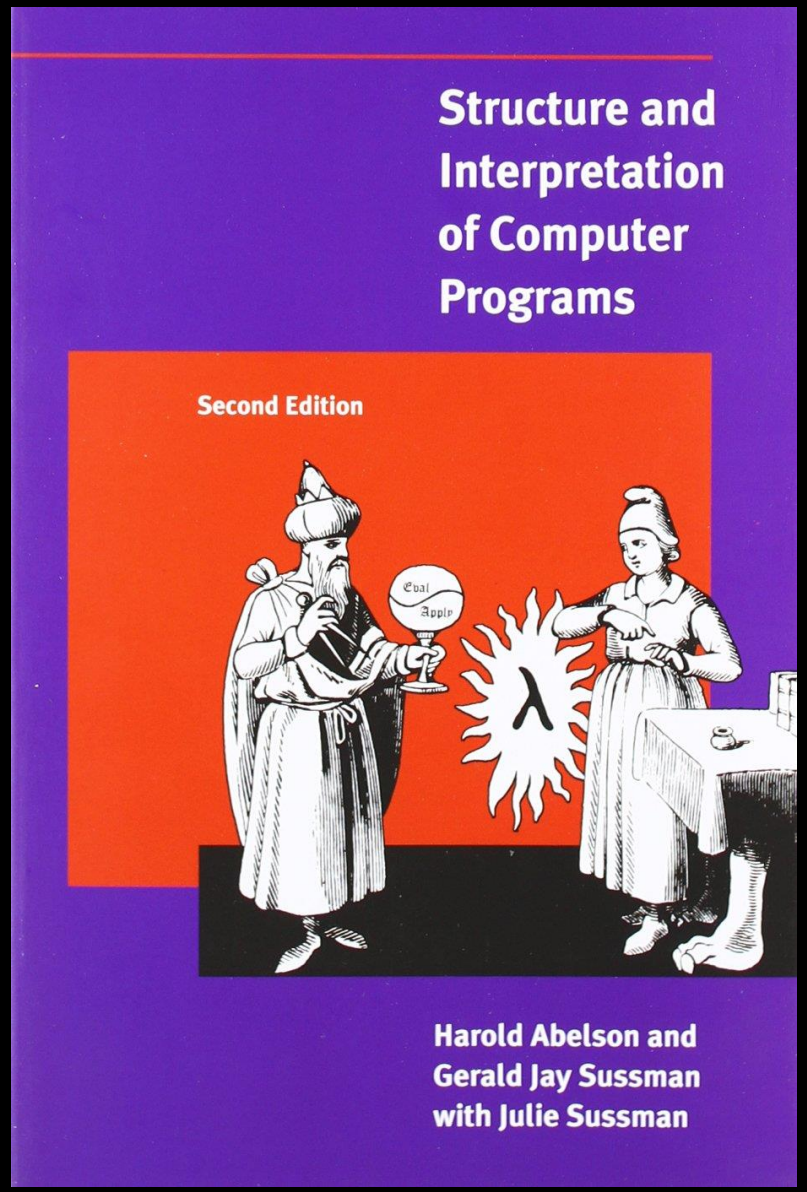

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 3.5 1/2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

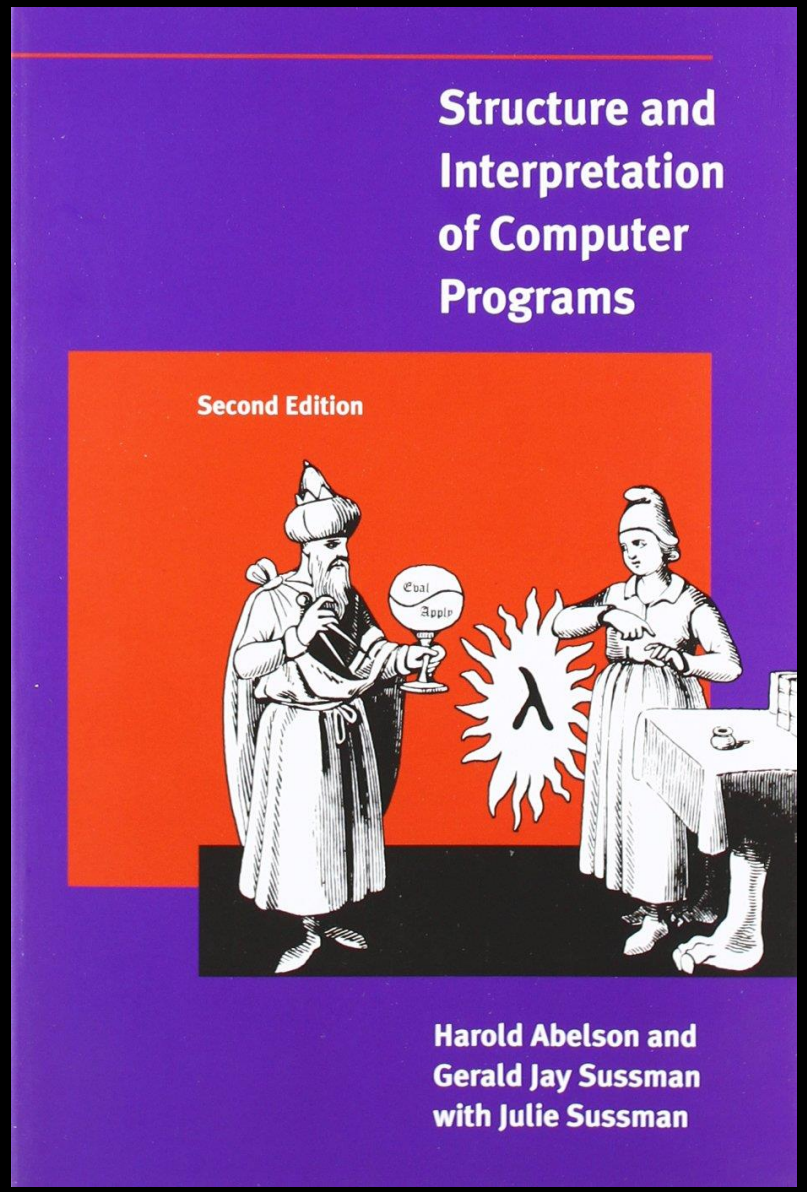
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

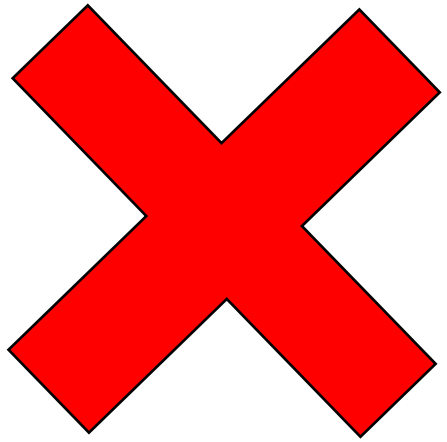
Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

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



Structure and Interpretation of Computer Programs

Chaper 3.5 1/2



Package java.util.stream Description

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Our implementation of streams will be based on a special form called `delay`. Evaluating (`delay` ⟨*exp*⟩) does not evaluate the expression ⟨*exp*⟩, but rather returns a so-called *delayed object*, which we can think of as a "promise" to evaluate ⟨*exp*⟩ at some future time. As a companion to `delay`, there is a procedure called `force` that takes a delayed object as argument and performs the evaluation—in effect, forcing the `delay` to fulfill its promise. We will see below how `delay` and `force` can be implemented, but first let us use these to construct streams.

`(cons-stream ⟨a⟩ ⟨b⟩)`

is equivalent to

`(cons ⟨a⟩ (delay ⟨b⟩))`

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

Implementing `delay` and `force`

Although `delay` and `force` may seem like mysterious operations, their implementation is really quite straightforward. `delay` must package an expression so that it can be evaluated later on demand, and we can accomplish this simply by treating the expression as the body of a procedure. `delay` can be a special form such that

```
(delay ⟨exp⟩)
```

is syntactic sugar for

```
(lambda () ⟨exp⟩)
```

`force` simply calls the procedure (of no arguments) produced by `delay`, so we can implement `force` as a procedure:

```
(define (force delayed-object) (delayed-object))
```

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
             (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

The inefficiency in using lists becomes painfully apparent if we use the sequence paradigm to compute the second prime in the interval from 10,000 to 1,000,000 by evaluating the expression

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 3 [1..10]
[1,2,3]
```

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 3 [1..10]
[1,2,3]
Prelude> take 3 [1..]
[1,2,3]
```

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 3 [1..10]
[1,2,3]
Prelude> take 3 [1..]
[1,2,3]
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17
```

Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\conorhoekstra> ghci
GHCi, version 7.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 3 [1..10]
[1,2,3]
Prelude> take 3 [1..]
[1,2,3]
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,5
2,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100
,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,1
37,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173
,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,208,209,2
10,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246
,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,2
83,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319
,320,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,3
56,357,358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,386,387,388,389,390,391,392
,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,4
29,430,431,432,433,434,435,436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465
,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,500,501,5
02,503,504,505,506,507,508,509,510,511,512,513,514,515,516,517,518,519,520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,537,538
,539,540,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,568,569,570,571,572,573,574,5
75,576,577,578,579,580,581,582,583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600,601,602,603,604,605,606,607,608,609,610,611
,612,613,614,615,616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,6
48,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,680,681,682,683,684
,685,686,687,688,689,690,691,692,693,694,695,696,697,698,699,700,701,702,703,704,705,706,707,708,709,710,711,712,713,714,715,716,717,718,719,720,7
21,722,723,724,725,726,727,728,729,730,731,732,733,734,735,736,737,738,739,740,741,742,743,744,745,746,747,748,749,750,751,752,753,754,755,756,757
,758,759,760,761,762,763,764,765,766,767,768,769,770,771,772,773,774,775,776,777,778,779,780,781,782,783,784,785,786,787,788,789,790,791,792,793,7
94,795,796,797,798,799,800,801,802,803,804,805,806,807,808,809,810,811,812,813,814,815,816,817,818,819,820,821,822,823,824,825,826,827,828,829,830
,831,832,833,834,835,836,837,838,839,840,841,842,843,844,845,846,847,848,849,850,851,852,853,854,855,856,857,858,859,860,861,862,863,864,865,866,8
```

```
(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))
(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))
(define (stream-for-each proc s)
  (if (stream-null? s)
      'done
      (begin (proc (stream-car s))
             (stream-for-each proc (stream-cdr s)))))
```

```
(define (display-stream s)
  (stream-for-each display-line s))
(define (display-line x) (newline) (display x))
```

```
(stream-car
 (stream-cdr
  (stream-filter prime?
                 (stream-enumerate-interval
                  10000 1000000))))
```

```
(define (stream-enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream
       low
       (stream-enumerate-interval (+ low 1) high))))
```

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                      (stream-filter
                       pred
                       (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

Exercise 3.50: Complete the following definition, which generalizes `stream-map` to allow procedures that take multiple arguments, analogous to `map` in Section 2.2.1, Footnote 12.

```
(define (stream-map proc . argstreams)
  (if (⟨??⟩ (car argstreams))
      the-empty-stream
      (⟨??⟩
       (apply proc (map ⟨??⟩ argstreams))
       (apply stream-map
              (cons proc (map ⟨??⟩ argstreams))))))
```

```
;; Exercise 3.50 (page 440)

(define (stream-map proc . argstreams)
  (if (stream-null? (car argstreams))
      the-empty-stream
      (cons-stream
       (apply proc (map stream-car argstreams))
       (apply stream-map
              (cons proc (map stream-cdr argstreams))))))
```

Exercise 3.51: In order to take a closer look at delayed evaluation, we will use the following procedure, which simply returns its argument after printing it:

```
(define (show x)
  (display-line x)
  x)
```

What does the interpreter print in response to evaluating each expression in the following sequence?[59]

```
(define x
  (stream-map show
              (stream-enumerate-interval 0 10)))
(stream-ref x 5)
(stream-ref x 7)
```

```
;; Exercise 3.51 (page 440-1)

(define (show x)
  (display-line x)
  x)

(define x (stream-map show (stream-enumerate-interval 0 10)))
;; 0

(stream-ref x 5)
;; 1
;; 2
;; 3
;; 4
;; 5

(stream-ref x 7)
;; 6
;; 7
```


For a look at a more exciting infinite stream, we can generalize the `no-sevens` example to construct the infinite stream of prime numbers, using a method known as the *sieve of Eratosthenes*.[60] We start with the integers beginning with 2, which is the first prime. To get the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of this stream. This leaves a stream beginning with 5, which is the next prime, and so on. In other words, we construct the primes by a sieving process, described as follows: To sieve a stream `S`, form a stream whose first element is the first element of `S` and the rest of which is obtained by filtering all multiples of the first element of `S` out of the rest of `S` and sieving the result. This process is readily described in terms of stream operations:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))


(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))
(define primes (sieve (integers-starting-from 2)))
```

Sieve of Eratosthenes

From Wikipedia, the free encyclopedia

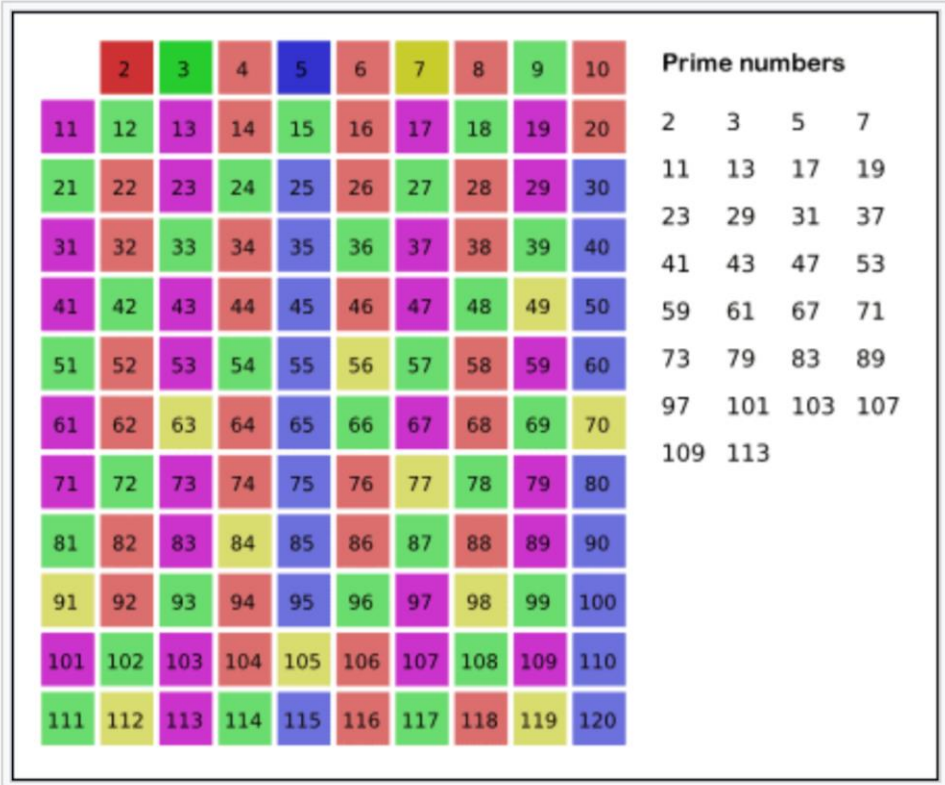
For the sculpture, see The Sieve of Eratosthenes (sculpture).

In mathematics, the **sieve of Eratosthenes** is an ancient algorithm for finding all prime numbers up to any given limit.

It does so by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with the first prime number, 2. The multiples of a given prime are generated as a sequence of numbers starting from that prime, with constant difference between them that is equal to that prime.[1] This is the sieve's key distinction from using trial division to sequentially test each candidate number for divisibility by each prime.[2]

The earliest known reference to the sieve (Ancient Greek: κόσκινον Ἐρατοσθένους, *kóskinon Eratosthénous*) is in Nicomachus of Gerasa's *Introduction to Arithmetic*,[3] which describes it and attributes it to Eratosthenes of Cyrene, a Greek mathematician.

One of a number of prime number sieves, it is one of the most efficient ways to find all of the smaller primes. It may be used to find primes in arithmetic progressions.[4]



Sieve of Eratosthenes: algorithm steps for primes below 121 (including optimization of starting from prime's square).

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))
(define integers (integers-starting-from 1))


(define (sieve stream)
  (cons-stream
   (stream-car stream)
   (sieve (stream-filter
           (lambda (x)
             (not (divisible? x (stream-car stream))))
           (stream-cdr stream)))))
(define primes (sieve (integers-starting-from 2)))
```

[60]Eratosthenes, a third-century B.C. Alexandrian Greek philosopher, is famous for giving the first accurate estimate of the circumference of the Earth, which he computed by observing shadows cast at noon on the day of the summer solstice. Eratosthenes's sieve method, although ancient, has formed the basis for special-purpose hardware "sieves" that, until recently, were the most powerful tools in existence for locating large primes. Since the 70s, however, these methods have been superseded by outgrowths of the probabilistic techniques discussed in Section 1.2.6.

Defining streams implicitly

The `integers` and `fibs` streams above were defined by specifying "generating" procedures that explicitly compute the stream elements one by one. An alternative way to specify streams is to take advantage of delayed evaluation to define streams implicitly. For example, the following expression defines the stream `ones` to be an infinite stream of ones:

```
(define ones (cons-stream 1 ones))
```

We can do more interesting things by manipulating streams with operations such as `add-streams`, which produces the elementwise sum of two given streams:[62]

```
(define (add-streams s1 s2) (stream-map + s1 s2))
```

Now we can define the integers as follows:

```
(define integers
  (cons-stream 1 (add-streams ones integers)))
```

Exercise 3.53: Without running the program, describe the elements of the stream defined by

```
(define s (cons-stream 1 (add-streams s s)))
```

```
;; Exercise 3.53 (page 447)

;; 1 2 4 8 ...
```

Exercise 3.54: Define a procedure `mul-streams`, analogous to `add-streams`, that produces the elementwise product of its two input streams. Use this together with the stream of `integers` to complete the following definition of the stream whose n^{th} element (counting from 0) is $n + 1$ factorial:

```
(define factorials
  (cons-stream 1 (mul-streams ⟨??⟩ ⟨??⟩)))
```

```
;; Exercise 3.54 (page 447-8)

;; Code from book
(define ones (cons-stream 1 ones))
(define (add-streams s1 s2) (stream-map + s1 s2))
(define integers (cons-stream 1 (add-streams ones integers)))

;; Solution
(define (mul-streams s1 s2) (stream-map * s1 s2))
(define factorials (cons-stream 1 (mul-streams factorials (stream-cdr integers))))

;; Test
(stream-ref factorials 9) ; 3628800
```

Exercise 3.55: Define a procedure `partial-sums` that takes as argument a stream S and returns the stream whose elements are S_0, S_0+S_1, $S_0+S_1+S_2$, ... For example, `(partial-sums integers)` should be the stream 1, 3, 6, 10, 15, ...

Conor Hoekstra
@code_report

Hoogle Translate

scan

Elixir	**scan**	Enum	Doc
F#	**scan**	Seq	Doc
Kotlin	**scan**	collections	Doc
APL	**\ (scan)**	-	Doc
Scala	**scan**	various	Doc
Rust	**scan**	trait.Iterator	Doc
q	**scan**	-	Doc
C++	**inclusive_scan**	<numeric>	Doc
CUDA	**inclusive_scan**	Thrust	Doc
J	**\ (prefix)**	-	Doc
Python	**accumulate**	itertools	Doc
Clojure	**reductions**	core	Doc
Haskell	**scanl1**	Data.List	Doc
C++	**partial_sum**	<numeric>	Doc
D	**cumulativeFold**	algorithm.iteration	Doc

```
;; Exercise 3.55 (page 448)

(define (partial-sums S)
  (cons-stream (stream-car S)
               (add-streams (partial-sums S) (stream-cdr S))))

;; Test
(stream-ref (partial-sums (stream-enumerate-interval 1 5)) 4) ; 15
```

Exercise 3.56: A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, let us call the required stream of numbers `S` and notice the following facts about it.

- `S` begins with 1.
- The elements of `(scale-stream S 2)` are also elements of `S`.
- The same is true for `(scale-stream S 3)` and `(scale-stream 5 S)`.
- These are all the elements of `S`.

Now all we have to do is combine elements from these sources. For this we define a procedure `merge` that combines two ordered streams into one ordered result stream, eliminating repetitions:

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (let ((s1car (stream-car s1))
               (s2car (stream-car s2)))
           (cond ((< s1car s2car)
                  (cons-stream
                   s1car
                   (merge (stream-cdr s1) s2)))
                 ((> s1car s2car)
                  (cons-stream
                   s2car
                   (merge s1 (stream-cdr s2))))
                 (else
                  (cons-stream
                   s1car
                   (merge (stream-cdr s1)
                          (stream-cdr s2)))))))))
```

Then the required stream may be constructed with `merge`, as follows:

```
(define S (cons-stream 1 (merge ⟨??⟩ ⟨??⟩)))
```

Fill in the missing expressions in the places marked ⟨??⟩ above.

```
(define S (cons-stream 1 (merge (scale-stream S 2)
                                (merge (scale-stream S 3)
                                       (scale-stream S 5)))))

;; Test
(map (lambda (x) (stream-ref S x))
     '(0 1 2 3 4 5 6 7 8 9 10)) ; (1 2 3 4 5 6 8 9 10 12 15)
```

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots.$$

```
;; Pi from alternating series

(define (pi-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (pi-summands (+ n 2)))))

(define pi-stream
  (scale-stream (partial-sums (pi-summands 1)) 4))

;; Pi with Euler acceleration

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

(define (euler-transform s)
  (let ((s0 (stream-ref s 0))  ; S n -1
        (s1 (stream-ref s 1))  ; S n
        (s2 (stream-ref s 2))) ; S n+1
    (cons-stream (- s2 (/ (square (- s2 s1))
                          (+ s0 (* -2 s1) s2)))
                 (euler-transform (stream-cdr s)))))
```

```
;; Pi with super-"tableu" acceleration

(define (make-tableau transform s)
  (cons-stream s (make-tableau transform (transform s))))

(define (accelerated-sequence transform s)
  (stream-map stream-car (make-tableau transform s)))

(stream-ref pi-stream 1000)       ; 3.1425916543395442

(stream-ref (euler-transform
             pi-stream) 1000)     ; 3.1415926538383

(stream-ref (accelerated-sequence
             euler-transform
             pi-stream) 9)        ; 3.141592653589795
```

Exercise 3.65: Use the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

to compute three sequences of approximations to the natural logarithm of 2, in the same way we did above for π. How rapidly do these sequences converge?

```
;; Exercise 3.65 (page 459)

;; ln(2) from alternating series

(define (ln2-summands n)
  (cons-stream (/ 1.0 n)
               (stream-map - (ln2-summands (+ n 1)))))

(define ln2-stream
  (partial-sums (ln2-summands 1)))

(stream-ref ln2-stream 1000)       ; 0.6936464315588232

(stream-ref (euler-transform
             ln2-stream) 1000)     ; 0.6931471806840143

(stream-ref (accelerated-sequence
             euler-transform
             ln2-stream) 9)        ; 0.6931471805599454
```

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream (stream-car s1)
                   (interleave s2 (stream-cdr s1)))))
```

```
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) (stream-cdr t)))))
```

Exercise 3.67: Modify the `pairs` procedure so that `(pairs integers integers)` will produce the stream of *all* pairs of integers (i, j) (without the condition $i \leq j$). Hint: You will need to mix in an additional stream.

```
;; Exercise 3.67 (page 462)

;; orginal
(define (pairs-orig s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs-orig (stream-cdr s) (stream-cdr t)))))
```

```
;; Exercise 3.67 (page 462)

;; modified
(define (pairs s t)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                (stream-cdr t))
    (pairs (stream-cdr s) t))))
```

Exercise 3.70: It would be nice to be able to generate streams in which the pairs appear in some useful order, rather than in the order that results from an *ad hoc* interleaving process. We can use a technique similar to the `merge` procedure of Exercise 3.56, if we define a way to say that one pair of integers is "less than" another. One way to do this is to define a "weighting function" $W(i, j)$ and stipulate that (i_1, j_1) is less than (i_2, j_2) if $W(i_1, j_1) < W(i_2, j_2)$. Write a procedure `merge-weighted` that is like `merge`, except that `merge-weighted` takes an additional argument `weight`, which is a procedure that computes the weight of a pair, and is used to determine the order in which elements should appear in the resulting merged stream.[69] Using this, generalize `pairs` to a procedure `weighted-pairs` that takes two streams, together with a procedure that computes a weighting function, and generates the stream of pairs, ordered according to weight. Use your procedure to generate

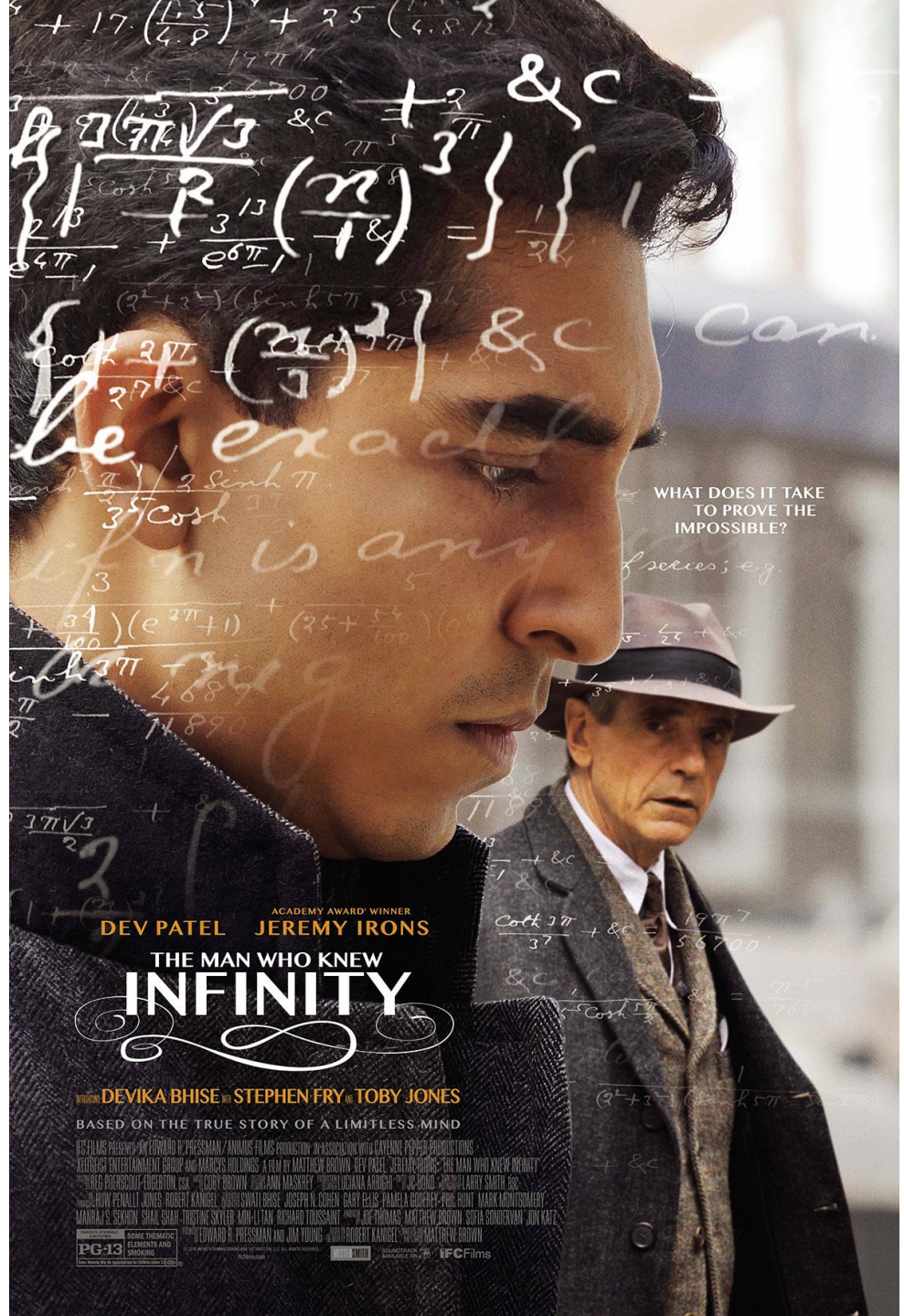
a. the stream of all pairs of positive integers (i, j) with $i \leq j$ ordered according to the sum $i + j$,

```
;; Exercise 3.70 (page 474)

(define (merge-weighted s t weight)
  (cond ((stream-null? s) t)
        ((stream-null? t) s)
        (else
         (let ((s-car (stream-car s))
               (t-car (stream-car t)))
           (cond ((< (weight s-car) (weight t-car))
                  (cons-stream s-car (merge-weighted (stream-cdr s) t weight)))
                 ((< (weight t-car) (weight s-car))
                  (cons-stream t-car (merge-weighted (stream-cdr t) s weight)))
                 (else (cons-stream
                        s-car
                        (cons-stream t-car (merge-weighted (stream-cdr s)
                                                           (stream-cdr t) weight)))))))))
```

```
(define (weighted-pairs s t weight)
  (cons-stream
   (list (stream-car s) (stream-car t))
   (merge-weighted (stream-map (lambda (x) (list (stream-car s) x)) (stream-cdr t))
                   (weighted-pairs (stream-cdr s) (stream-cdr t) weight)
                   weight)))

;; a)
(define wp (weighted-pairs integers integers (lambda (p) (apply + p))))
(map (lambda (x) (stream-ref wp x)) '(0 1 2 3 4 5 6 7 8 9))
;; ((1 1) (1 2) (1 3) (2 2) (1 4) (2 3) (1 5) (2 4) (3 3) (2 5))
```

Exercise 3.71: Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.[70] Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$ (see Exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a procedure to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

[70]To quote from G. H. Hardy's obituary of Ramanujan (Hardy 1921): "It was Mr. Littlewood (I believe) who remarked that 'every positive integer was one of his friends.' I remember once going to see him when he was lying ill at Putney. I had ridden in taxi-cab No. 1729, and remarked that the number seemed to me a rather dull one, and that I hoped it was not an unfavorable omen. 'No,' he replied, 'it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.' " The trick of using weighted pairs to generate the Ramanujan numbers was shown to us by Charles Leiserson.


WHAT DOES IT TAKE TO PROVE THE IMPOSSIBLE?
ACADEMY AWARD® WINNER
DEV PATEL
JEREMY IRONS
THE MAN WHO KNEW
INFINITY
DEVIKA BHISE
STEPHEN FRY
TOBY JONES
BASED ON THE TRUE STORY OF A LIMITLESS MIND
PG-13
SOME THEMATIC ELEMENTS AND SMOKING
IFCFilms

Exercise 3.71: Numbers that can be expressed as the sum of two cubes in more than one way are sometimes called *Ramanujan numbers*, in honor of the mathematician Srinivasa Ramanujan.[70] Ordered streams of pairs provide an elegant solution to the problem of computing these numbers. To find a number that can be written as the sum of two cubes in two different ways, we need only generate the stream of pairs of integers (i, j) weighted according to the sum $i^3 + j^3$ (see Exercise 3.70), then search the stream for two consecutive pairs with the same weight. Write a procedure to generate the Ramanujan numbers. The first such number is 1,729. What are the next five?

```
(define (ramanujan-numbers n)
  (let* ((cube (lambda (x) (* x x x)))
         (ram-w (lambda (p) (apply + (map cube p)))))
    (define (iter s left)
      (cond ((= left 0) 'done)
            (else
             (let ((a (stream-car s))
                   (b (stream-car (stream-cdr s))))
               (if (= (ram-w a) (ram-w b))
                   (begin (display (list (ram-w a) a b))
                          (newline)
                          (iter (stream-cdr s) (- left 1)))
                   (iter (stream-cdr s) left))))))
    (iter (weighted-pairs integers integers ram-w) n)))

;; Test
(ramanujan-numbers 5)
;; (1729 (9 10) (1 12))
;; (4104 (9 15) (2 16))
;; (13832 (18 20) (2 24))
;; (20683 (19 24) (10 27))
;; (32832 (18 30) (4 32))
;; done
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

"This sense of **conventional interfaces** – so you can start putting a lot of things together – the **stream** is the uniform data structure that supports that…"

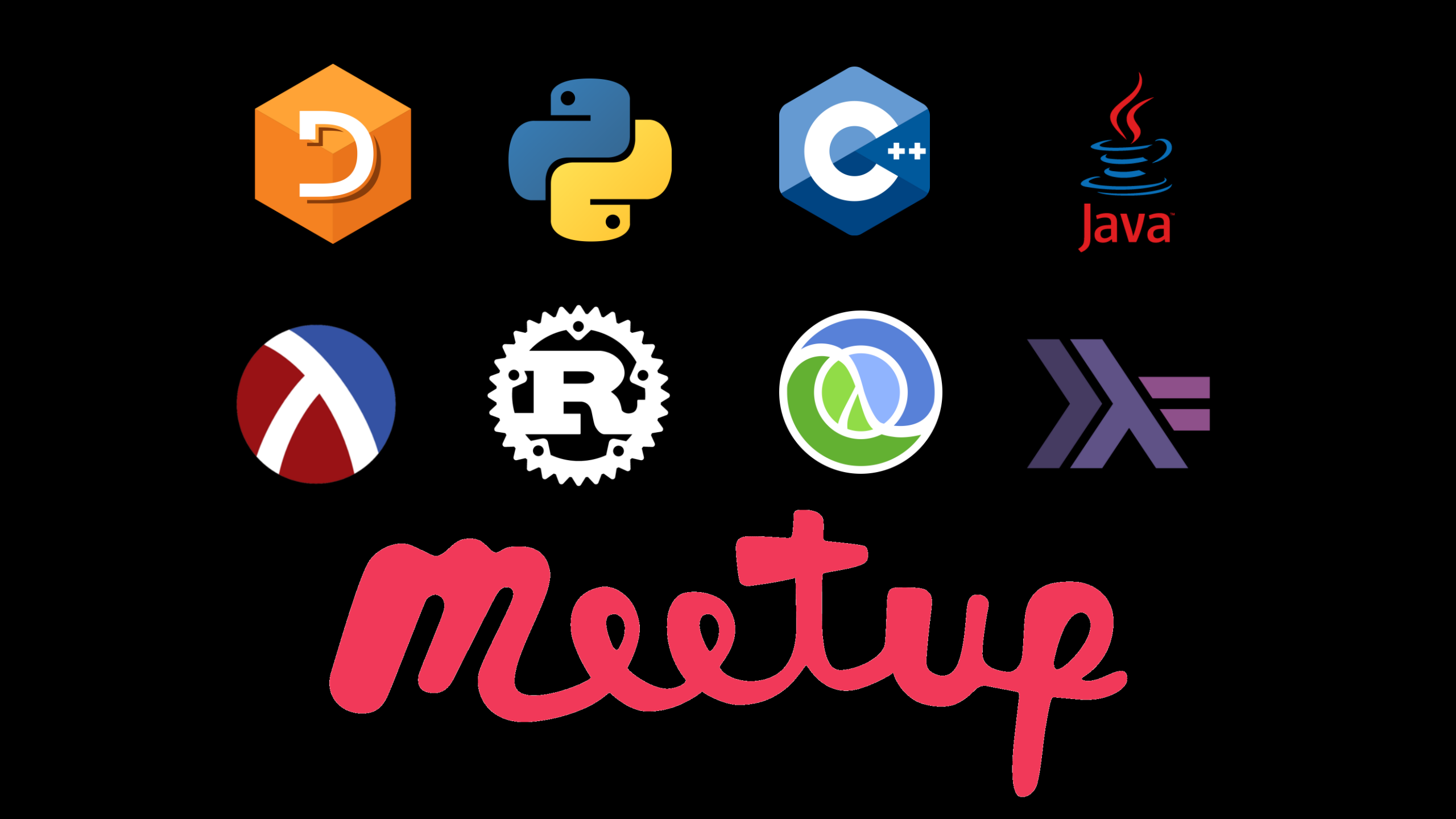
Hal Abelson

Lecture 6A, SICP

"This is very much like **APL** by the way.
APL has very much the same idea except
in APL instead of this stream you have arrays
and vectors and a lot of the **power of APL** is
exactly the same reason of the power of this"

Hal Abelson

Lecture 6A, SICP


Java
meetup