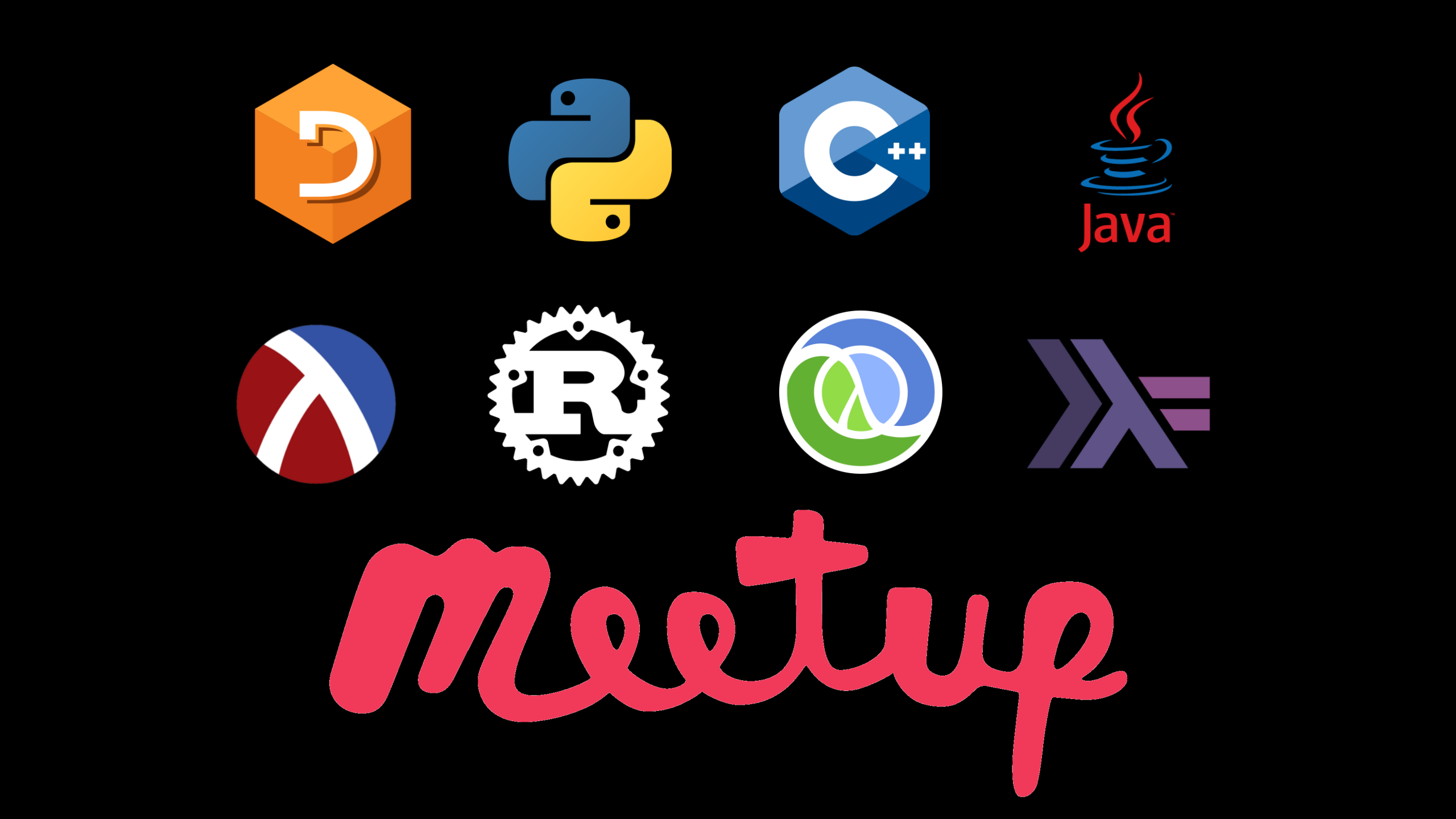
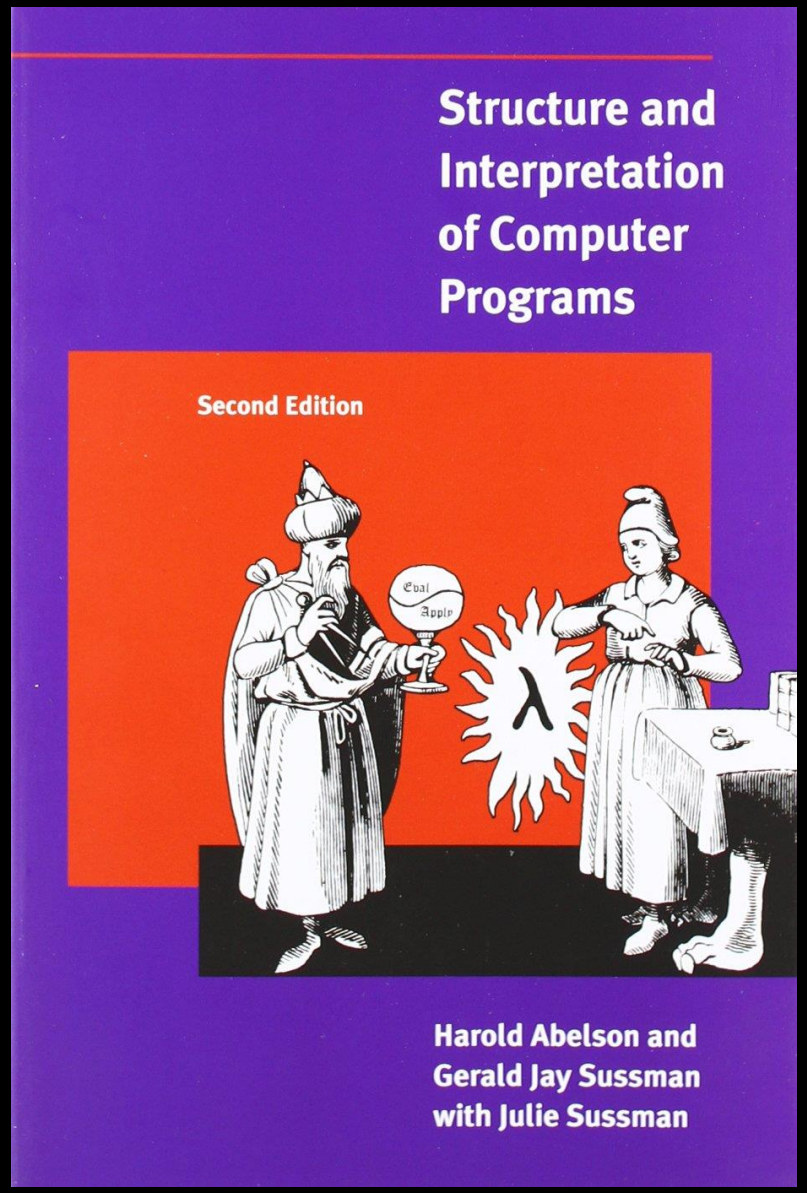

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 3.1

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

Kite (geometry)

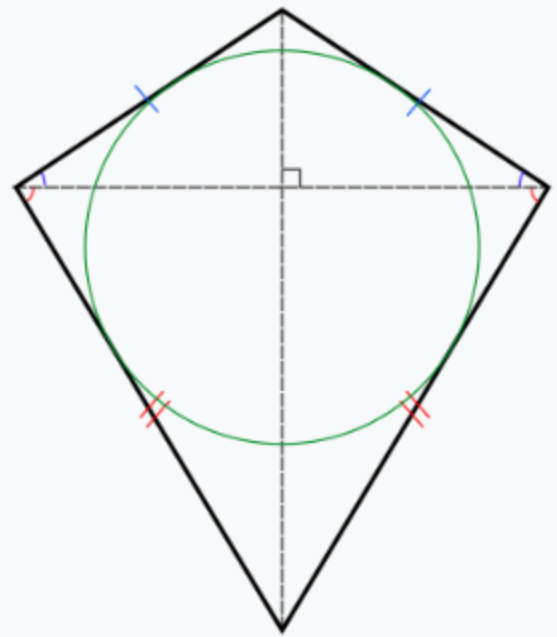
From Wikipedia, the free encyclopedia

In Euclidean geometry, a **kite** is a quadrilateral whose four sides can be grouped into two pairs of equal-length sides that are adjacent to each other. In contrast, a parallelogram also has two pairs of equal-length sides, but they are opposite to each other rather than adjacent. Kite quadrilaterals are named for the wind-blown, flying kites, which often have this shape and which are in turn named for a bird. Kites are also known as **deltoids**, but the word "deltoid" may also refer to a deltoid curve, an unrelated geometric object.

A kite, as defined above, may be either convex or concave, but the word "kite" is often restricted to the convex variety. A concave kite is sometimes called a "dart" or "arrowhead", and is a type of pseudotriangle.

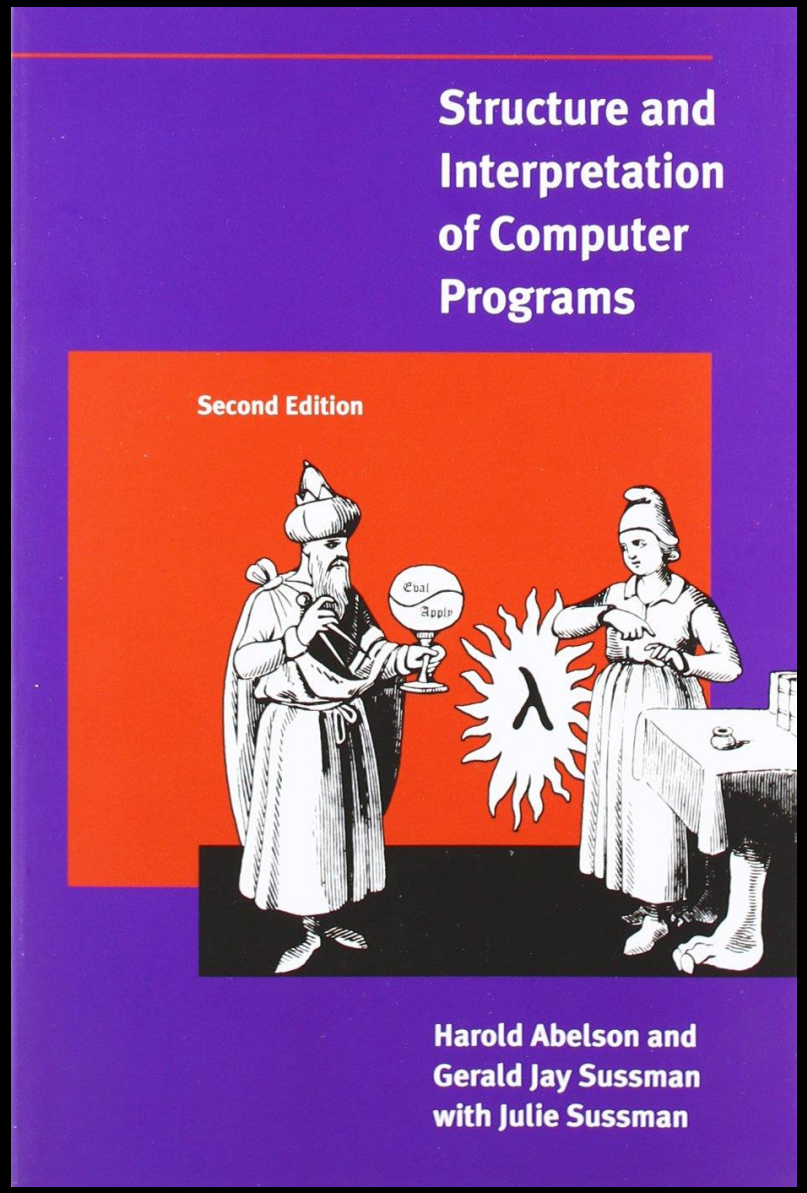
Contents [hide]

Kite



A kite, showing its pairs of equal length sides and its inscribed circle.

Type	Quadrilateral
Edges and vertices	4
Symmetry group	D_1 (*)
Dual polygon	Isosceles trapezoid



Structure and Interpretation of Computer Programs

Chaper 3.1

THE PRECEDING CHAPTERS introduced the basic elements from which programs are made. We saw how primitive procedures and primitive data are combined to construct compound entities, and we learned that abstraction is vital in helping us to cope with the complexity of large systems. But these tools are not sufficient for designing programs. Effective program synthesis also requires organizational principles that can guide us in formulating the overall design of a program. In particular, we need strategies to help us structure large systems so that they will be *modular*, that is, so that they can be divided "naturally" into coherent parts that can be separately developed and maintained.

object-based approach and the stream-processing approach

Both the object-based approach and the stream-processing approach raise significant linguistic issues in programming. With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to abandon our old substitution model of computation (Section 1.1.5) in favor of a more mechanistic but less theoretically tractable *environment model* of computation.

```
(withdraw 25)
75
(withdraw 25)
50
(withdraw 60)
"Insufficient funds"
(withdraw 15)
35
```

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

Decrementing `balance` is accomplished by the expression

```
(set! balance (- balance amount))
```

This uses the `set!` special form, whose syntax is

```
(set! ⟨name⟩ ⟨new-value⟩)
```

[2]The value of a `set!` expression is implementation-dependent. `set!` should be used only for its effect, not for its value.

The name `set!` reflects a naming convention used in Scheme: Operations that change the values of variables (or that change data structures, as we will see in Section 3.3) are given names that end with an exclamation point. This is similar to the convention of designating predicates by names that end with a question mark.

[3]We have already used `begin` implicitly in our programs, because in Scheme the body of a procedure can be a sequence of expressions. Also, the ⟨*consequent*⟩ part of each clause in a `cond` expression can be a sequence of expressions rather than a single expression.

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds"))))
```

[4]In programming-language jargon, the variable `balance` is said to be *encapsulated* within the `new-withdraw` procedure. Encapsulation reflects the general system-design principle known as the *hiding principle*: One can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a "need to know."

```scheme
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Exercise 3.1: An *accumulator* is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure `make-accumulator` that generates accumulators, each maintaining an independent sum. The input to `make-accumulator` should specify the initial value of the sum; for example

```
(define A (make-accumulator 5))
(A 10)
15
(A 10)
25
```

```
;; Exercise 3.1 (page 303-4)

(require rackunit)

(define (make-accumulator init)
  (let ((sum init))
    (λ (val) (begin (set! sum (+ sum val))
                    sum))))

(define A (make-accumulator 5))

(check-equal? (A 10) 15)
(check-equal? (A 10) 25)
```

Exercise 3.2: In software-testing applications, it is useful to be able to count the number of times a given procedure is called during the course of a computation. Write a procedure `make-monitored` that takes as input a procedure, `f`, that itself takes one input. The result returned by `make-monitored` is a third procedure, say `mf`, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to `mf` is the special symbol `how-many-calls?`, then `mf` returns the value of the counter. If the input is the special symbol `reset-count`, then `mf` resets the counter to zero. For any other input, `mf` returns the result of calling `f` on that input and increments the counter. For instance, we could make a monitored version of the `sqrt` procedure:

```
(define s (make-monitored sqrt))
(s 100)
10
(s 'how-many-calls?)
1
```

```
;; Exercise 3.2 (page 304)

;; only works for one parameter function f
(define (make-monitored f)
  (let ((count 0))
    (λ (arg-or-symbol)
      (cond ((eq? arg-or-symbol 'how-many-calls) count)
            ((eq? arg-or-symbol 'reset) (set! count 0))
            (else (set! count (+ count 1))
                  (f arg-or-symbol))))))

(define s (make-monitored sqrt))
(check-equal? (s 'how-many-calls) 0)
(check-equal? (s 100) 10)
(check-equal? (s 'how-many-calls) 1)
```

```scheme
;; works for variadic number of parameters
(define (make-monitored f)
  (let ((count 0))
    (λ (head . tail)
      (cond ((eq? head 'how-many-calls) count)
            ((eq? head 'reset) (set! count 0))
            (else (set! count (+ count 1))
                  (apply f (cons head tail)))))))

(define s (make-monitored sqrt))
(check-equal? (s 'how-many-calls) 0)
(check-equal? (s 100) 10)
(check-equal? (s 'how-many-calls) 1)

(define p (make-monitored +))
(check-equal? (p 'how-many-calls) 0)
(check-equal? (p 1 2) 3)
(check-equal? (p 'how-many-calls) 1)
```

Exercise 3.3: Modify the `make-account` procedure so that it creates password-protected accounts. That is, `make-account` should take a symbol as an additional argument, as in

```
(define acc (make-account 100 'secret-password))
```

The resulting account object should process a request only if it is accompanied by the password with which the account was created, and should otherwise return a complaint:

```
((acc 'secret-password 'withdraw) 40)
60
((acc 'some-other-password 'deposit) 50)
"Incorrect password"
```

```
;; Exercise 3.3 (page 304-5)

;; original from book

(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT"
                       m))))
  dispatch)
```

```scheme
;; modified

(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (dispatch input-password m)
    (cond ((not (eq? password input-password)) (λ (_) "Incorrect password"))
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)

(define acc (make-account 100 '123abc))
(check-equal? ((acc '123abc 'withdraw) 40) 60)
(check-equal? ((acc '456xyz 'withdraw) 40) "Incorrect password")
(check-equal? ((acc '123abc 'deposit)  10) 70)
```

Exercise 3.4: Modify the `make-account` procedure of Exercise 3.3 by adding another local state variable so that, if an account is accessed more than seven consecutive times with an incorrect password, it invokes the procedure `call-the-cops`.

```scheme
;; Exercise 3.4 (page 305)

(define (make-account balance password)
  (let ((fail-count 0))
    (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 (set! fail-count 0)
                 balance)
          "Insufficient funds"))
    (define (deposit amount)
      (set! balance (+ balance amount))
      (set! fail-count 0)
      balance)
    (define (dispatch input-password m)
      (cond ((> fail-count 7) (λ (_) "CALL THE COPS"))
            ((not (eq? password input-password))
             (λ (_) (begin (set! fail-count (add1 fail-count))
                           "Incorrect password")))
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            (else (error "Unknown request: MAKE-ACCOUNT" m))))
    dispatch))
```

```
(define rand (let ((x random-init))
               (lambda ()
                 (set! x (rand-update x))
                 x)))
```

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else
            (iter (- trials-remaining 1)
                  trials-passed))))
   (iter trials 0))
```

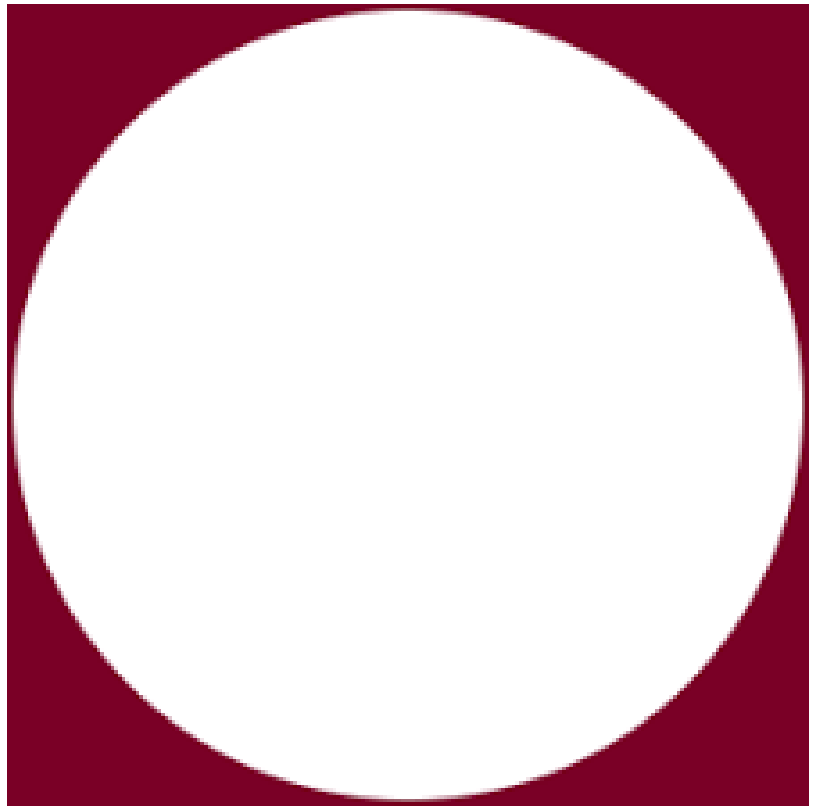
```
(define (estimate-pi trials)
  (sqrt (/ 6 (random-gcd-test trials random-init))))
(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0)
               (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2))
              (else
               (iter (- trials-remaining 1)
                     trials-passed
                     x2))))))
  (iter trials 0 initial-x))
```

Exercise 3.5: *Monte Carlo integration* is a method of estimating definite integrals by means of Monte Carlo simulation. Consider computing the area of a region of space described by a predicate $P(x, y)$ that is true for points (x, y) in the region and false for points not in the region. For example, the region contained within a circle of radius 3 centered at (5, 7) is described by the predicate that tests whether $(x - 5)^2 + (y - 7)^2 \leq 3^2$. To estimate the area of the region described by such a predicate, begin by choosing a rectangle that contains the region. For example, a rectangle with diagonally opposite corners at (2, 4) and (8, 10) contains the circle above. The desired integral is the area of that portion of the rectangle that lies in the region. We can estimate the integral by picking, at random, points (x, y) that lie in the rectangle, and testing $P(x, y)$ for each point to determine whether the point lies in the region. If we try this with many points, then the fraction of points that fall in the region should give an estimate of the proportion of the rectangle that lies in the region. Hence, multiplying this fraction by the area of the entire rectangle should produce an estimate of the integral.

Implement Monte Carlo integration as a procedure `estimate-integral` that takes as arguments a predicate `P`, upper and lower bounds `x1`, `x2`, `y1`, and `y2` for the rectangle, and the number of trials to perform in order to produce the estimate. Your procedure should use the same `monte-carlo` procedure that was used above to estimate π. Use your `estimate-integral` to produce an estimate of π by measuring the area of a unit circle.

You will find it useful to have a procedure that returns a number chosen at random from a given range. The following `random-in-range` procedure implements this in terms of the `random` procedure used in Section 1.2.6, which returns a nonnegative number less than its input.[8]

```
(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (random range))))
```



```
;; Exercise 3.5 (page 309-11)

(require threading)
(require algorithms) ; generate, sum

(define (random-in-range low high)
  (let ((range (- high low)))
    (+ low (* range (/ (random 10000) 10000.0)))))

(define (estimate-interval P x1 x2 y1 y2 n)
  (let ((rect-area (* (- x2 x1) (- y2 y1)))
        (random-point (λ () (list (random-in-range x1 x2) (random-in-range y1 y2)))))
    (~>> (generate n random-point)
         (map (λ (p) (if (apply P p) 1.0 0)))
         (sum)
         (* (/ rect-area n)))))
```

```
(define (sq x) (* x x))
(estimate-interval (λ (x y) (< (+ (sq (- x 5)) (sq (- y 7))) 9))
                   2 8 4 10 100000)

(println "PI estimate")
(/ (estimate-interval (λ (x y) (< (+ (sq (- x 5)) (sq (- y 7))) 9))
                   2 8 4 10 100000) 9)

;; 28.18404
;; "PI estimate"
;; 3.1414400000000002
```

Programming without any use of assignments, as we did throughout the first two chapters of this book, is accordingly known as *functional programming*.

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
```

A language that supports the concept that "equals can be substituted for equals" in an expression without changing the value of the expression is said to be *referentially transparent*. Referential transparency is violated when we include `set!` in our computer language. This makes it tricky to determine when we can simplify expressions by substituting equivalent expressions. Consequently, reasoning about programs that use assignment becomes drastically more difficult.

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*.

In contrast to functional programming, programming that makes extensive use of assignment is known as *imperative programming*.

[11]In view of this, it is ironic that introductory programming is most often taught in a highly imperative style. This may be a vestige of a belief, common throughout the 1960s and 1970s, that programs that call procedures must inherently be less efficient than programs that perform assignments. (Steele 1977 debunks this argument.) Alternatively it may reflect a view that step-by-step assignment is easier for beginners to visualize than procedure call. Whatever the reason, it often saddles beginning programmers with "should I set this variable before or after that one" concerns that can complicate programming and obscure the important ideas.

The complexity of imperative programs becomes even worse if we consider applications in which several processes execute concurrently. We will return to this in Section 3.4. First, however, we will address the issue of providing a computational model for expressions that involve assignment, and explore the uses of objects with local state in designing simulations.

Exercise 3.7: Consider the bank account objects created by `make-account`, with the password modification described in Exercise 3.3. Suppose that our banking system requires the ability to make joint accounts. Define a procedure `make-joint` that accomplishes this. `make-joint` should take three arguments. The first is a password-protected account. The second argument must match the password with which the account was defined in order for the `make-joint` operation to proceed. The third argument is a new password. `make-joint` is to create an additional access to the original account using the new password.

```scheme
;; Exercise 3.7 (page 319-20)

(define (make-account balance password)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (define (make-joint joint-password)
    (dispatch joint-password))
  (define (dispatch account-password)
    (λ (input-password m)
      (cond ((not (eq? account-password input-password)) (λ (_) "Incorrect password"))
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'make-joint) make-joint)
            (else (error "Unknown request: MAKE-ACCOUNT" m)))))
  (dispatch password))

(define acc (make-account 100 '123abc))
(check-equal? ((acc '123abc 'withdraw) 40) 60)
(check-equal? ((acc '456xyz 'withdraw) 40) "Incorrect password")
(check-equal? ((acc '123abc 'deposit)  10) 70)

(define acc2 ((acc '123abc 'make-joint) 'canIjoin))
(check-equal? ((acc2 'canIjoin 'withdraw) 70) 0)
```

Exercise 3.8: When we defined the evaluation model in Section 1.1.3, we said that the first step in evaluating an expression is to evaluate its subexpressions. But we never specified the order in which the subexpressions should be evaluated (e.g., left to right or right to left). When we introduce assignment, the order in which the arguments to a procedure are evaluated can make a difference to the result. Define a simple procedure `f` such that evaluating

```
(+ (f 0) (f 1))
```

will return 0 if the arguments to + are evaluated from left to right but will return 1 if the arguments are evaluated from right to left.

```
;; Exercise 3.8 (page 320)

(define i 0)
(define vals '(-0.5 0.5))

(define (f x)
  (set! i (+ i x))
  (list-ref vals i))

(check-equal? (+ (f 0) (f 1)) 0.0)
(set! i 0)
(check-equal? (+ (f 1) (f 0)) 1.0)
```

Structure & Interpretation of Computer Programs

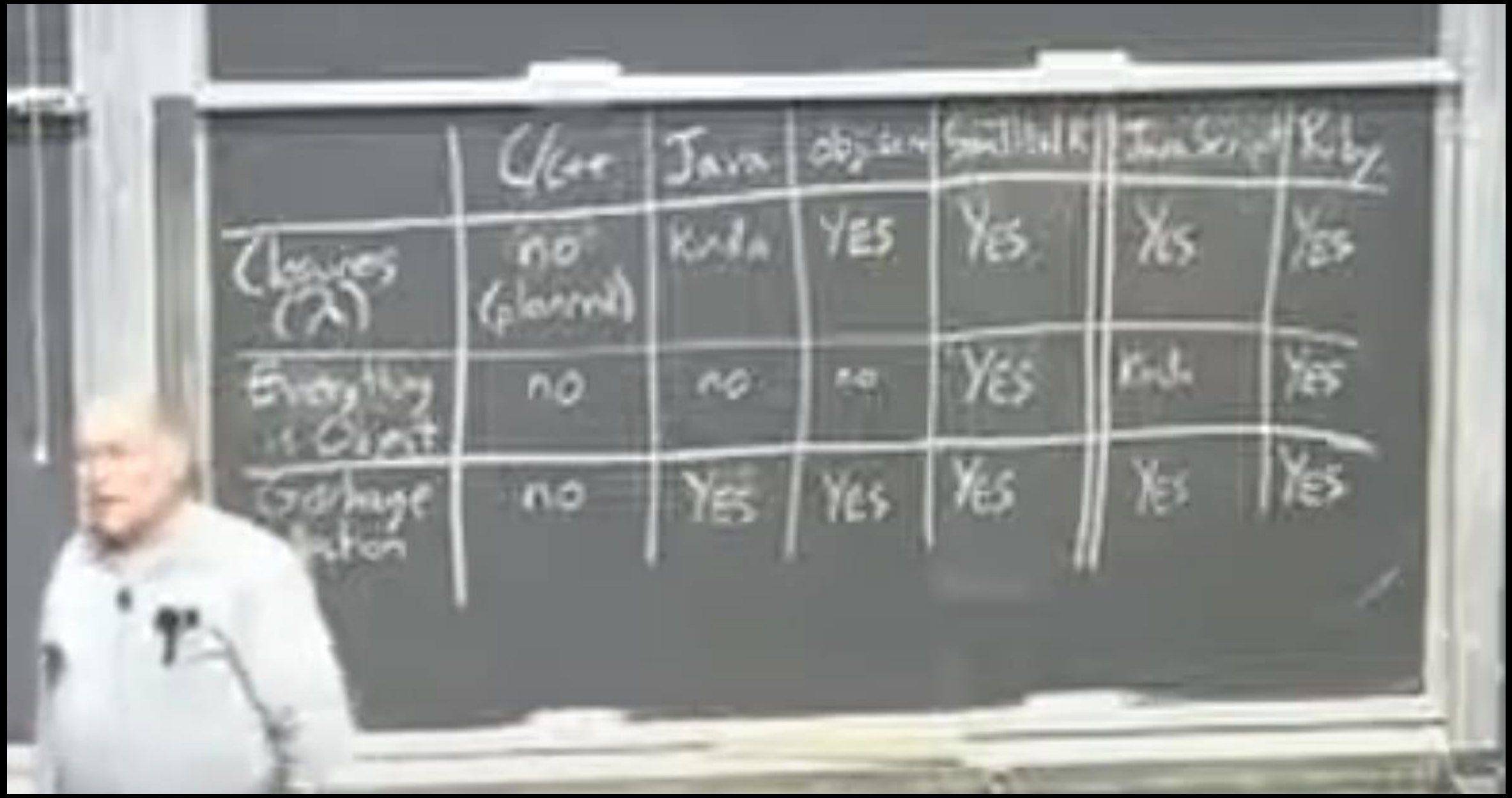
Harold Abelson

Gerald Jay Sussman

Assignment, State, & Side-effects

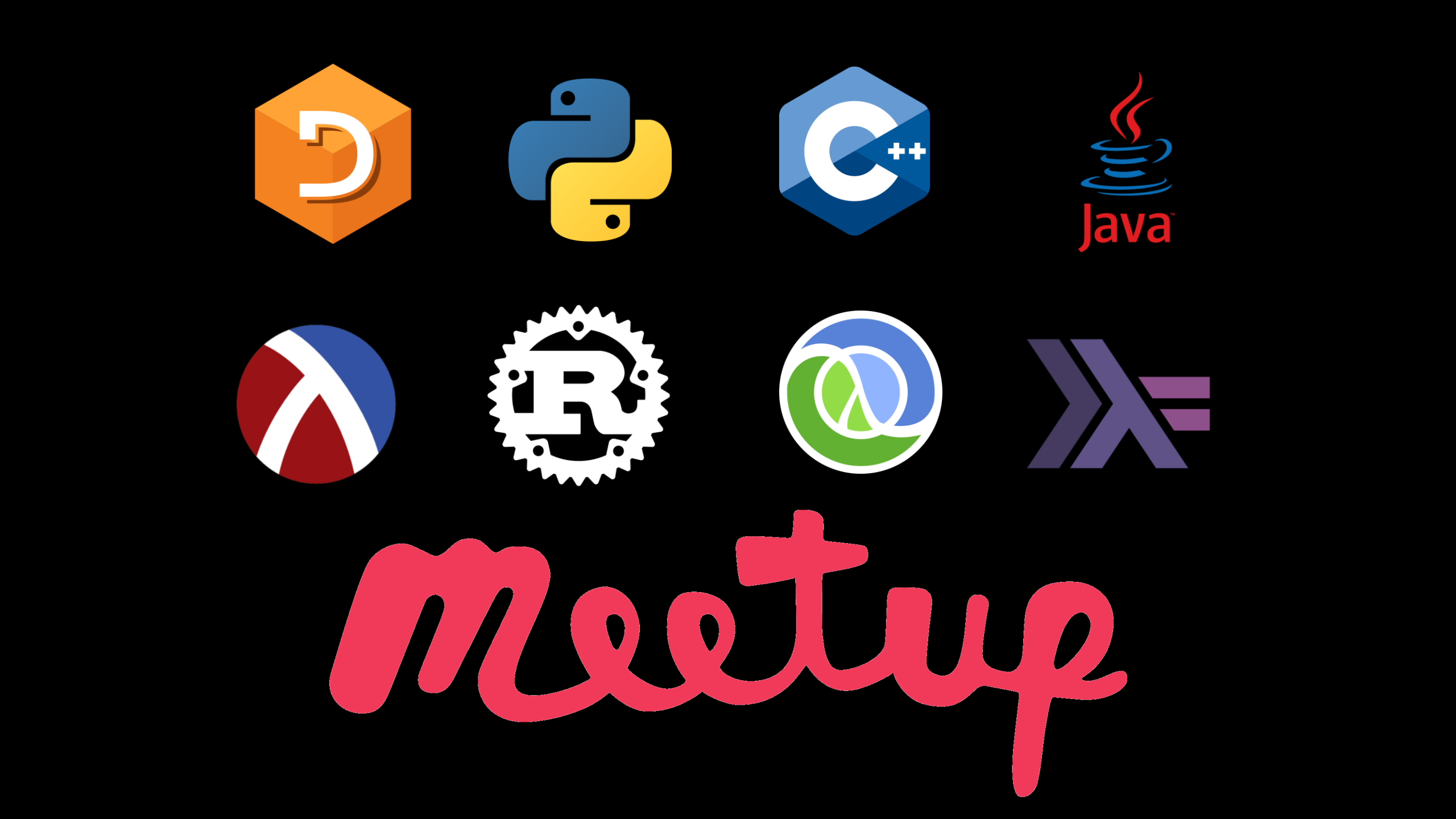
Lecture 5A

Part 3-Text Section 3.1 & 3.2


C++
Java
Ruby
Closures (λ)
no (planned)
YES
YES
YES
YES
no
no
YES
YES
no
YES
YES
YES
YES

						
Closures (λ)	NO (planned)	kind of	YES	YES	YES	YES
Everything is an object (uniform reference)	NO	NO	NO	YES	kind of	YES
Garbage Collection	NO	YES	YES	YES	YES	YES

	C++	Java	Erlang	Smalltalk	JS	Ruby
Closures (λ)	YES	YES	YES	YES	YES	YES
Everything is an object (uniform reference)	NO	NO	NO	YES	kind of	YES
Garbage Collection	NO	YES	YES	YES	YES	YES


Java
meetup