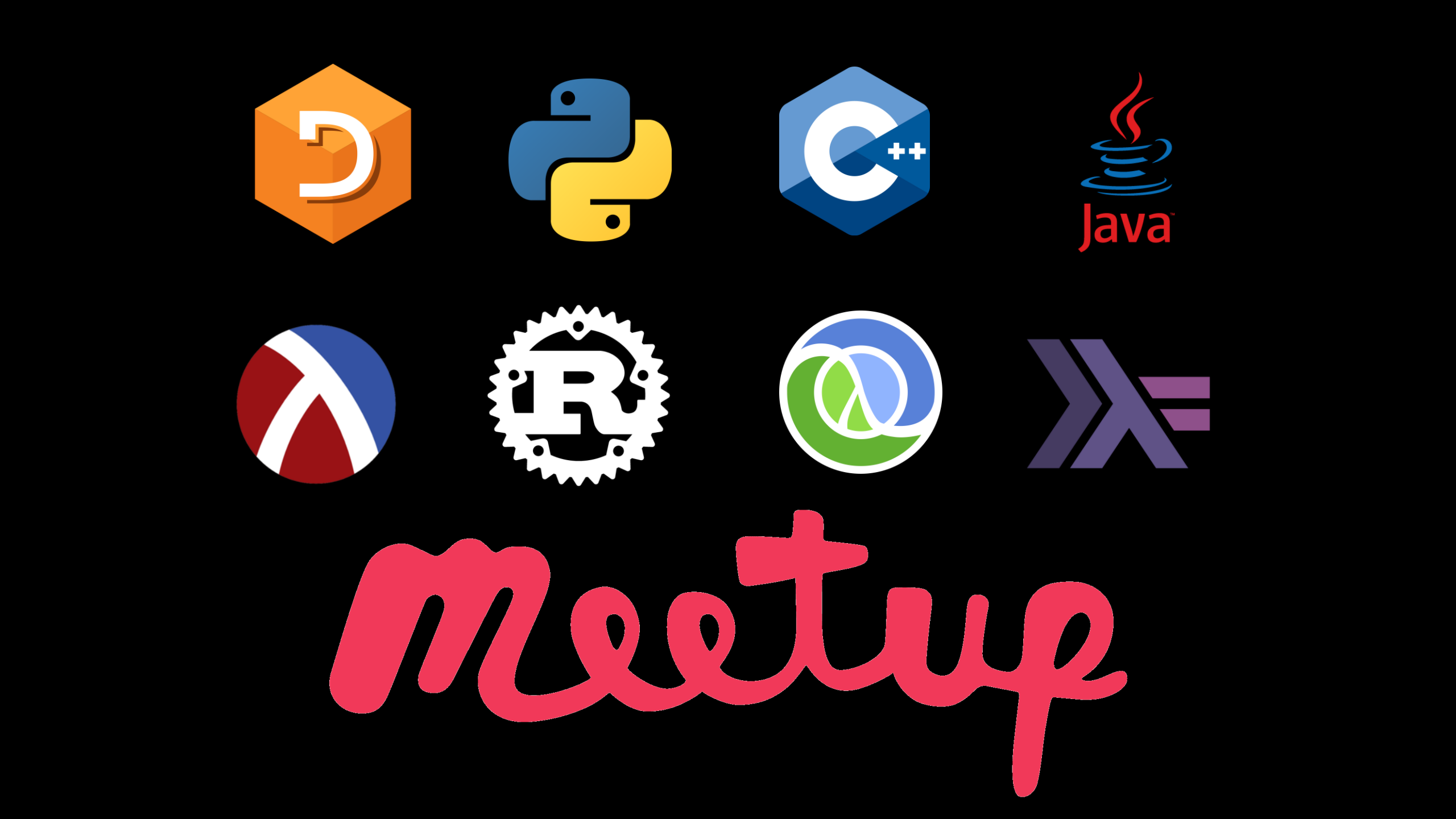
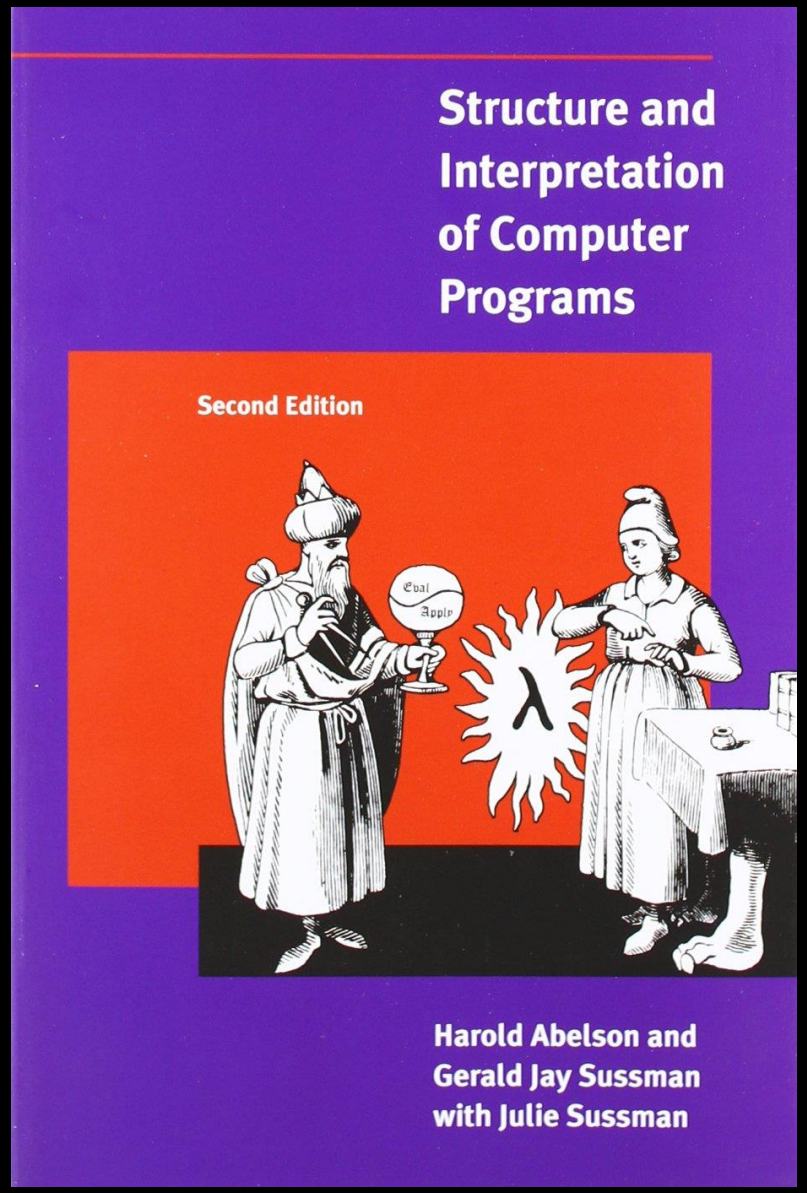

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 2.4

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

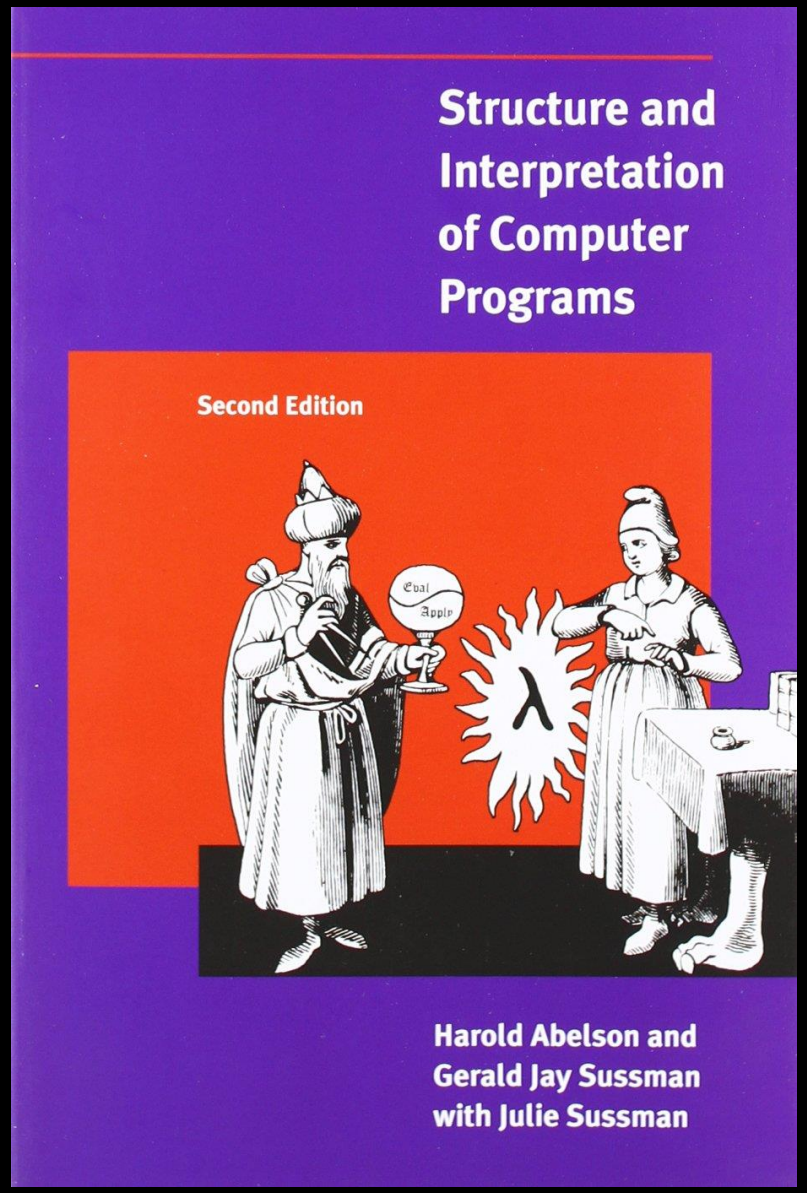
Following

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 2.4

We have introduced data abstraction, a methodology for structuring systems in such a way that much of a program can be specified independent of the choices involved in implementing the data objects that the program manipulates.

But this kind of data abstraction is not yet powerful enough, because it may not always make sense to speak of "the underlying representation" for a data object.

For one thing, there might be more than one useful representation for a data object, and we might like to design systems that can deal with multiple representations.

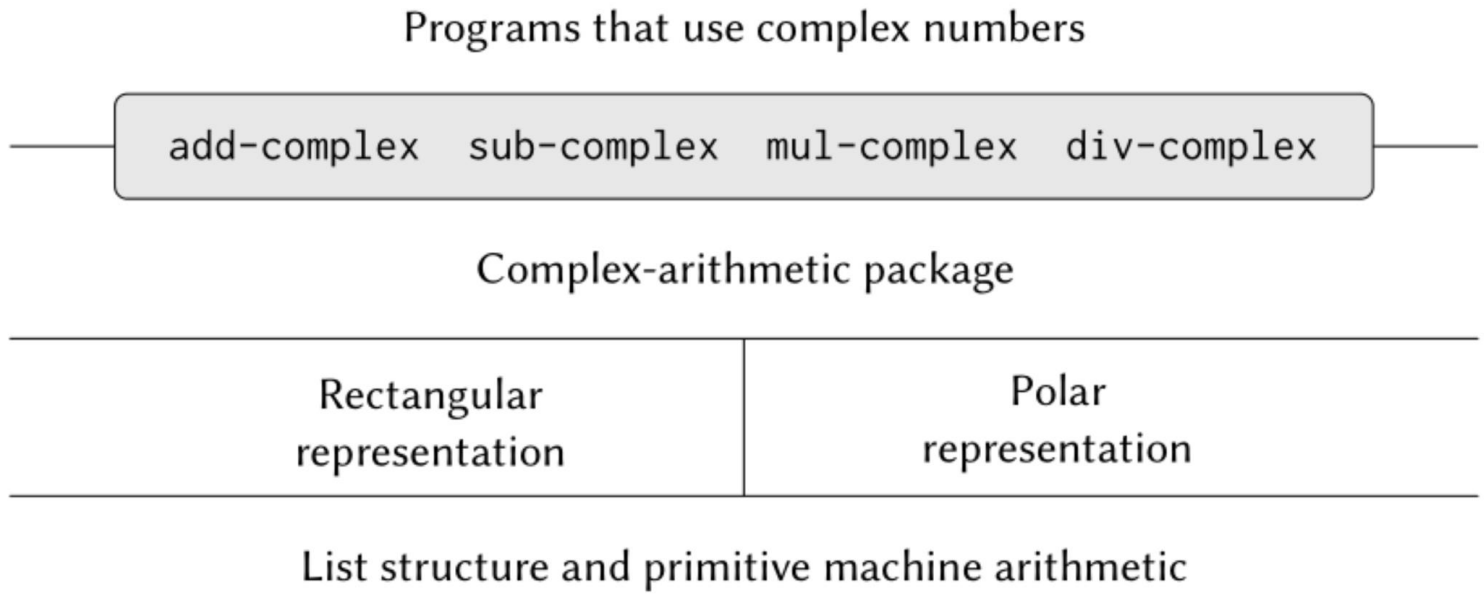


Figure 2.19: Data-abstraction barriers in the complex-number system.

```
(make-from-real-imag (real-part z) (imag-part z))
```

and

```
(make-from-mag-ang (magnitude z) (angle z))
```

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z))
           (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

```
(define (real-part z) (* (magnitude z) (cos (angle z))))
(define (imag-part z) (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
```

2.4.2 Tagged data

```scheme
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z) (eq? (type-tag z) 'polar))
```

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z))))
```

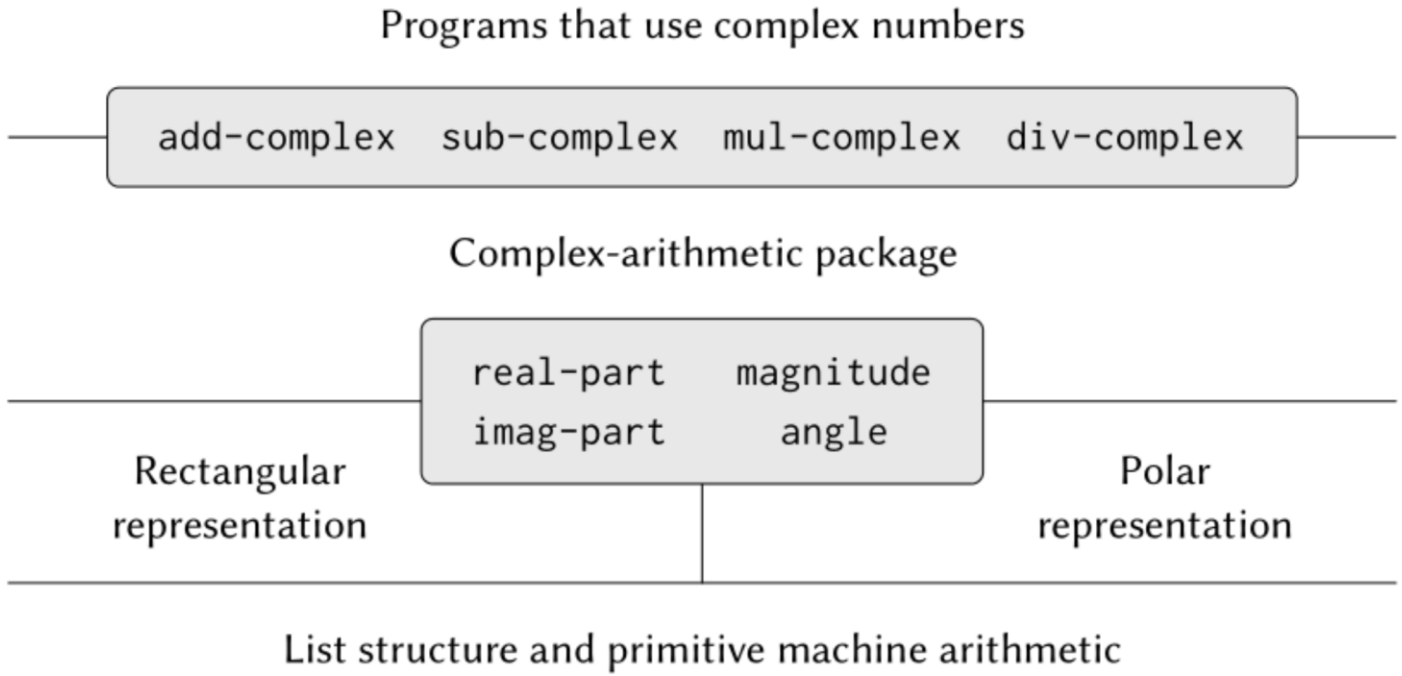


Figure 2.21: Structure of the generic complex-arithmetic system.

2.4.3 Data-Directed Programming and Additivity

The general strategy of checking the type of a datum and calling an appropriate procedure is called *dispatching on type*. This is a powerful strategy for obtaining modularity in system design. On the other hand, implementing the dispatch as in Section 2.4.2 has two significant weaknesses. One weakness is that the generic interface procedures (`real-part`, `imag-part`, `magnitude`, and `angle`) must know about all the different representations. For instance, suppose we wanted to incorporate a new representation for complex numbers into our complex-number system. We would need to identify this new representation with a type, and then add a clause to each of the generic interface procedures to check for the new type and apply the appropriate selector for that representation.

Another weakness of the technique is that even though the individual representations can be designed separately, we must guarantee that no two procedures in the entire system have the same name. This is why Ben and Alyssa had to change the names of their original procedures from Section 2.4.1.

To implement this plan, assume that we have two procedures, `put` and `get`, for manipulating the operation-and-type table:

- `(put` ⟨*op*⟩ ⟨*type*⟩ ⟨*item*⟩`)` installs the ⟨*item*⟩ in the table, indexed by the ⟨*op*⟩ and the ⟨*type*⟩.
- `(get` ⟨*op*⟩ ⟨*type*⟩`)` looks up the ⟨*op*⟩, ⟨*type*⟩ entry in the table and returns the item found there. If no item is found, `get` returns false.

Exercise 2.73: Section 2.3.2 described a program that performs symbolic differentiation:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum (make-product
                    (multiplier exp)
                    (deriv (multiplicand exp) var))
                   (make-product
                    (deriv (multiplier exp) var)
                    (multiplicand exp))))
        (else (error "unknown expression type: DERIV" exp))))
```

We can regard this program as performing a dispatch on the type of the expression to be differentiated. In this situation the "type tag" of the datum is the algebraic operator symbol (such as +) and the operation being performed is `deriv`. We can transform this program into data-directed style by rewriting the basic derivative procedure as

```
;; revised (using "data-directed" style)

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp))
               (operands exp) var))))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

a. Explain what was done above. Why can't we assimilate the predicates `number?` and `variable?` into the data-directed dispatch?

```
;; a) What was done? dispath of sum?, product?, etc were consolidated into
;;      the else ((get 'deriv ... clause of the conditional expression. Lookup
;;      is now done based on the operator using a selector
;;    Why can't we assimilate the predicates number? and variable into the
;;      data-directed dispatch? No "tag" to dispatch on
```

b. Write the procedures for derivatives of sums and products, and the auxiliary code required to install them in the table used by the program above.

c. Choose any additional differentiation rule that you like, such as the one for exponents (Exercise 2.56), and install it in this data-directed system.

```
;; Code from 2.3

(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (=number? exp num) (and (number? exp) (= exp num)))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2))
         (+ a1 a2))
        (else (list '+ a1 a2))))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))

(define (make-exponentiation base exp)
  (cond ((=number? base 1) 1)
        ((=number? exp 1) base)
        ((=number? exp 0) 1)
        (else (list '^ base exp))))

(define base cadr)
(define exponent caddr)
```

```
;; Modified code

(define (deriv-sum expr var)
  (make-sum (deriv (addend expr) var)
            (deriv (augend expr) var)))

(define (deriv-product expr var)
  (make-sum
   (make-product (multiplier expr)
                 (deriv (multiplicand expr) var))
   (make-product (deriv (multiplier expr) var)
                 (multiplicand expr))))

(define (deriv-exponentiation expr var)
  (make-product
   (make-product
    (exponent expr)
    (make-exponentiation (base expr)
                         (make-sum (exponent expr) -1)))
   (deriv (base expr) var)))

(put 'deriv '+ deriv-sum)
(put 'deriv '* deriv-product)
(put 'deriv '^ deriv-exponentiation)
```

```
;; boilerplate for get & put

(define table (make-hash))
(define (put key1 key2 value) (hash-set! table (list key1 key2) value))
(define (get key1 key2)       (hash-ref  table (list key1 key2) #f))
```

```
;; New deriv procedure

(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp) (if (same-variable? exp var) 1 0))
        (else ((get 'deriv (operator exp)) (operands exp) var))))

(define (operator exp) (car exp))
(define (operands exp) exp) ;; note this change

(check-equal? (deriv '(+ x 3) 'x) 1)
(check-equal? (deriv '(* x 3) 'x) 3)
(check-equal? (deriv '(^ x 3) 'x) '(* 3 (^ x 2)))
```

d. In this simple algebraic manipulator the type of an expression is the algebraic operator that binds it together. Suppose, however, we indexed the procedures in the opposite way, so that the dispatch line in `deriv` looked like

```
((get (operator exp) 'deriv) (operands exp) var)
```

What corresponding changes to the derivative system are required?

```
;; d) Just swap key1 & key2 in your get proceduce implementation


  (define table (make-hash))
  (define (put key1 key2 value) (hash-set! table (list key1 key2) value))
  (define (get key1 key2)       (hash-ref  table (list key1 key2) #f))
```

Exercise 2.74: Insatiable Enterprises, Inc., is a highly decentralized conglomerate company consisting of a large number of independent divisions located all over the world. The company's computer facilities have just been interconnected by means of a clever network-interfacing scheme that makes the entire network appear to any user to be a single computer. Insatiable's president, in her first attempt to exploit the ability of the network to extract administrative information from division files, is dismayed to discover that, although all the division files have been implemented as data structures in Scheme, the particular data structure used varies from division to division. A meeting of division managers is hastily called to search for a strategy to integrate the files that will satisfy headquarters' needs while preserving the existing autonomy of the divisions.

division's personnel records consist of a single file, which contains a set of records keyed on employees' names. The structure of the set varies from division to division. Furthermore, each employee's record is itself a set (structured differently from division to division) that contains information keyed under identifiers such as `address` and `salary`. In particular:

a. Implement for headquarters a `get-record` procedure that retrieves a specified employee's record from a specified personnel file. The procedure should be applicable to any division's file. Explain how the individual divisions' files should be structured. In particular, what type information must be supplied?

```
;; Assume unique names for indexing

(define divA-record-set '(("Bob" (address "123 Lane")
                                 (salary  100000))
                          ("Jen" (salary  150000)
                                 (adresss "456 Drive"))))

;; name | address | salary
(define divB-record-set '(("Jim" "789 Street" 200000)
                          ("Ann" "000 Ave"    250000)))
```

```
;; a)

(define (findf proc lst)
  (let ((temp (memf proc lst)))
    (if temp (car temp) #f)))

(define (car=? x lst) (equal? x (car lst)))

(define (get-recordA name) (findf (curry car=? name) divA-record-set))
(define (get-recordB name) (findf (curry car=? name) divB-record-set))

(put 'A 'record get-recordA)
(put 'B 'record get-recordB)

(define (get-record div name)
  ((get div 'record) name))

;; > (get-record 'A "Jen")
;; '("Jen" (salary 150000) (adresss "456 Drive"))
;; > (get-record 'B "Jim")
;; '("Jim" "789 Street" 200000)
```

b. Implement for headquarters a `get-salary` procedure that returns the salary information from a given employee's record from any division's personnel file. How should the record be structured in order to make this operation work?

```
;; b)

(require threading)

(put 'A 'salary (λ (name) (~>> (get-recordA name)
                               (cdr)
                               (findf (curry car=? 'salary))
                               (cadr))))
(put 'B 'salary (λ (name) (caddr (get-recordB name))))

(define (get-salary div employee-name)
  ((get div 'salary) employee-name))

;; > (get-salary 'A "Jen")
;; 150000
;; > (get-salary 'B "Ann")
;; 250000
```

c. Implement for headquarters a `find-employee-record` procedure. This should search all the divisions' files for the record of a given employee and return the record. Assume that this procedure takes as arguments an employee's name and a list of all the divisions' files.

```
;; c)

(define (find-employee-record name div-list)
  (if (null? div-list)
      "Failed to find employee"
      (let* ((div (car div-list))
             (temp (get-record div name)))
        (if temp
            temp
            (find-employee-record name (cdr div-list))))))

;; > (find-employee-record "Jim" '(A B))
;; '("Jim" "789 Street" 200000)
;; > (find-employee-record "Jen" '(A B))
;; '("Jen" (salary 150000) (adresss "456 Drive"))
```

d. When Insatiable takes over a new company, what changes must be made in order to incorporate the new personnel information into the central system?

```
;; d) Add get-record, get-salary + corresponding put methods
```

Message passing

The key idea of data-directed programming is to handle generic operations in programs by dealing explicitly with operation-and-type tables, such as the table in Figure 2.22. The style of programming we used in Section 2.4.2 organized the required dispatching on type by having each operation take care of its own dispatching. In effect, this decomposes the operation-and-type table into rows, with each generic operation procedure representing a row of the table.

An alternative implementation strategy is to decompose the table into columns and, instead of using "intelligent operations" that dispatch on data types, to work with "intelligent data objects" that dispatch on operation names. We can do this by arranging things so that a data object, such as a rectangular number, is represented as a procedure that takes as input the required operation name and performs the operation indicated. In such a discipline, `make-from-real-imag` could be written as

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op: MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

Exercise 2.75: Implement the constructor `make-from-mag-ang` in message-passing style. This procedure should be analogous to the `make-from-real-imag` procedure given above.

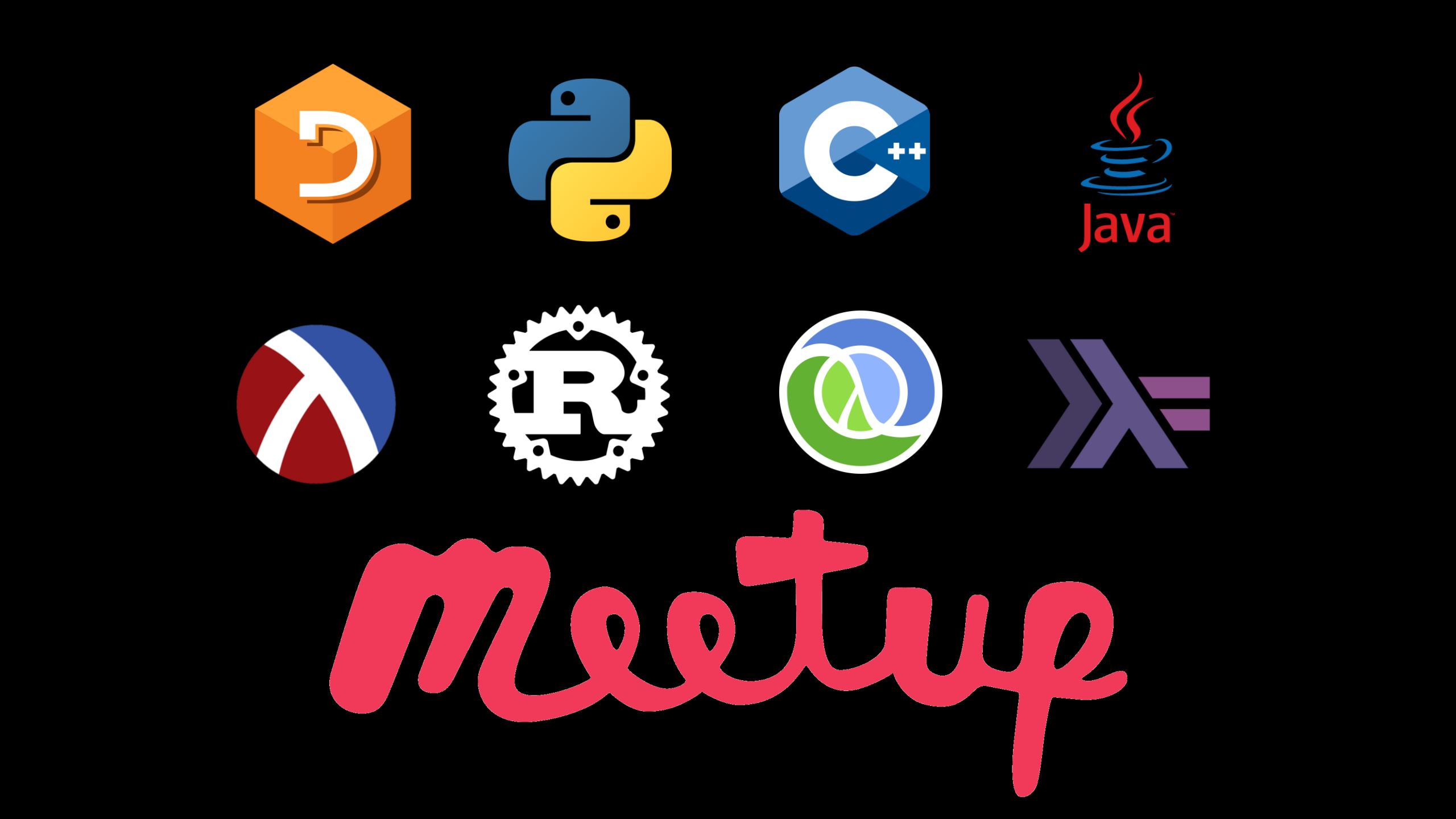
```
;; Exercise 2.75

(define (make-from-mag-ang r a)
  (define (dispatch op)
    (cond ((eq? op 'real-part) (* r (cos a)))
          ((eq? op 'imag-part) (* r (sin a)))
          ((eq? op 'magnitude) r)
          ((eq? op 'angle) a)
          (else (error "Unkown op --- MAKE-FROM-MAG-ANG" op))))
  dispatch)
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


Java
meetup