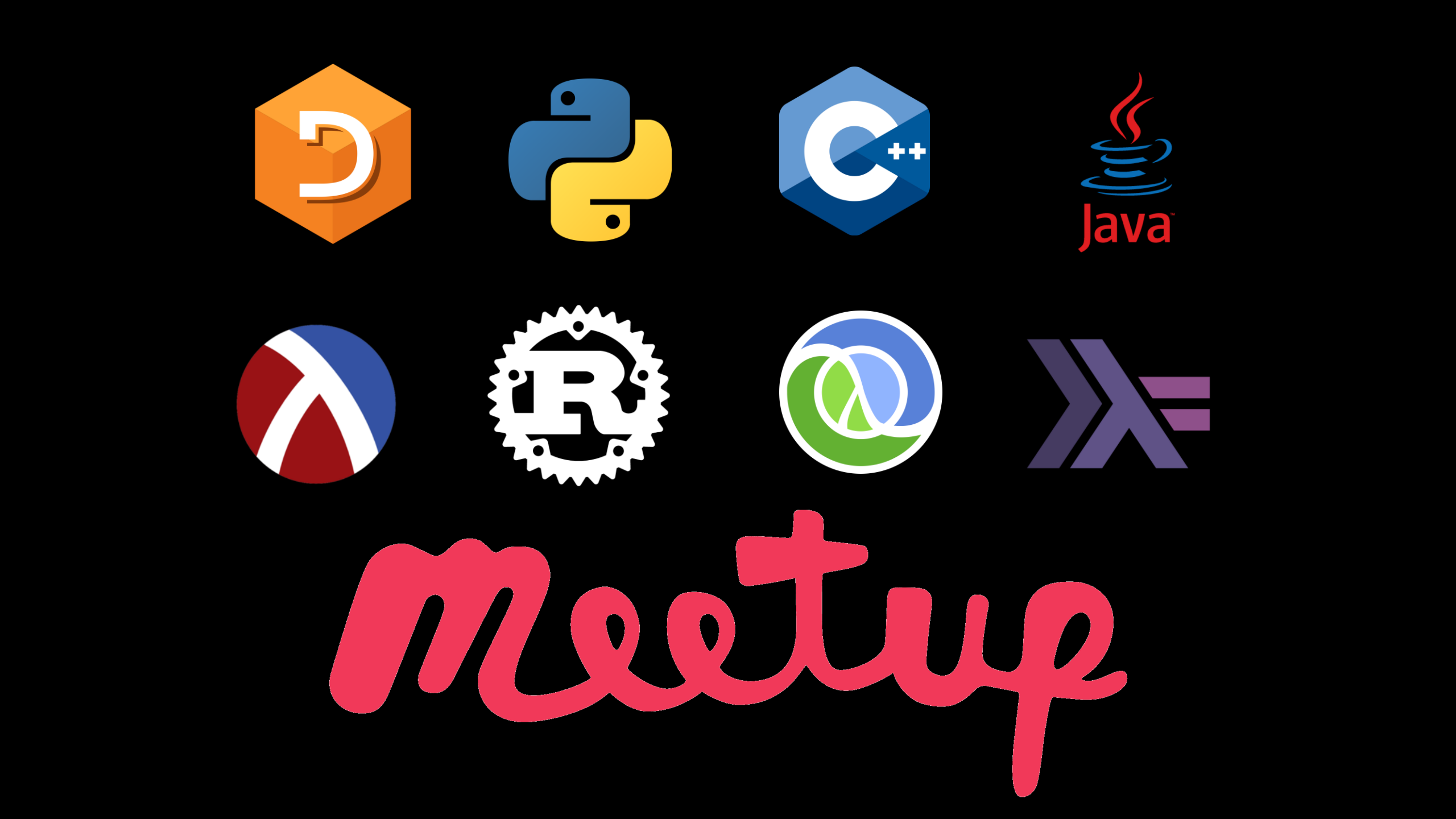
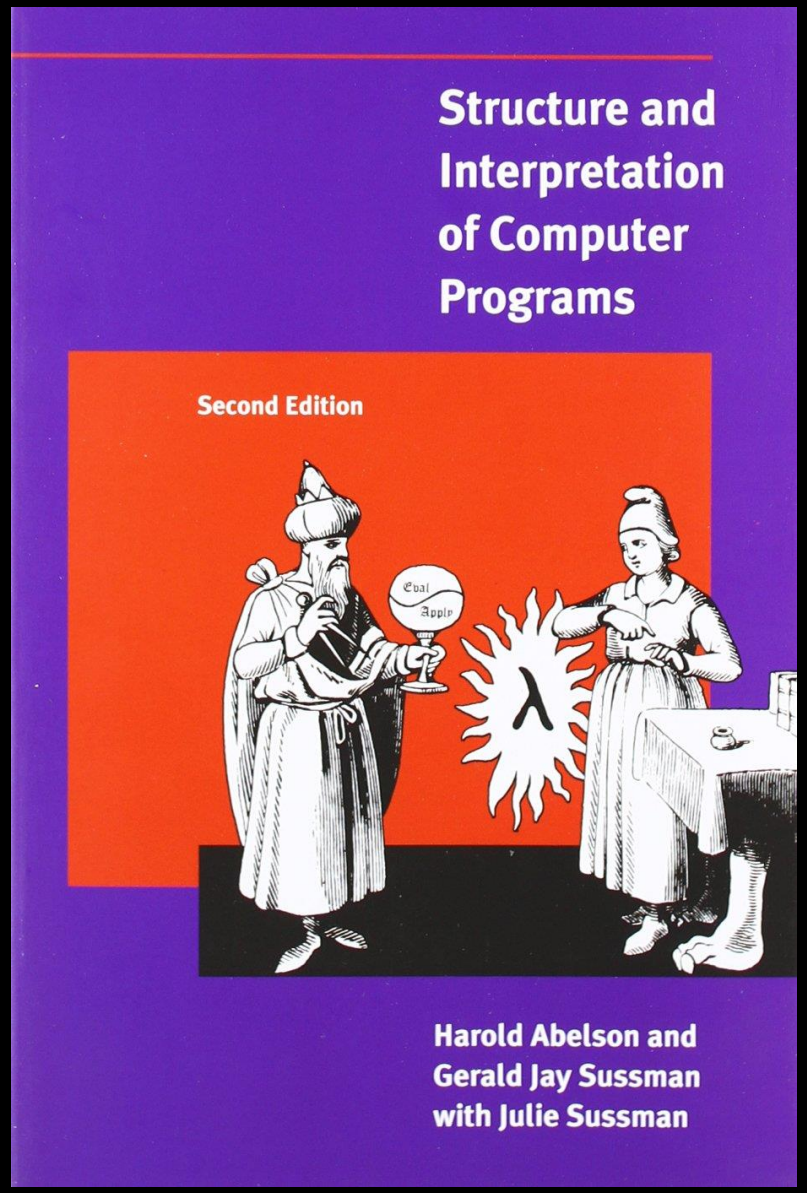

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 2.3

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

foldl vs foldr

```
;; 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
```

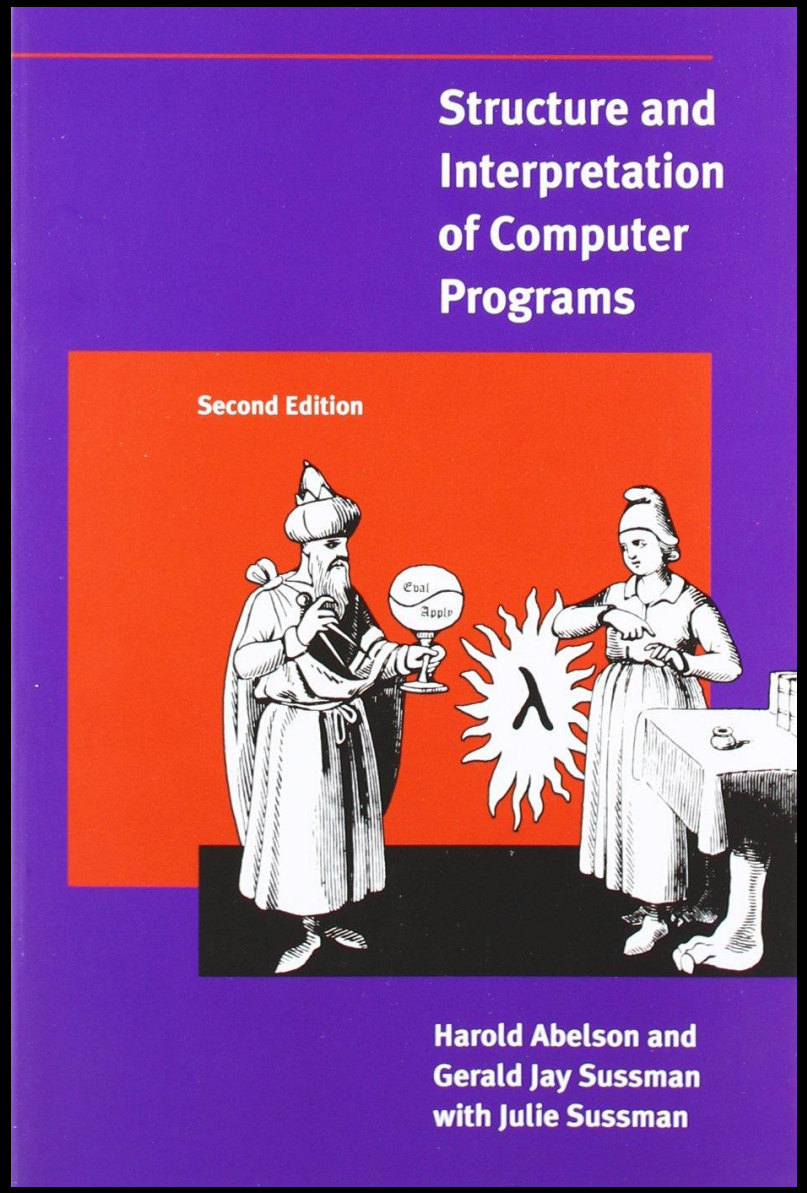
```
> (foldl (λ (v acc) (cons v acc)) '() '(1 2 3 4))
'(4 3 2 1)

> (foldr (λ (v acc) (cons v acc)) '() '(1 2 3 4))
'(1 2 3 4)
```

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

Prelude> foldl (/) 1 [1..3]
0.16666666666666666
Prelude> foldr (/) 1 [1..3]
1.5
```

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

Prelude> foldl (/) 1 [1..3]
0.16666666666666666
Prelude> foldr (/) 1 [1..3]
1.5

Prelude> scanl (/) 1 [1..3]
[1.0,1.0,0.5,0.16666666666666666]
-- init, acc (1) / 1, acc (1) / 2, acc (1/2) / 3
Prelude> scanr (/) 1 [1..3]
[1.5,0.6666666666666666,3.0,1.0]
-- 1 / acc (2/3), 2 / acc (3), 3 / acc (1.0), init (1)
```

```
> (foldl / 1 '(1 2 3))
3/2
> (foldr / 1 '(1 2 3))
3/2
```



Structure and Interpretation of Computer Programs

Chaper 2.3

All the compound data objects we have used so far were constructed ultimately from numbers. In this section we extend the representational capability of our language by introducing the ability to work with arbitrary symbols as data.

```
(define a 1)
(define b 2)
(list a b)
(1 2)
(list 'a 'b)
(a b)
(list 'a b)
(a 2)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists:[34]

[32] Allowing quotation in a language wreaks havoc with the ability to reason about the language in simple terms, because it destroys the notion that equals can be substituted for equals. For example, three is one plus two, but the word "three" is not the phrase "one plus two." Quotation is powerful because it gives us a way to build expressions that manipulate other expressions (as we will see when we write an interpreter in Chapter 4). But allowing statements in a language that talk about other statements in that language makes it very difficult to maintain any coherent principle of what "equals can be substituted for equals" should mean. For example, if we know that the evening star is the morning star, then from the statement "the evening star is Venus" we can deduce "the morning star is Venus." However, given that "John knows that the evening star is Venus" we cannot infer that "John knows that the morning star is Venus."

Exercise 2.54: Two lists are said to be `equal?` if they contain equal elements arranged in the same order. For example,

```
(equal? '(this is a list) '(this is a list))
```

is true, but

```
(equal? '(this is a list) '(this (is a) list))
```

is false. To be more precise, we can define `equal?` recursively in terms of the basic `eq?` equality of symbols by saying that `a` and `b` are `equal?` if they are both symbols and the symbols are `eq?`, or if they are both lists such that `(car a)` is `equal?` to `(car b)` and `(cdr a)` is `equal?` to `(cdr b)`. Using this idea, implement `equal?` as a procedure.[36]

```scheme
(define (equal? a b)
  (cond ((and (null? a) (null? b)) #t)
        ((or (null? a) (null? b)) #f)
        ((eq? (car a) (car b)) (equal? (cdr a) (cdr b)))
        (else #f)))

(check-equal? (equal? '(0 1 2) (range 3)) #t)
(check-equal? (equal? '(0 1 2) (range 2)) #f)
```

2.3.2 Example: Symbolic Differentiation

$$\frac{dc}{dx} = 0, \quad \text{for } c \text{ a constant or a variable different from } x,$$

$$\frac{dx}{dx} = 1,$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx},$$

$$\frac{d(uv)}{dx} = u\frac{dv}{dx} + v\frac{du}{dx}.$$

```scheme
;; Symbolic Differentiation (from the book)

(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 (sum? x) (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? x) (and (pair? x) (eq? (car x) '*)))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
```

```
(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))))
```

Exercise 2.56: Show how to extend the basic differentiator to handle more kinds of expressions. For instance, implement the differentiation rule

$$\frac{d(u^n)}{dx} = nu^{n-1}\frac{du}{dx}$$

by adding a new clause to the `deriv` program and defining appropriate procedures `exponentiation?`, `base`, `exponent`, and `make-exponentiation`. (You may use the symbol `**` to denote exponentiation.) Build in the rules that anything raised to the power 0 is 1 and anything raised to the power 1 is the thing itself.

```
;; Exercise 2.56 (page 203)

(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)

(define (exponentiation? exp)
  (and (list? exp) (eq? (car exp) '^)))
```

```
(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))))
        ((exponentiation? exp)
         (make-product
          (make-product
           (exponent exp)
           (make-exponentiation (base exp)
                                (make-sum (exponent exp) -1)))
          (deriv (base exp) var)))
        (else
         (error "unknown expression type: DERIV" exp))))

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

2.3.3 Example: Representing Sets

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
```

```
(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))
```

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

Exercise 2.59: Implement the `union-set` operation for the unordered-list representation of sets.

```
;; Exercise 2.59 (page 207)

(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)))))

(check-equal? (element-of-set? 1 '(1 2 3)) #t)
(check-equal? (element-of-set? 4 '(1 2 3)) #f)

(define (union-set a b)
  (cond ((null? b) a)
        ((element-of-set? (car b) a) (union-set a (cdr b)))
        (else (union-set (cons (car b) a) (cdr b)))))

(check-equal? (union-set '(1 2 3) '(4 5 6)) '(6 5 4 1 2 3))
(check-equal? (union-set '(1 2 3) '(2 3 4)) '(4 1 2 3 ))
```

Sets as ordered lists

Sets as binary trees

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection-set (cdr set1)
                                          (cdr set2))))
              ((< x1 x2)
               (intersection-set (cdr set1) set2))
              ((< x2 x1)
               (intersection-set set1 (cdr set2)))))))
```

Exercise 2.62: Give a $\Theta(n)$ implementation of `union-set` for sets represented as ordered lists.

```
;; Exercise 2.62 (page 210)

(define (union-set a b)
  (define (iter a b c)
    (cond ((null? a) (append (reverse b) c))
          ((null? b) (append (reverse a) c))
          ((= (car a) (car b)) (iter (cdr a) (cdr b) (cons (car a) c)))
          ((< (car a) (car b)) (iter (cdr a)    b    (cons (car a) c)))
          ((> (car a) (car b)) (iter    a    (cdr b) (cons (car b) c)))))
  (reverse (iter a b '())))

(check-equal? (union-set '(1 2 3) '(4 5 6)) '(1 2 3 4 5 6))
(check-equal? (union-set '(4 5 6) '(1 2 3)) '(1 2 3 4 5 6))
(check-equal? (union-set '(1 2 3) '(2 3 4)) '(1 2 3 4))
(check-equal? (union-set '(1 2 3) '(3 4)) '(1 2 3 4))
(check-equal? (union-set '(1 3) '(2 3 4)) '(1 2 3 4))
```

Sets as ordered lists

Sets as binary trees

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))
```

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))
```

Exercise 2.63: Each of the following two procedures converts a binary tree to a list.

```
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1
                      (right-branch tree))))))
```

```scheme
(define (tree->list tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list
                             (right-branch tree)
                             result-list)))))
  (copy-to-list tree '()))
```

a. Do the two procedures produce the same result for every tree? If not, how do the results differ? What lists do the two procedures produce for the trees in Figure 2.16?

b. Do the two procedures have the same order of growth in the number of steps required to convert a balanced tree with n elements to a list? If not, which one grows more slowly?

```
;; Exercise 2.63 (page 213/14)

;; a) the same (in-order traversal)
;; b) append vs cons = O(nlogn) vs O(n)
```

Exercise 2.65: Use the results of Exercise 2.63 and Exercise 2.64 to give $\Theta(n)$ implementations of `union-set` and `intersection-set` for sets implemented as (balanced) binary trees.[41]

```
(define (list->tree elements)
  (car (partial-tree elements (length elements))))
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result
               (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result
                   (partial-tree
                    (cdr non-left-elts)
                    right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts
                     (cdr right-result)))
                (cons (make-tree this-entry
                                 left-tree
                                 right-tree)
                      remaining-elts)))))))))
```

```
;; union-set via balanced-tree

(define (union-set a b)
  (list->tree
   (remove-duplicates
    (append (tree->list a)
            (tree->list b)))))

;; > (union-set (list->tree '(1 2 3 5 7 9 46))
;;              (list->tree '(5 6 10 11 20 23 46)))
;; '(9
;;   (3 (1 () (2 () ())) (5 () (7 () ())))
;;   (10 (46 () (6 () ())) (20 (11 () ()) (23 () ()))))
```

2.3.4 Example: Huffman Encoding Trees

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

BACADAEAFABBAAAGAH

is encoded as the string of 54 bits

001000010000011000100000101000001001000000000110000111

```
A 0       C 1010    E 1100    G 1110
B 100     D 1011    F 1101    H 1111
```

With this code, the same message as above is encoded as the string

```
100010100101101100011010100100000111001111
```

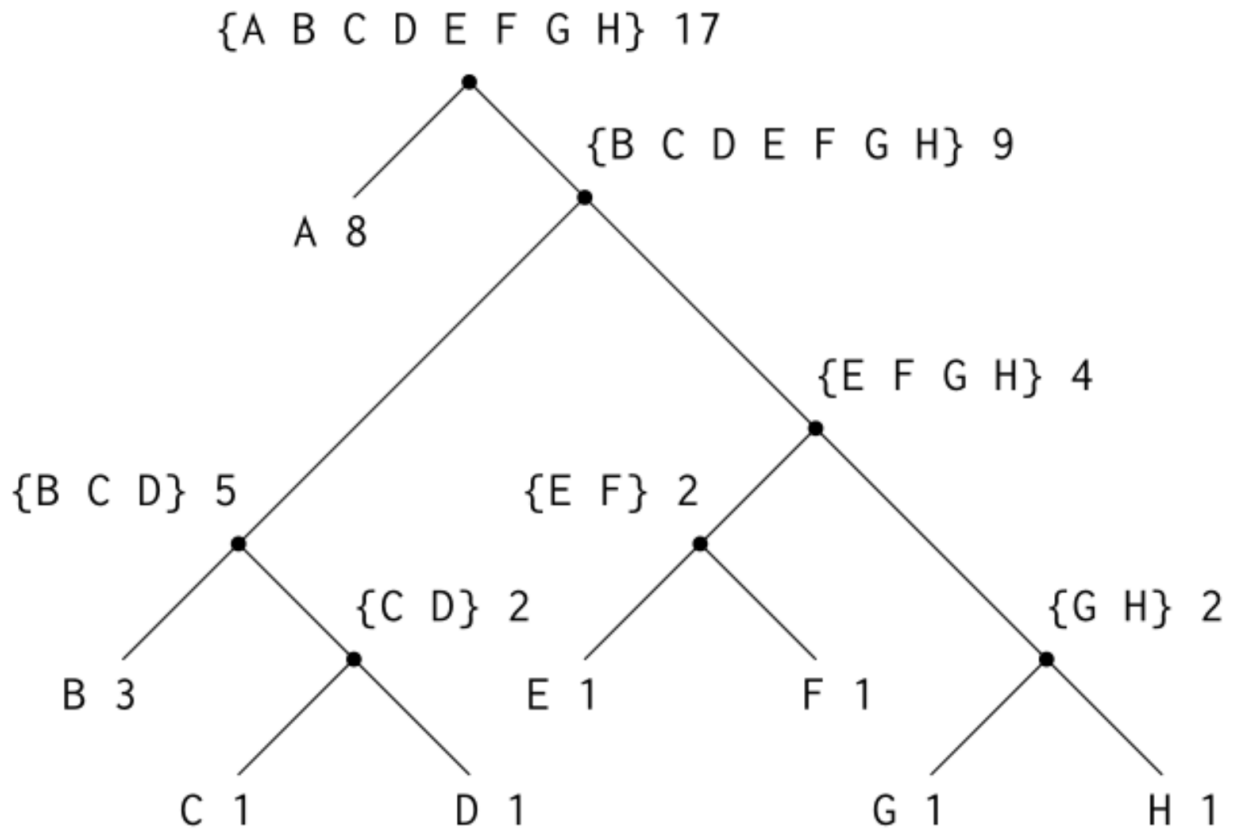
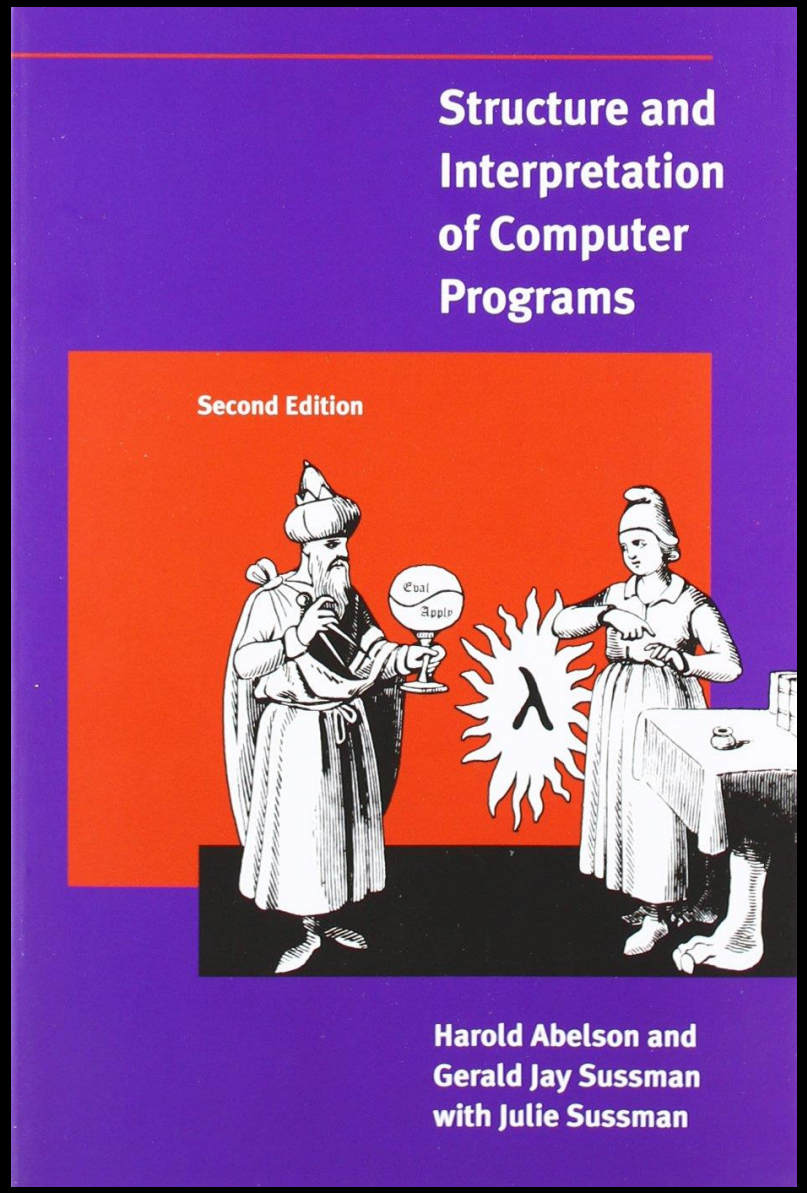


Figure 2.18: A Huffman encoding tree.



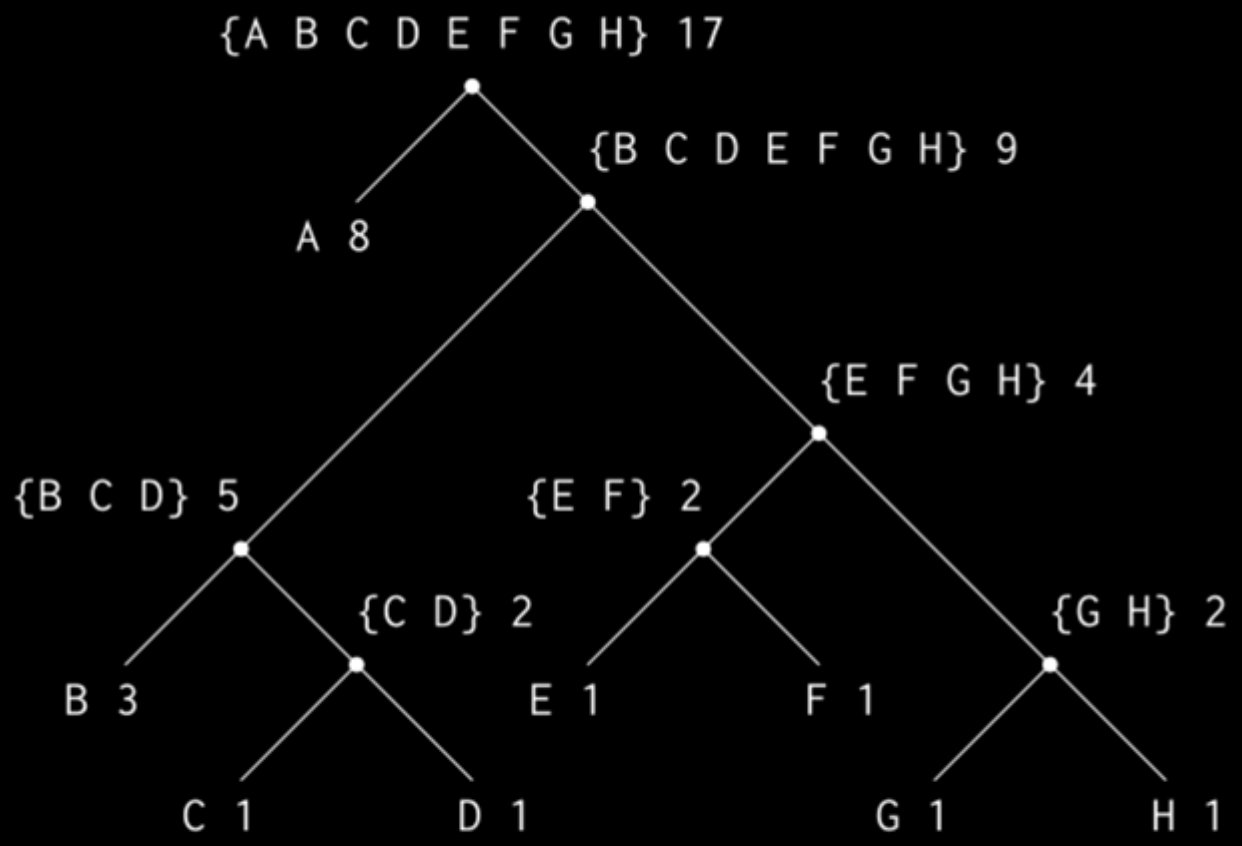


Figure 2.18: A Huffman encoding tree.

Exercise 2.67: Define an encoding tree and a sample message:

```
(define sample-tree
  (make-code-tree (make-leaf 'A 4)
                  (make-code-tree
                   (make-leaf 'B 2)
                   (make-code-tree
                    (make-leaf 'D 1)
                    (make-leaf 'C 1)))))
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
```

Use the decode procedure to decode the message, and give the result.

```
;;    root
;; 0  /  \  1
;;   A    .
;;       / \
;;   0  B   .  1
;;         / \
;;     0  D   c  1
```

```
(define sample-message '(0 1 1 0 0 1 0 1 0 1 1 1 0))
;;                       A D     A B   B   C     A
```

Exercise 2.68: The encode procedure takes as arguments a message and a tree and produces the list of bits that gives the encoded message.

```
(define (encode message tree)
  (if (null? message)
      '()
      (append (encode-symbol (car message) tree)
              (encode (cdr message) tree))))
```

`encode-symbol` is a procedure, which you must write, that returns the list of bits that encodes a given symbol according to a given tree. You should design `encode-symbol` so that it signals an error if the symbol is not in the tree at all. Test your procedure by encoding the result you obtained in Exercise 2.67 with the sample tree and seeing whether it is the same as the original sample message.

```
(define (encode-symbol s tree)
  (if (leaf? tree)
      (if (eq? s (symbol-leaf tree))
          '()
          (error "fail"))
      (if (memq s (symbols (left-branch tree)))
          (cons 0 (encode-symbol s (left-branch tree)))
          (cons 1 (encode-symbol s (right-branch tree))))))

;; > (encode '(A D A B B C A) sample-tree)
;; '(0 1 1 0 0 1 0 1 0 1 1 1 0)
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

“**quotation** is a very **complex** concept and adding it to a language causes a great deal of troubles”

Gerald Sussman

Lecture 3B, SICP

"Chicago" has seven letters.

Chicago is the biggest city in Illinois.

"The biggest city in Illinois" has seven letters.

“we can’t substitute into what is called **referentially opaque** contexts”

Gerald Sussman

Lecture 3B, SICP

“at this point we’ve built our **sledgehammer**”

Gerald Sussman

Lecture 3B, SICP

LANGUAGE HISTORY CHART

First letter of each name has been aligned with the approximate date on which work began.

THIS TYPE STYLE indicates languages of major importance, because of their wide usage or technical significance.

THIS TYPE STYLE indicates languages of moderate importance.

THIS TYPE STYLE is used for all other languages.

Parentheses were used to indicate alternate names, or the later addition of the sequence number "1."

═══ indicates that the second language is a direct extension of the first

——— indicates that the second language is an approximate extension of the first, i.e., very similar to the first, but not completely upward compatible

– – – – indicates strong influence; sometimes the second language is "like, or in the style of" the first

- - - - - - - - - - - - indicates an approximate subset

Each of the following marks is associated with the language above or to its left:

● indicates preliminary or informal specifications or manual

■ indicates a public manual, or formal publication via technical paper, or public presentation

▲ release for usage outside development group

General Comments

This chart represents only the personal opinions of the author as far as value judgments are involved, and the author's best estimate in many cases as far as dates are involved. The indications of the start of the work are the most questionable.

The information for languages in 1971 is based solely on those listed in "Roster of Programming Languages-1971," Computers and Automation, Vol. 20, No. 6B (June 1971), pp. 6-13.

In most cases, dialects with differing names have been omitted. This has the unfortunate effect of appearing to minimize the importance of some languages which spawned numerous versions under differing names (e.g., JOSS).

Languages for specialized application areas (e.g., simulation, machine tool control, civil engineering, systems programming) have not been included because of space considerations. This explains the absence of such obvious languages as APT, GPSS, SIMSCRIPT, COGO, BLISS.

Acknowledgment: The idea for such a chart in such a format came from the one by Christopher J. Shaw entitled "Milestones in Computer Programming" and included with the [ACM Los Angeles Chapter] SIGPLAN notices, February 1965.

"Programming Languages: History and Future"
by Jean E. Sammet
Communications of the ACM, Vol. 15, July 1972

NUMERICAL SCIENTIFIC

Two-Dimensional ➤

ON-LINE

NON-NUMERICAL SCIENTIFIC (= Formula Manipulation)

BUSINESS DATA PROCESSING

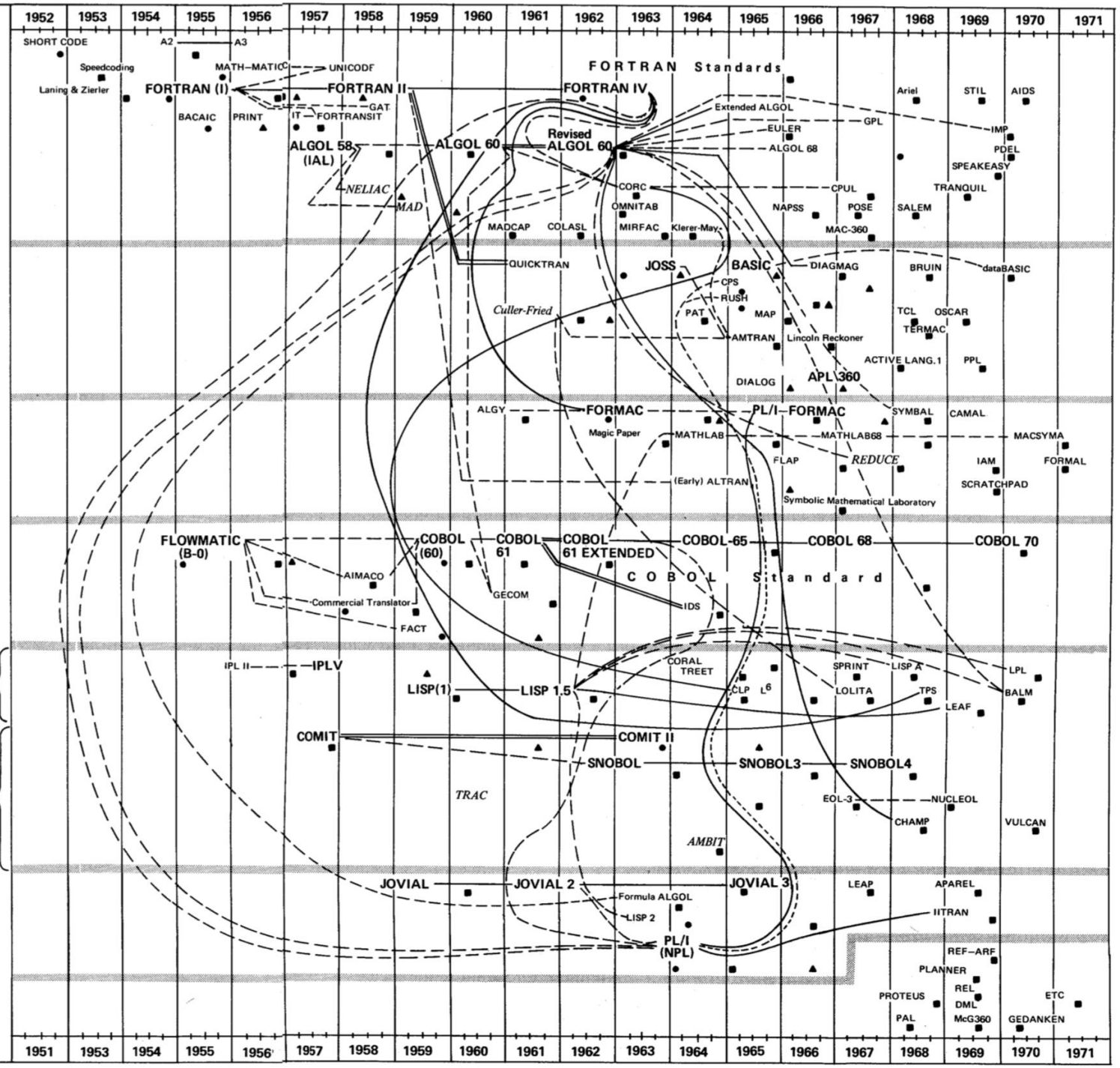
List Processing

STRING AND LIST PROCESSING

String Processing

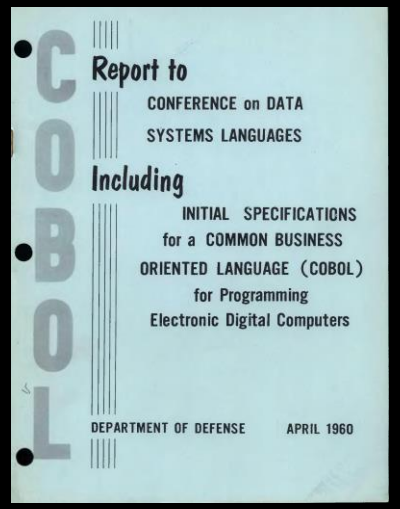
MULTIPURPOSE

EXPERIMENTAL AND OTHERS



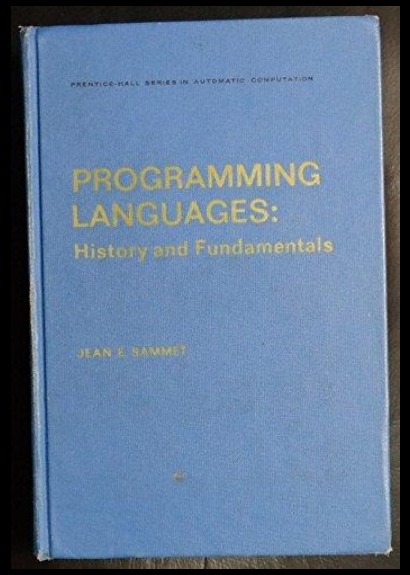
Who is Jean Sammet?

- Member of the COBOL group
- BA & MA in Mathematics (later PhD)
- Developed FORMAC (**FOR**mula **MA**nipulation **C**ompiler) – this was an extension to FORTRAN IV - while at
- She was president of the ACM from 1974-76 (notably she was the first female president)
- Author of Programming Languages: History and Fundamentals
- First job was at MetLife



IBM

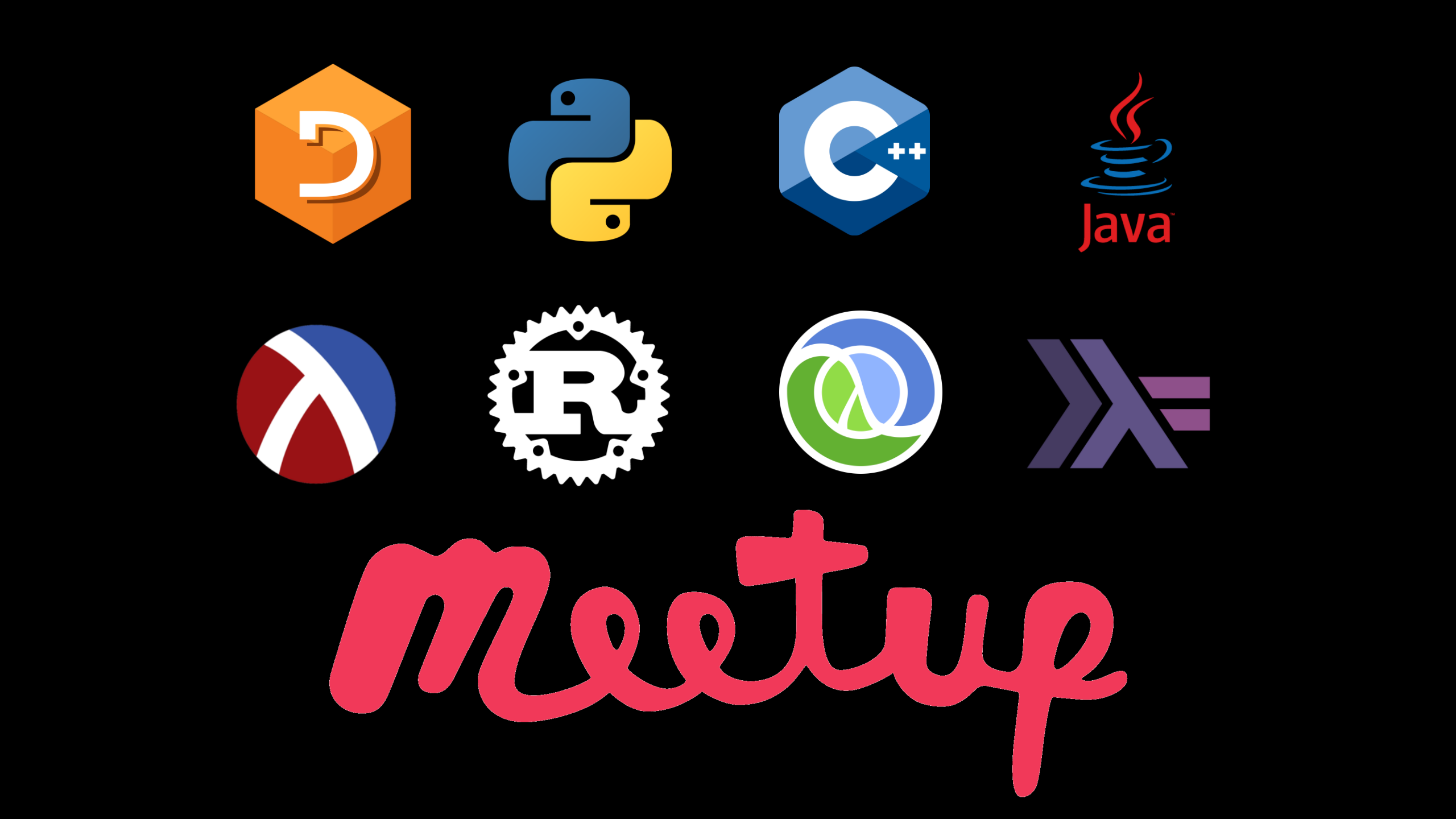
Mount Holyoke College 1837 PS.CXLIV:XII



University of Illinois Learning & Labor Chartered 1867

MetLife

MetLife


Java
meetup