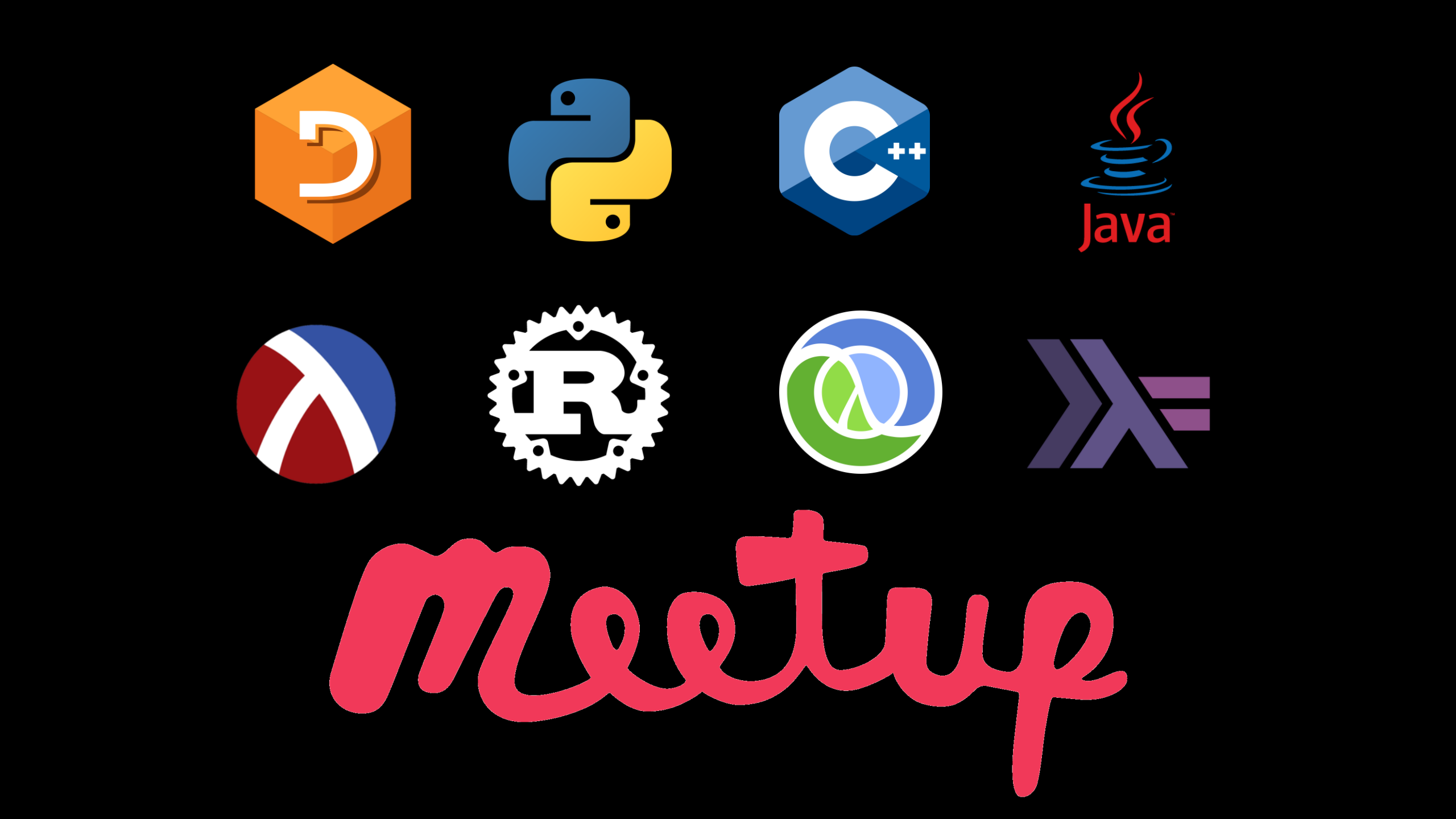
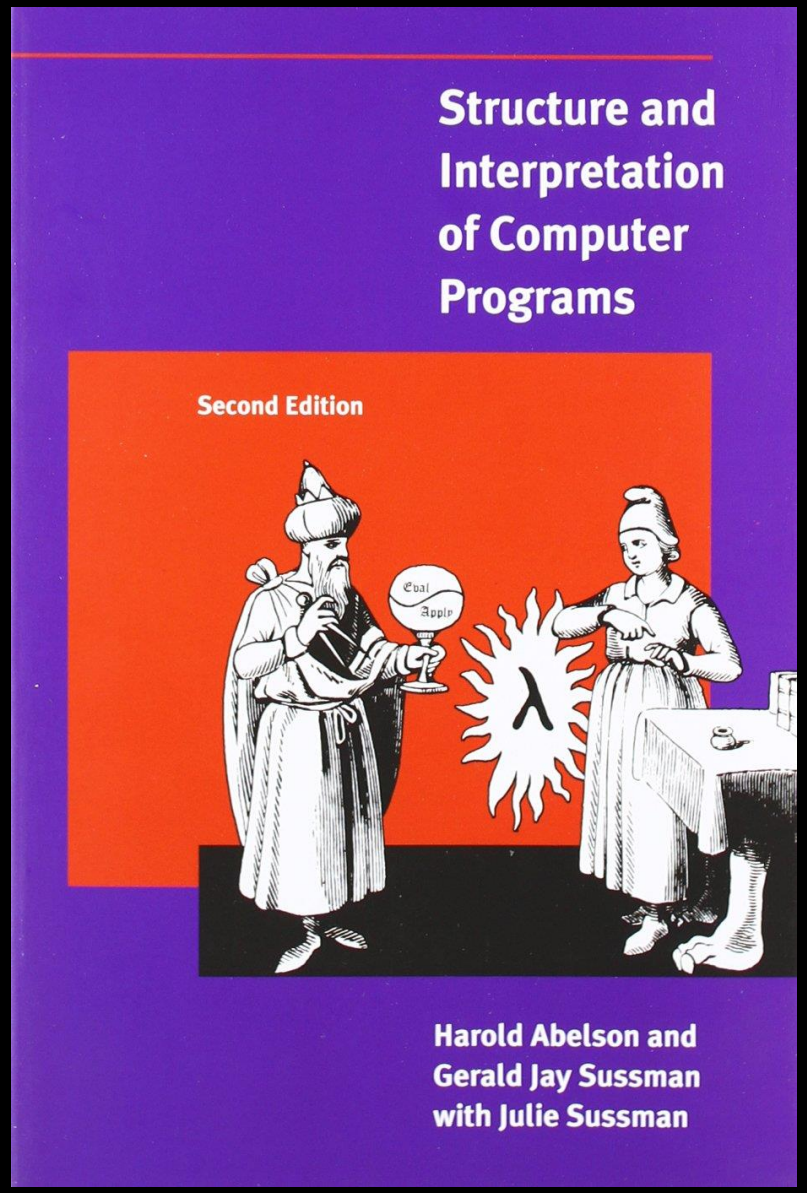

Java
meetup



Structure and Interpretation of Computer Programs

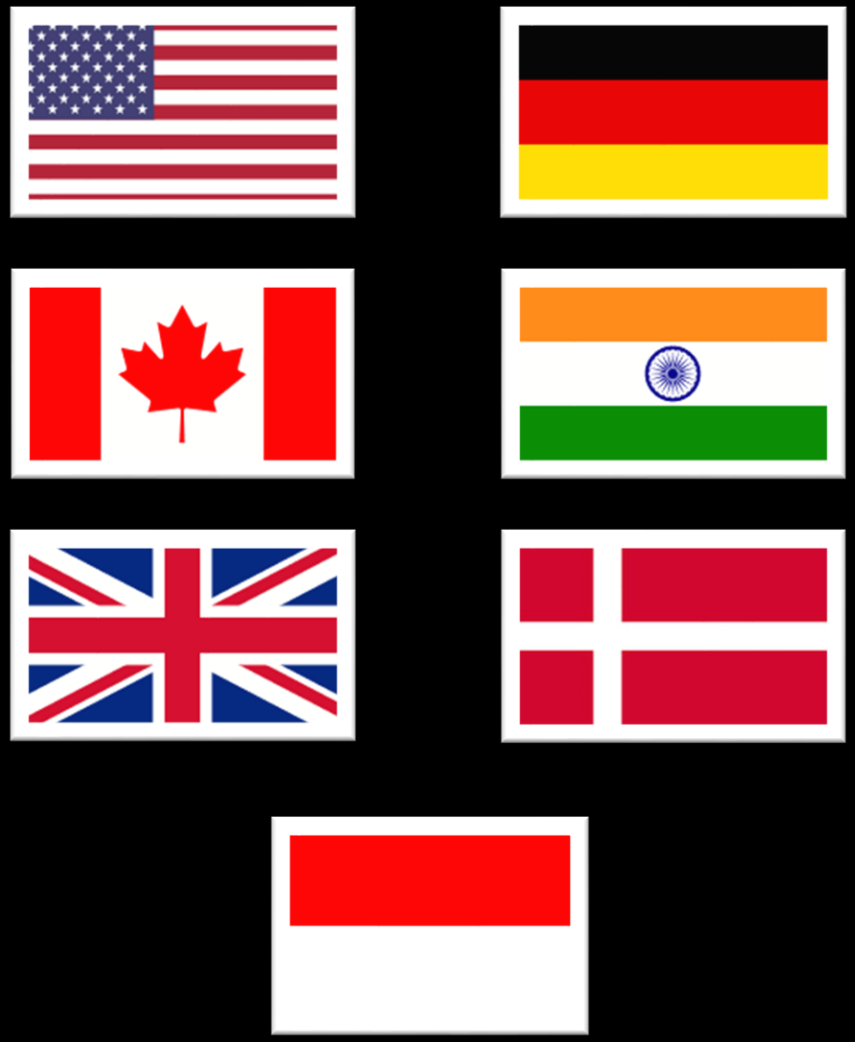
Chaper 2.2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

In aggregate




DISCORD

Programming Languages Virtual Meetup

1 Tweet

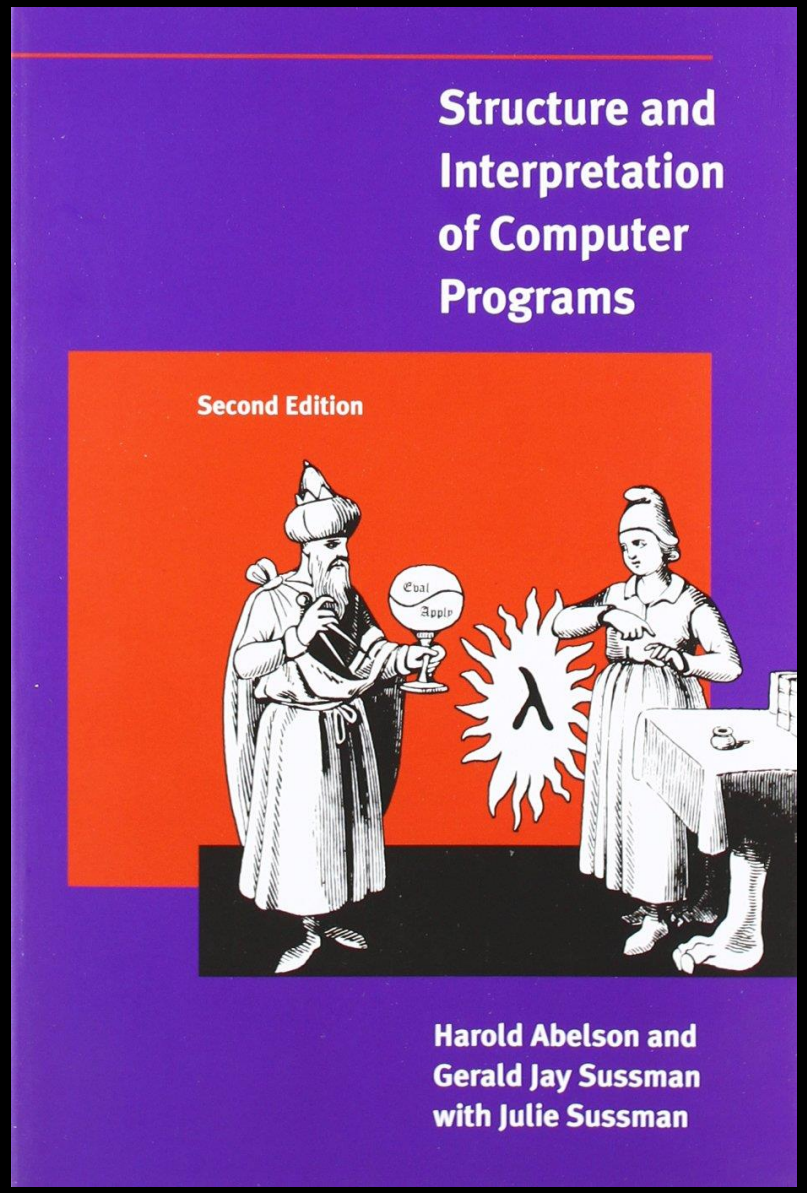
Following

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 2.2 1/2

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool. We refer to this ability as the *closure property* of cons. In general, an operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation.[6]

We describe some conventional techniques for using pairs to represent sequences and trees, and we exhibit a graphics language that illustrates closure in a vivid way.[7]

[6] The use of the word "closure" here comes from abstract algebra, where a set of elements is said to be closed under an operation if applying the operation to elements in the set produces an element that is again an element of the set. The Lisp community also (unfortunately) uses the word "closure" to describe a totally unrelated concept: A closure is an implementation technique for representing procedures with free variables. We do not use the word "closure" in this second sense in this book.

[7] The notion that a means of combination should satisfy closure is a straightforward idea. Unfortunately, the data combiners provided in many popular programming languages do not satisfy closure, or make closure cumbersome to exploit. In Fortran or Basic, one typically combines data elements by assembling them into arrays—but one cannot form arrays whose elements are themselves arrays. Pascal and C admit structures whose elements are structures. However, this requires that the programmer manipulate pointers explicitly, and adhere to the restriction that each field of a structure can contain only elements of a prespecified form. Unlike Lisp with its pairs, these languages have no built-in general-purpose glue that makes it easy to manipulate compound data in a uniform way. This limitation lies behind Alan Perlis's comment in his foreword to this book: "In Pascal the plethora of declarable data structures induces a specialization within functions that inhibits and penalizes casual cooperation. It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

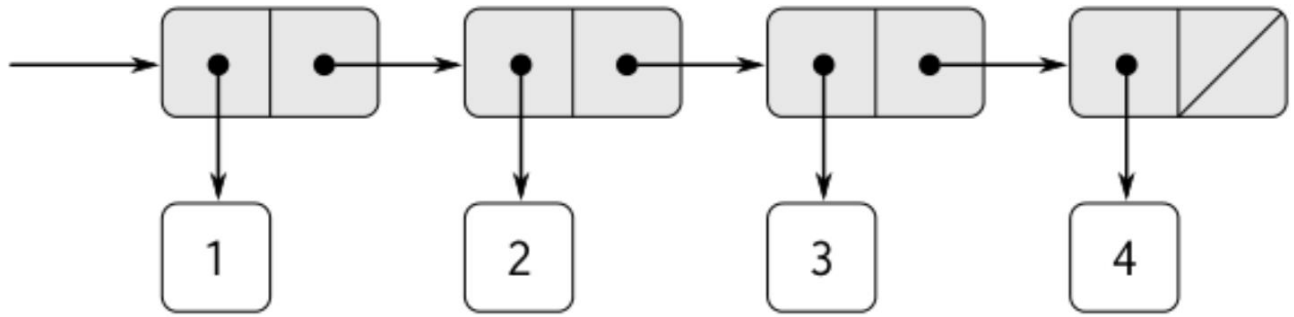
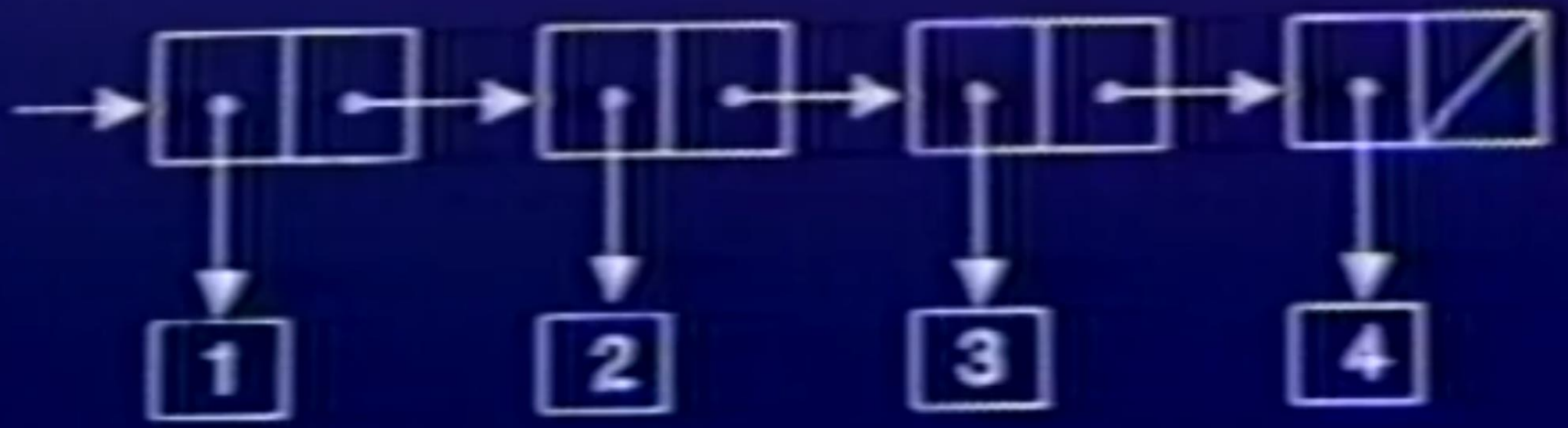


Figure 2.4: The sequence 1, 2, 3, 4 represented as a chain of pairs.

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
```




```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))
```

[9]Since nested applications of `car` and `cdr` are cumbersome to write, Lisp dialects provide abbreviations for them—for instance,

`(cadr ⟨arg⟩) = (car (cdr ⟨arg⟩))`

The names of all such procedures start with `c` and end with `r`. Each `a` between them stands for a `car` operation and each `d` for a `cdr` operation, to be applied in the same order in which they appear in the name. The names `car` and `cdr` persist because simple combinations like `cadr` are pronounceable.


[10]It's remarkable how much energy in the standardization of Lisp dialects has been dissipated in arguments that are literally over nothing: Should `nil` be an ordinary name? Should the value of `nil` be a symbol? Should it be a list? Should it be a pair? In Scheme, `nil` is an ordinary name, which we use in this section as a variable whose value is the end-of-list marker (just as `true` is an ordinary variable that has a true value). Other dialects of Lisp, including Common Lisp, treat `nil` as a special symbol. The authors of this book, who have endured too many language standardization brawls, would like to avoid the entire issue. Once we have introduced quotation in Section 2.3, we will denote the empty list as `'()` and dispense with the variable `nil` entirely.

Exercise 2.17: Define a procedure `last-pair` that returns the list that contains only the last element of a given (nonempty) list:

```
;; Exercise 2.17

;; Solution 1

(require threading)

(define (last-pair lst)
  (~> lst
      reverse
      car))
```

```scheme
;; Solution 2

(define (last-pair lst)
  (list-ref lst (- (length lst) 1)))
```

```
;; Solution 3

(define (last-pair lst)
  (if (null? (cdr lst))
      (car lst)
      (last-pair (cdr lst))))
```

Exercise 2.18: Define a procedure `reverse` that takes a list as argument and returns a list of the same elements in reverse order:

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

```scheme
;; Exercise 2.18

(define (reverse lst)
  (define (iter lst acc)
    (if (null? lst)
        acc
        (iter (cdr lst) (cons (car lst) acc))))
  (iter lst '()))
```

Exercise 2.20:

```
(define (f x y . z) ⟨body⟩)
```

Use this notation to write a procedure `same-parity` that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument. For example,

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)
(same-parity 2 3 4 5 6 7)
(2 4 6)
```

```
(define (same-parity x . xs)
  (filter (λ (n) (= (remainder x 2)
                    (remainder n 2)))
          xs))

;; > (same-parity 1 1 2 3 4 5)
;; '(1 3 5)
;; > (same-parity 2 1 2 3 4 5)
;; '(2 4)
```

Exercise 2.21: The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Here are two different definitions of `square-list`. Complete both of them by filling in the missing expressions:

```
(define (square-list items)
  (if (null? items)
      nil
      (cons ⟨??⟩ ⟨??⟩)))
(define (square-list items)
  (map ⟨??⟩ ⟨??⟩))
```

```
;; Exercise 2.21

(define (sq x) (* x x))

(define (square-list items)
  (if (null? items)
      '()
      (cons (sq (car items))
            (square-list (cdr items)))))

(define (square-list2 items)
  (map sq items))
```

Exercise 2.23: The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all—`for-each` is used with procedures that perform an action, such as printing. For example,

```
(for-each (lambda (x)
            (newline)
            (display x))
          (list 57 321 88))
57
321
88
```

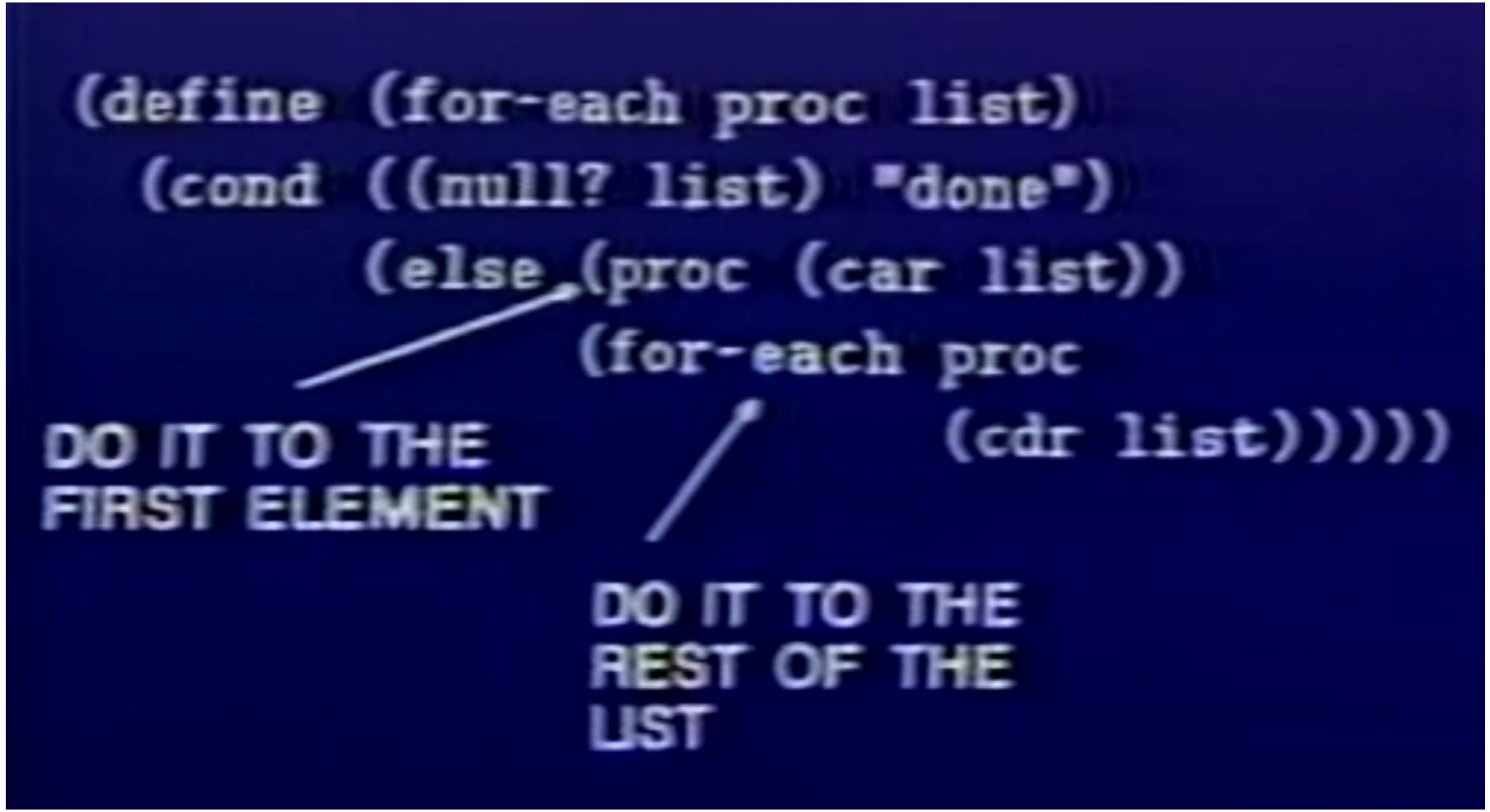
The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

```
;; Exercise 2.23

;; this fails :(
(define (for-each proc lst)
  (if (null? lst)
      (λ (x) (x))
      ((proc (car lst))
       (for-each proc (cdr lst)))))
```

```
;; this prints a #t at the end :(
(define (for-each proc lst)
  (cond ((null? lst) #t)
        (else (proc (car lst))
              (for-each proc (cdr lst)))))
```

```
;; this works
(define (for-each proc lst)
  (cond ((null? (cdr lst)) (proc (car lst)))
        (else (proc (car lst))
              (for-each proc (cdr lst)))))
```


(define (for-each proc list)
(cond ((null? list) "done")
(else (proc (car list))
(for-each proc
(cdr list)))))
DO IT TO THE
FIRST ELEMENT
DO IT TO THE
REST OF THE
LIST

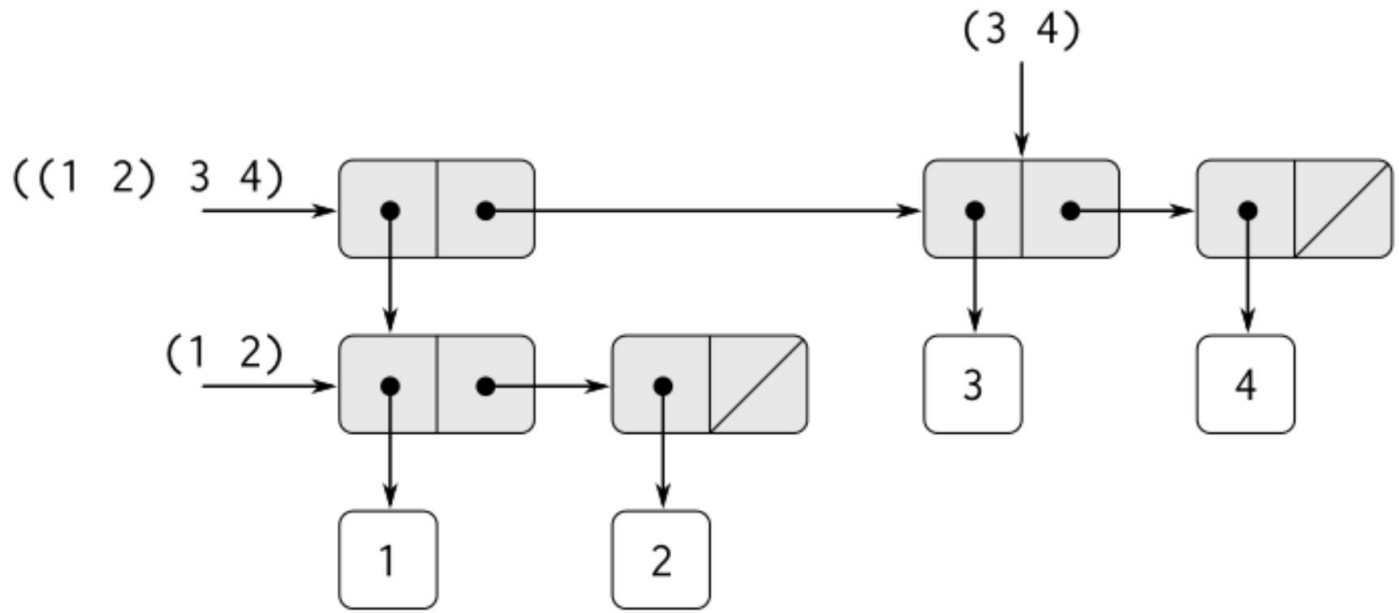


Figure 2.5: Structure formed by `(cons (list 1 2) (list 3 4))`.

```
(cons (list 1 2) (list 3 4))
```

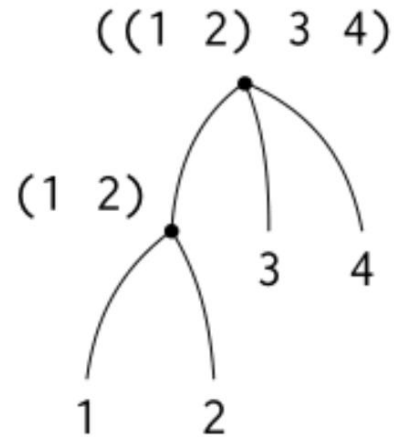


Figure 2.6: The list structure in Figure 2.5 viewed as a tree.

Exercise 2.25: Give combinations of `cars` and `cdrs` that will pick 7 from each of the following lists:

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

```
> (define x '(1 3 (5 7) 9))
> (car (cdaddr x))
;; 7
```

```
> (define x '((7)))
> (caar x)
;; 7
```

```
(require threading)

> (define x '(1 (2 (3 (4 (5 (6 7)))))))
> (~> x
      cadadr
      cadadr
      cadadr)
;; 7
```

Exercise 2.27: Modify your `reverse` procedure of Exercise 2.18 to produce a `deep-reverse` procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well. For example,

```
(define x (list (list 1 2) (list 3 4)))
x
((1 2) (3 4))
(reverse x)
((3 4) (1 2))
(deep-reverse x)
((4 3) (2 1))
```

```
;; Exercise 2.27

(define (deep-reverse lst)
  (define (iter lst acc)
    (if (null? lst)
        acc
        (let ((fst (car lst)))
          (iter (cdr lst)
                (cons (if (list? fst)
                          (reverse fst)
                          fst)
                      acc)))))
  (iter lst '()))
```

Exercise 2.28: Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
;; Exercise 2.28

;; Solution 1

(define fringe flatten) ; :p
```

```
;; Solution 2

(define (fringe tree)
  (if (null? tree)
      '()
      (let ((x (car tree)))
        (append (if (list? x)
                    (fringe x)
                    (list x))
                (fringe (cdr tree))))))
```

```
;; Example from book

(define (scale-tree tree factor)
  (cond ((null? tree) '())
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))
```

```
;; Example from book

(define (scale-tree tree factor)
  (map (λ (sub-tree)
         (if (pair? sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor)))
       tree))
```

Exercise 2.30: Define a procedure `square-tree` analogous to the `square-list` procedure of Exercise 2.21. That is, `square-tree` should behave as follows:

```
(square-tree
 (list 1
       (list 2 (list 3 4) 5)
       (list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

```
;; Exercise 2.30 (direct)

(define (square-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (sq tree))
        (else (cons (square-tree (car tree))
                    (square-tree (cdr tree))))))
```

```
;; Exercise 2.30 (map)

(define (square-tree tree)
  (map (λ (sub-tree)
         (if (pair? sub-tree)
             (square-tree sub-tree)
             (sq sub-tree)))
       tree))
```

Exercise 2.31: Abstract your answer to Exercise 2.30 to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```
(define (square-tree tree) (tree-map square tree))
```

```
;; Exercise 2.31

(define (tree-map tree proc)
  (map (λ (sub-tree)
         (if (pair? sub-tree)
             (tree-map sub-tree proc)
             (proc sub-tree)))
       tree))

(define (square-tree tree) (tree-map tree sq))
(define (scale-tree tree factor) (tree-map tree (λ (x) (* x factor))))
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

“cdr-ing down the list”

Hal Abelson

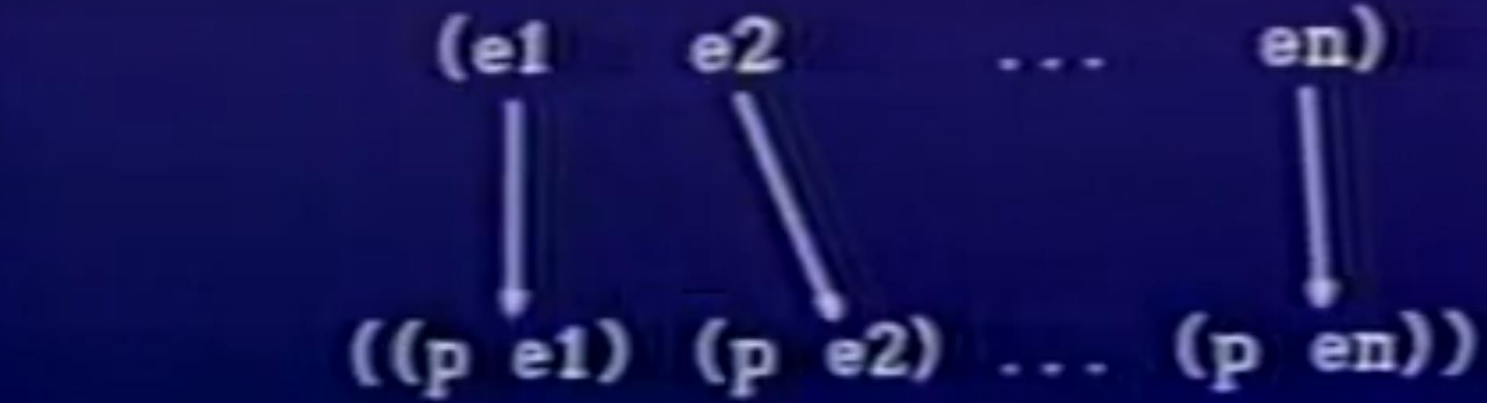
Lecture 3A, SICP

CDR-ing down a list

```
(define (scale-list s l)
  (if (null? l)
      nil
      (cons ((* (car l) s)
             (scale-list s (cdr l))))))
```

FIRST ELEMENT OF SCALED LIST

REST OF SCALED LIST




```
(define (map p l)
  (if (null? l)
      nil
      (cons (p (car l))
            (map p (cdr l)))))
```

APPLY P TO FIRST ELEMENT

MAP DOWN THE REST OF THE LIST

“once you start thinking in terms of **map** …
you stop thinking about the particular
control structure or order.
This is a **very, very important idea** …

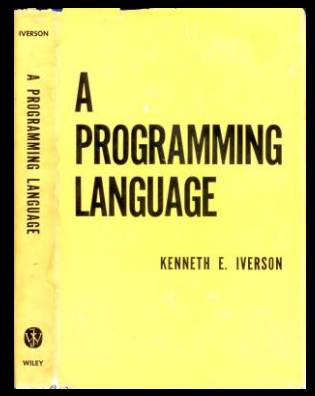
Hal Abelson
Lecture 3A, SICP

"… it really comes out of **APL**. It is the really important idea in **APL** - that you stop thinking about control structures and you start thinking about **operations on aggregates**.

Hal Abelson
Lecture 3A, SICP

Who is Kenneth Iverson?

- Creator of APL
- Co-creator of J
- Won the 1979 Turing Award
- Author or 1962 “A Programming Language”
- Bachelors at Queens, PhD at Harvard where he developed Iverson Notation
- Worked at IBM in 70s, 80s then left to IPSA
- Bill Gates visited IPSA in the early 80s to inquire about the possibility of APL on PC



Queen's UNIVERSITY

IBM

HARVARD

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
©1972, Association for Computing Machinery, Inc.

NUMERICAL SCIENTIFIC

Two-Dimensional ➤

ON-LINE

NON-NUMERICAL SCIENTIFIC (= Formula Manipulation)

BUSINESS DATA PROCESSING

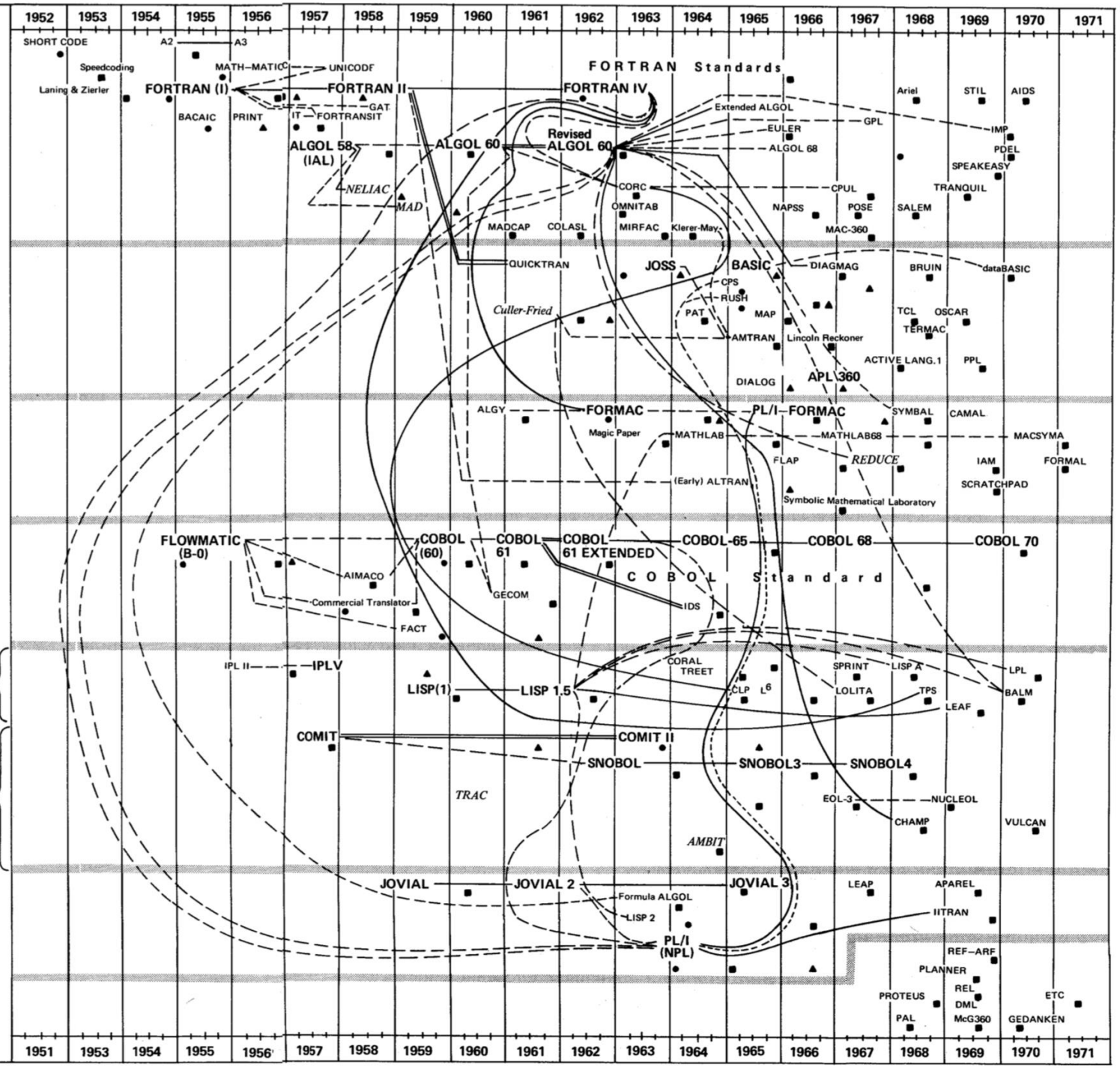
List Processing

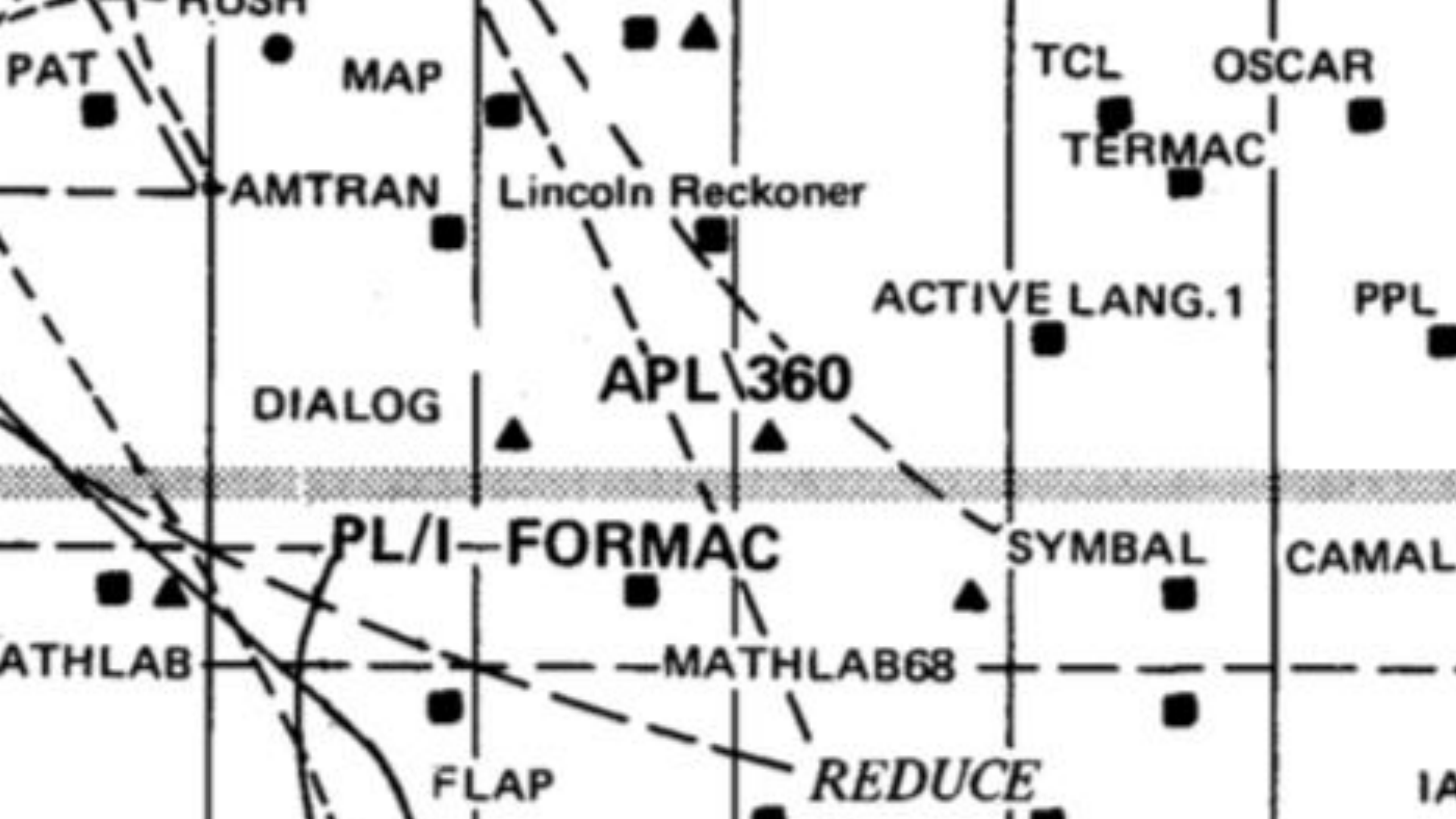
STRING AND LIST PROCESSING

String Processing

MULTIPURPOSE

EXPERIMENTAL AND OTHERS




PAT
MAP
TCL
OSCAR
TERMAC
AMTRAN
Lincoln Reckoner
ACTIVE LANG.1
PPL
APL\360
DIALOG
PL/I
FORMAC
SYMBAL
CAMAL
ATHLAB
MATHLAB68
FLAP
REDUCE

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
©1972, Association for Computing Machinery, Inc.

NUMERICAL SCIENTIFIC

Two-Dimensional ➤

ON-LINE

NON-NUMERICAL SCIENTIFIC (= Formula Manipulation)

BUSINESS DATA PROCESSING

List Processing

STRING AND LIST PROCESSING

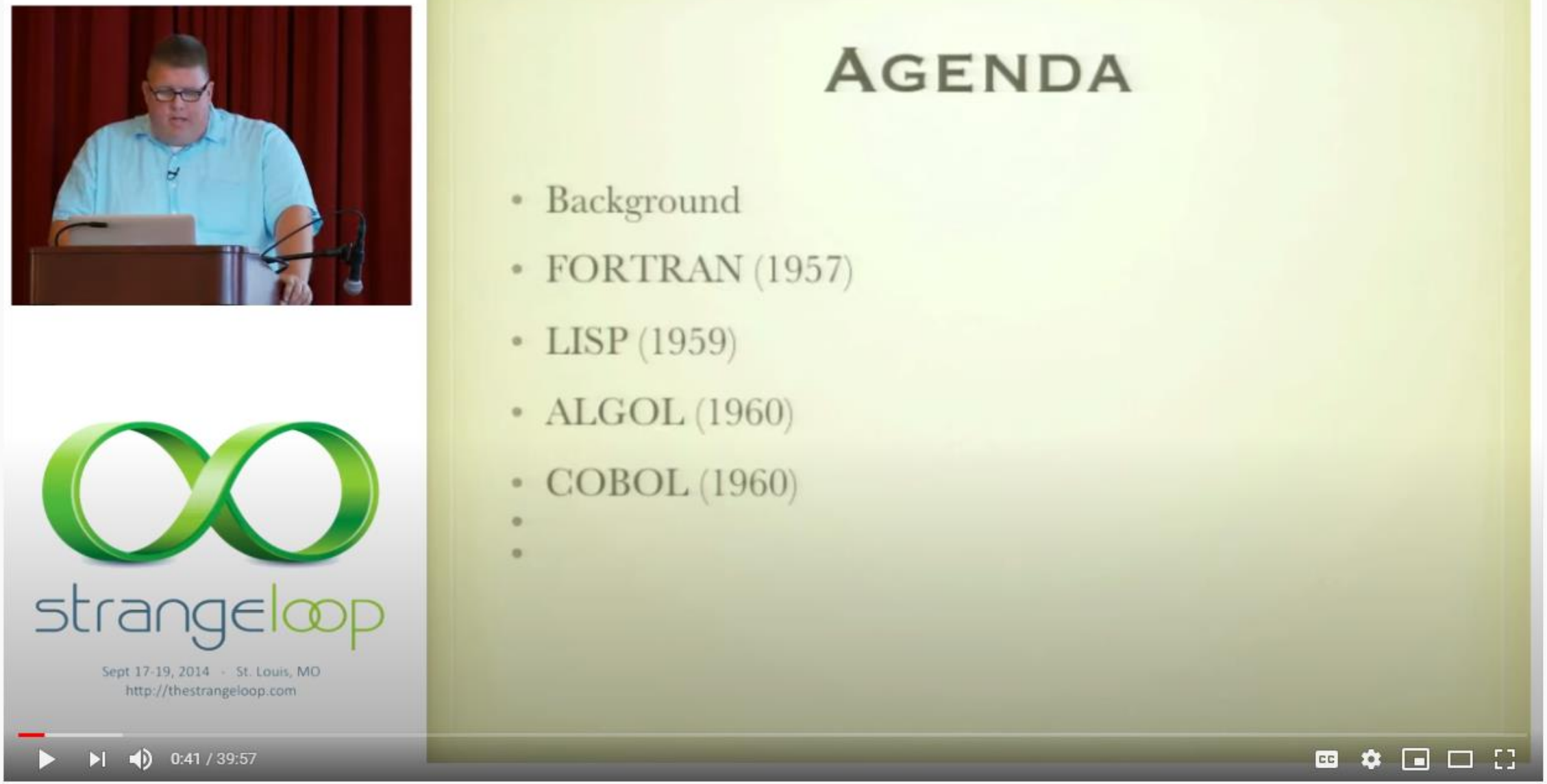
String Processing

MULTIPURPOSE

EXPERIMENTAL AND OTHERS

1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971

SHORT CODE
A2 — A3
Speedcoding
Laning & Zierler
MATH-MATIC
UNICODE
FORTRAN (I)
FORTRAN II
GAT
ALGOL 58 (IAL)
NELIAC
MAD
FORTRAN Standards
FORTRAN IV
ALGOL 60
Revised ALGOL 60
Extended ALGOL
EULER
GPL
ALGOL 68
Ariel
STIL
AIDS
IMP
PDEL
SPEAKEASY
TRANQUIL
CORC
CPUL
OMNITAB
NAPSS
POSE
SALEM
MIRFAC
Klerer-May
MAC-360
MADCAP
COLASL
QUICKTRAN
JOSS
BASIC
DIAGMAG
BRUIN
dataBASIC
CPS
RUSH
PAT
MAP
TCL
OSCAR
TERMAC
Culler-Fried
AMTRAN
Lincoln Reckoner
APL\360
PPL
CAMAL
ALGY
FORMAC
Magic Paper
MATHLAB
MATHLAB68
MACSYMA
FLAP
REDUCE
IAM
FORMAL
SCRATCHPAD
(Early) ALTRAN
Symbolic Mathematical Laboratory
FLOWMATIC (B-0)
COBOL (60)
COBOL 61 EXTENDED
COBOL-65
COBOL 68
COBOL 70
C O B O L S t a n d a r d
Commercial Translator
GECOM
IDS
FACT
IPL II
IPLV
LISP(1)
LISP 1.5
CORAL
TREET
CLP
L6
SPRINT
LISP A
LOLITA
TPS
LPL
BALM
LEAF
COMIT
COMIT II
SNOBOL
SNOBOL3
SNOBOL4
TRAC
EOL-3
NUCLEOL
CHAMP
VULCAN
AMBIT
JOVIAL
JOVIAL 2
JOVIAL 3
Formula ALGOL
LISP 2
PL/I (NPL)
LEAP
APAREL
IITRAN
REF-ARF
PLANNER
REL
DML
PROTEUS
PAL
McG360
GEDANKEN
ETC



"All of this has happened before, and it will all happen again." by Mark Allen

6,343 views • Sep 21, 2014

106 2 SHARE SAVE ...

Proceedings of the ACM on Programming Languages

Search with

Proceedings of the ACM on Programming Languages

Association for Computing Machinery

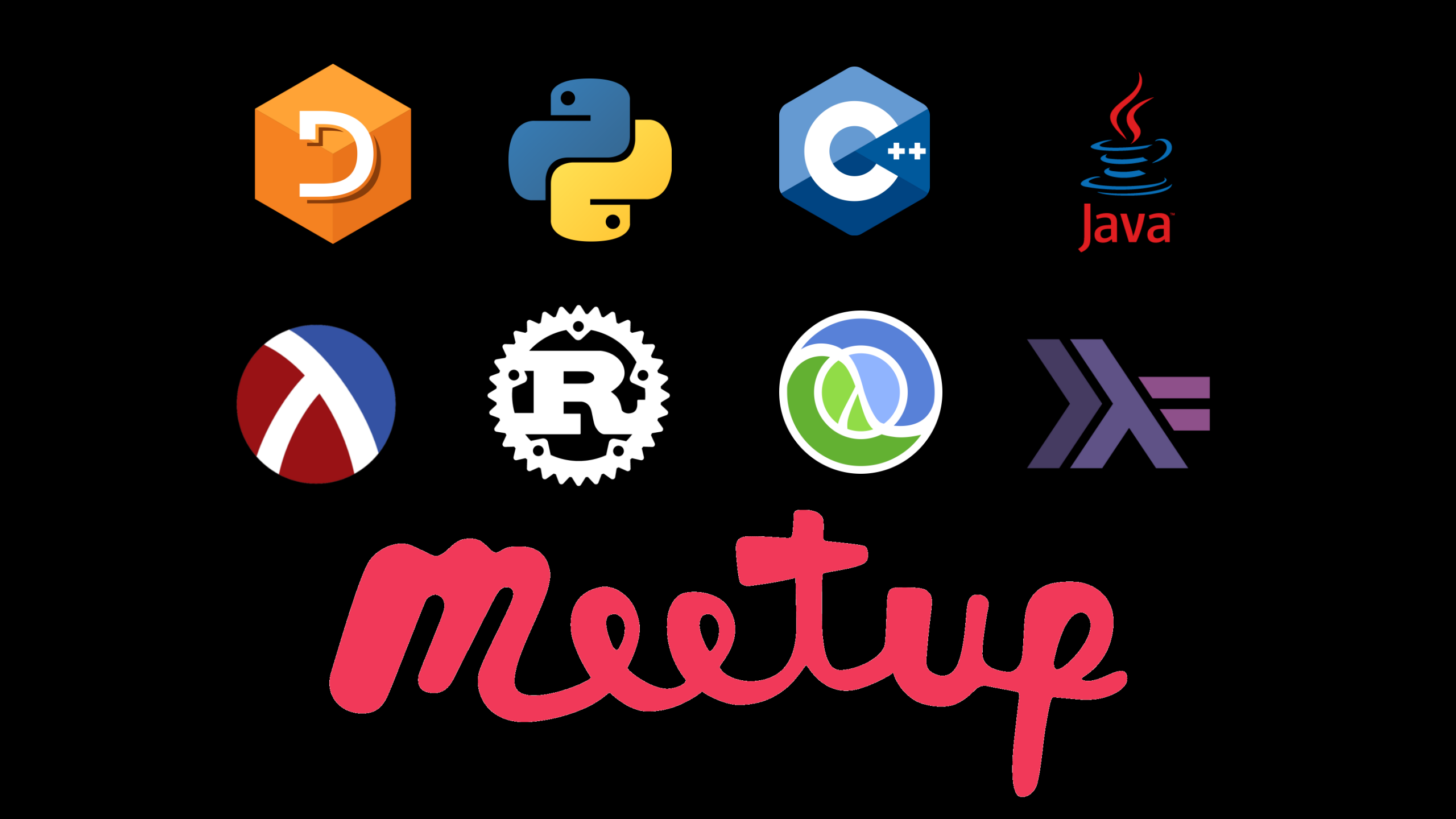
Proceedings of the ACM on Programming Languages (PACMPL) is a Gold Open Access journal publishing research on all aspects of programming languages, from design to implementation and from mathematical formalisms to empirical studies. Each issue of the journal is devoted to a particular subject area within ... (More)

APL Since 1978

ROGER K.W. HUI, Dyalog Limited, Canada
MORTEN J. KROMBERG, Dyalog Limited, United Kingdom

Shepherd: Guy L. Steele Jr., Oracle Labs, USA

The Evolution of APL, the HOPL I paper by Falkoff and Iverson on APL, recounted the fundamental design principles which shaped the implementation of the APL language in 1966, and the early uses and other influences which shaped its first decade of enhancements.


Java
meetup