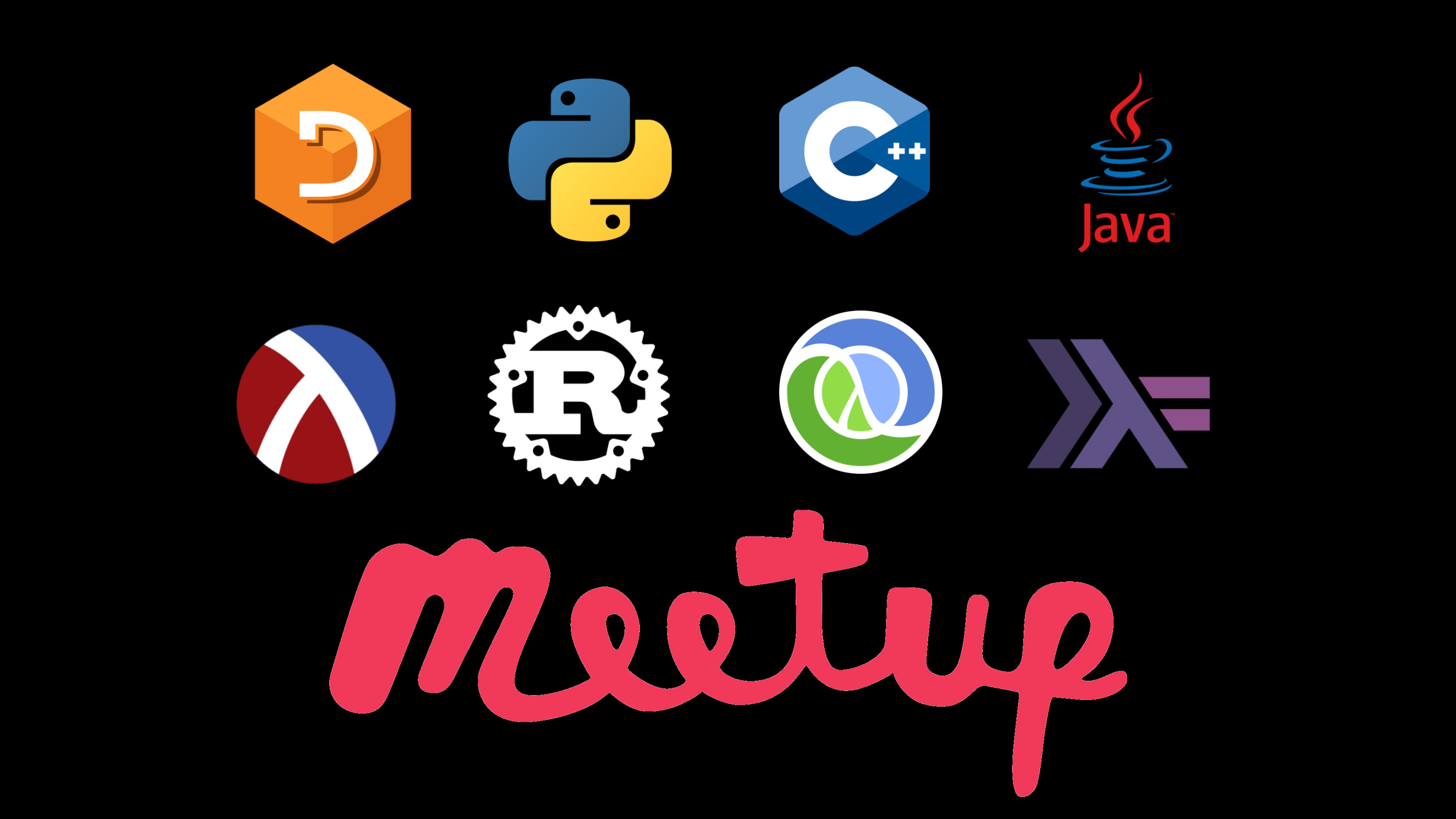
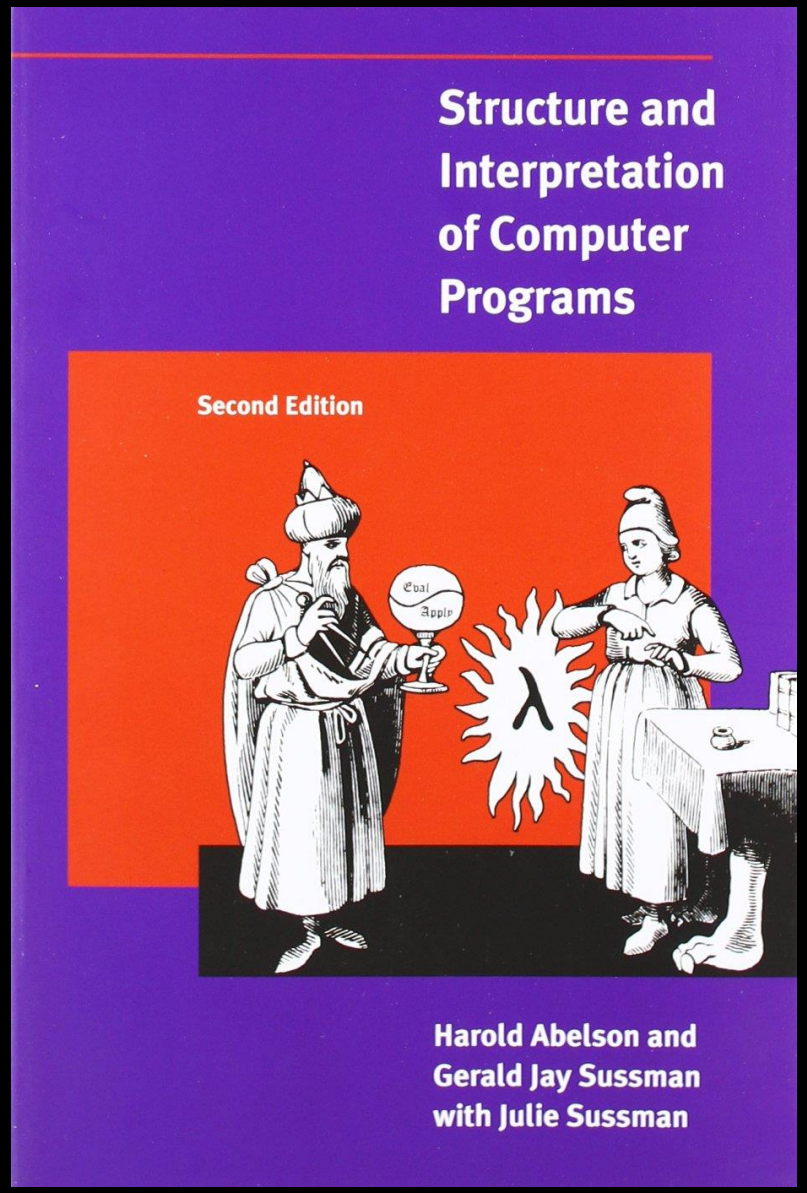

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 4.4

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

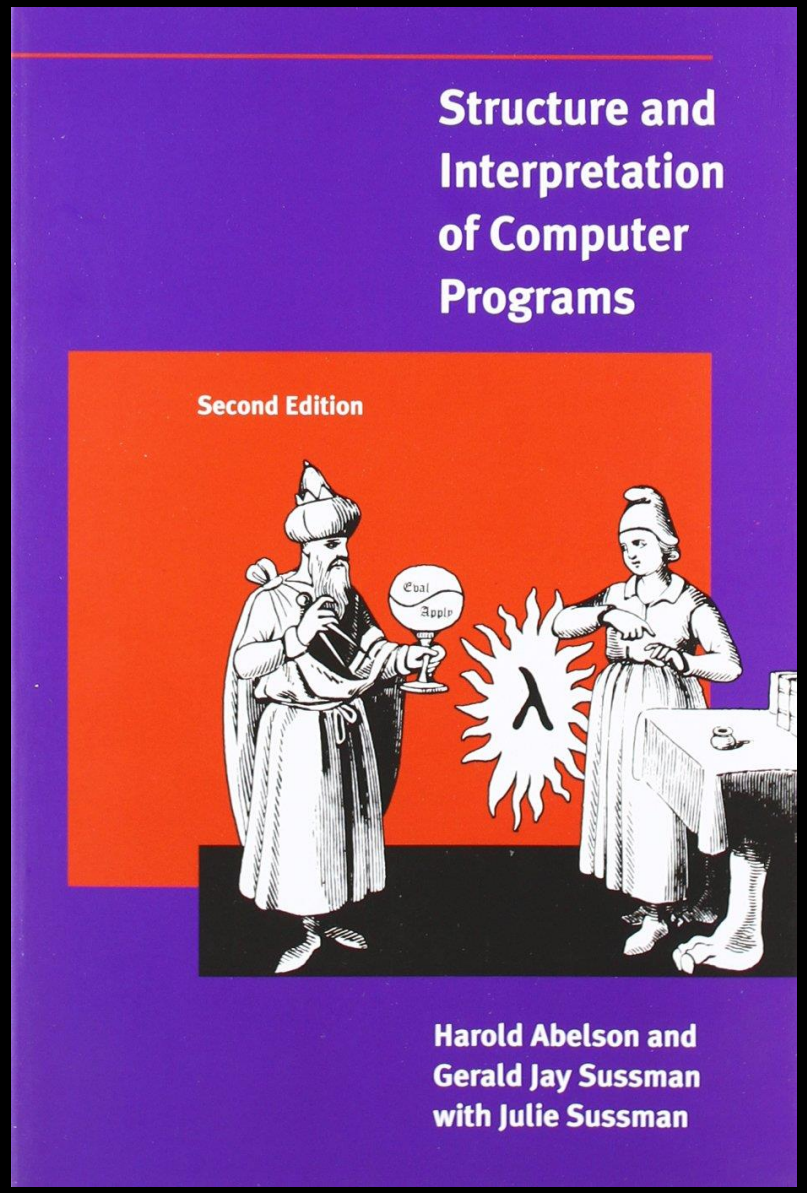
Following

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....

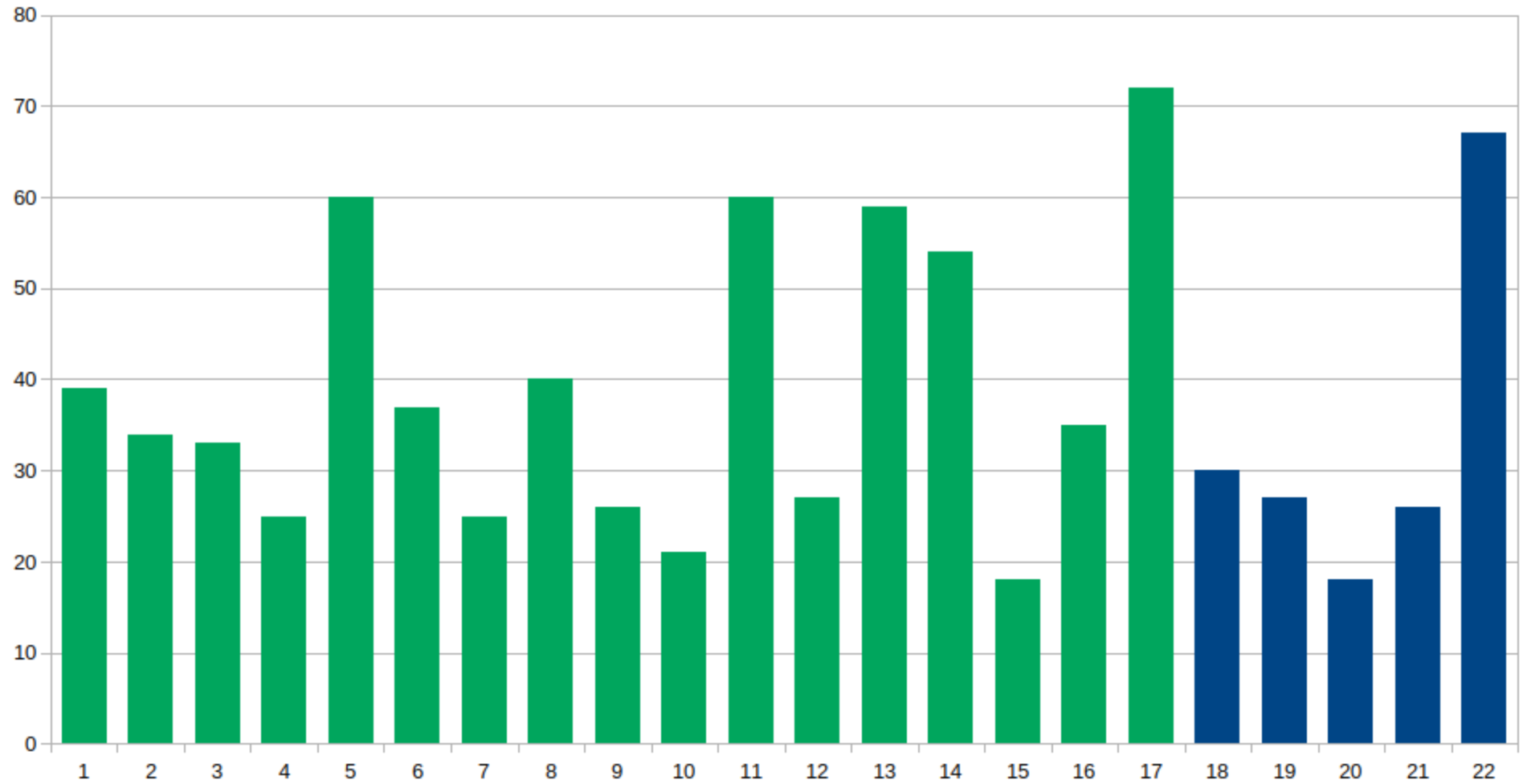
Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 4.4

4.4 Logic Programming 594
4.4.1 Deductive Information Retrieval 599
4.4.2 How the Query System Works 615
4.4.3 Is Logic Programming Mathematical Logic? . . 627
4.4.4 Implementing the Query System 635
4.4.4.1 The Driver Loop and Instantiation . . 636
4.4.4.2 The Evaluator 638
4.4.4.3 Finding Assertions by Pattern Matching 642
4.4.4.4 Rules and Unification 645
4.4.4.5 Maintaining the Data Base 651
4.4.4.6 Stream Operations 654
4.4.4.7 Query Syntax Procedures 656
4.4.4.8 Frames and Bindings 659


80
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

The nondeterministic program evaluator of Section 4.3 also moves away from the view that programming is about constructing algorithms for computing unidirectional functions. In a nondeterministic language, expressions can have more than one value, and, as a result, the computation is dealing with relations rather than with single-valued functions. Logic programming extends this idea by combining a relational vision of programming with a powerful kind of symbolic pattern matching called *unification*.[58]

[58]Logic programming has grown out of a long history of research in automatic theorem proving. Early theorem-proving programs could accomplish very little, because they exhaustively searched the space of possible proofs. The major breakthrough that made such a search plausible was the discovery in the early 1960s of the *unification algorithm* and the *resolution principle* (Robinson 1965). Resolution was used, for example, by Green and Raphael (1968) (see also Green 1969) as the basis for a deductive question-answering system. During most of this period, researchers concentrated on algorithms that are guaranteed to find a proof if one exists. Such algorithms were difficult to control and to direct toward a proof. Hewitt (1969) recognized the possibility of merging the control structure of a programming language with the operations of a logic-manipulation system, leading to the work in automatic search mentioned in Section 4.3.1 (Footnote 4.47). At the same time that this was being done, Colmerauer, in Marseille, was developing rule-based systems for manipulating natural language (see Colmerauer et al. 1973). He invented a programming language called Prolog for representing those rules. Kowalski (1973; 1979), in Edinburgh, recognized that execution of a Prolog program could be interpreted as proving theorems (using a proof technique called linear Horn-clause resolution). The merging of the last two strands led to the logic-programming movement. Thus, in assigning credit for the development of logic programming, the French can point to Prolog's genesis at the University of Marseille, while the British can highlight the work at the University of Edinburgh. According to people at MIT, logic programming was developed by these groups in an attempt to figure out what Hewitt was talking about in his brilliant but impenetrable Ph.D. thesis. For a history of logic programming, see Robinson 1983.

Earlier in this chapter we explored the technology of implementing interpreters and described the elements that are essential to an interpreter for a Lisp-like language (indeed, to an interpreter for any conventional language). Now we will apply these ideas to discuss an interpreter for a logic programming language. We call this language the *query language*, because it is very useful for retrieving information from data bases by formulating *queries*, or questions, expressed in the language.

4.4 Logic Programming 594
4.4.1 Deductive Information Retrieval 599
4.4.2 How the Query System Works 615
4.4.3 Is Logic Programming Mathematical Logic? . . 627
4.4.4 Implementing the Query System 635
4.4.4.1 The Driver Loop and Instantiation . . 636
4.4.4.2 The Evaluator 638
4.4.4.3 Finding Assertions by Pattern Matching 642
4.4.4.4 Rules and Unification 645
4.4.4.5 Maintaining the Data Base 651
4.4.4.6 Stream Operations 654
4.4.4.7 Query Syntax Procedures 656
4.4.4.8 Frames and Bindings 659

```
(define database-assertions
  '((address (Warbucks Oliver) (Swellesley (Top Head Road)))
    (job (Warbucks Oliver) (administration big wheel))
    (salary (Warbucks Oliver) 150000)

    (address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
    (job (Bitdiddle Ben) (computer wizard))
    (salary (Bitdiddle Ben) 60000)
    (supervisor (Bitdiddle Ben) (Warbucks Oliver))

    (address (Hacker Alyssa P) (Cambridge (Mass Ave) 78))
    (job (Hacker Alyssa P) (computer programmer))
    (salary (Hacker Alyssa P) 40000)
    (supervisor (Hacker Alyssa P) (Bitdiddle Ben))

    (address (Fect Cy D) (Cambridge (Ames Street) 3))
    (job (Fect Cy D) (computer programmer))
    (salary (Fect Cy D) 35000)
    (supervisor (Fect Cy D) (Bitdiddle Ben))
```

```
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa P))

(address (Scrooge Eben) (Westen (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))

(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(job (Aull DeWitt) (administration secretary))
(salary (Aull DeWitt) 25000)
(supervisor (Aull DeWitt) (Warbucks Oliver))

(can-do-job (computer wizard) (computer programmer))
(can-do-job (computer wizard) (computer technician))
(can-do-job (computer programmer) (computer programmer trainee))
(can-do-job (administration secretary) (administration big wheel))
```

```
(rule (lives-near ?person-1 ?person-2)
      (and (address ?person-1 (?town . ?rest-1))
           (address ?person-2 (?town . ?rest-2))
           (not (same ?person-1 ?person-2))))

(rule (same ?x ?x))

(rule (wheel ?person)
      (and (supervisor ?middle-manager ?person)
           (supervisor ?x ?middle-manager)))
```

Exercise 4.55: Give simple queries that retrieve the following information from the data base:

1. all people supervised by Ben Bitdiddle;
2. the names and jobs of all people in the accounting division;
3. the names and addresses of all people who live in Slumerville.

1. all people supervised by Ben Bitdiddle;

```
;;; Query input:
(supervisor ?x (Bitdiddle Ben))

;;; Query results:
(supervisor (Tweakit Lem E) (Bitdiddle Ben))
(supervisor (Fect Cy D) (Bitdiddle Ben))
(supervisor (Hacker Alyssa P) (Bitdiddle Ben))
```

2. the names and jobs of all people in the accounting division;

```
;;; Query input:
(job ?x (accounting . ?y))

;;; Query results:
(job (Cratchet Robert) (accounting scrivener))
(job (Scrooge Eben) (accounting chief accountant))
```

3. the names and addresses of all people who live in Slumerville.

```
;;; Query input:
(address ?x (Slumerville . ?y))

;;; Query results:
(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(address (Bitdiddle Ben) (Slumerville (Ridge Road) 10))
```

Exercise 4.58: Define a rule that says that a person is a "big shot" in a division if the person works in the division but does not have a supervisor who works in the division.

```
;; Exercise 4.58 (page 610)

;;; Query input:
 (assert! (rule (bigshot ?x ?div)
                (and (job ?x (?div . ?rest))
                     (or (not (supervisor ?x ?y))
                         (and (supervisor ?x ?y)
                              (not (job ?y (?div . ?r))))))))

;; Assertion added to data base.

;;; Query input:
(bigshot . ?x)

;;; Query results:
(bigshot (Warbucks Oliver) administration)
(bigshot (Scrooge Eben) accounting)
(bigshot (Bitdiddle Ben) computer)
```

4.4 Logic Programming 594

4.4.1 Deductive Information Retrieval 599

4.4.2 How the Query System Works 615

4.4.3 Is Logic Programming Mathematical Logic? . . 627

4.4.4 Implementing the Query System 635

4.4.4.1 The Driver Loop and Instantiation . . 636

4.4.4.2 The Evaluator 638

4.4.4.3 Finding Assertions by Pattern Matching 642

4.4.4.4 Rules and Unification 645

4.4.4.5 Maintaining the Data Base 651

4.4.4.6 Stream Operations 654

4.4.4.7 Query Syntax Procedures 656

4.4.4.8 Frames and Bindings 659

It should be apparent that the query evaluator must perform some kind of search in order to match queries against facts and rules in the data base. One way to do this would be to implement the query system as a nondeterministic program, using the `amb` evaluator of Section 4.3 (see Exercise 4.78). Another possibility is to manage the search with the aid of streams. Our implementation follows this second approach.

Pattern matching

A *pattern matcher* is a program that tests whether some datum fits a specified pattern. For example, the data list `((a b) c (a b))` matches the pattern `(?x c ?x)` with the pattern variable `?x` bound to `(a b)`. The same data list matches the pattern `(?x ?y ?z)` with `?x` and `?z` both bound to `(a b)` and `?y` bound to `c`. It also matches the pattern `((?x ?y) c (?x ?y))` with `?x` bound to `a` and `?y` bound to `b`. However, it does not match the pattern `(?x a ?y)`, since that pattern specifies a list whose second element is the symbol `a`.

Streams of frames

The testing of patterns against frames is organized through the use of streams. Given a single frame, the matching process runs through the data-base entries one by one. For each data-base entry, the matcher generates either a special symbol indicating that the match has failed or an extension to the frame. The results for all the data-base entries are collected into a stream, which is passed through a filter to weed out the failures. The result is a stream of all the frames that extend the given frame via a match to some assertion in the data base.[67]

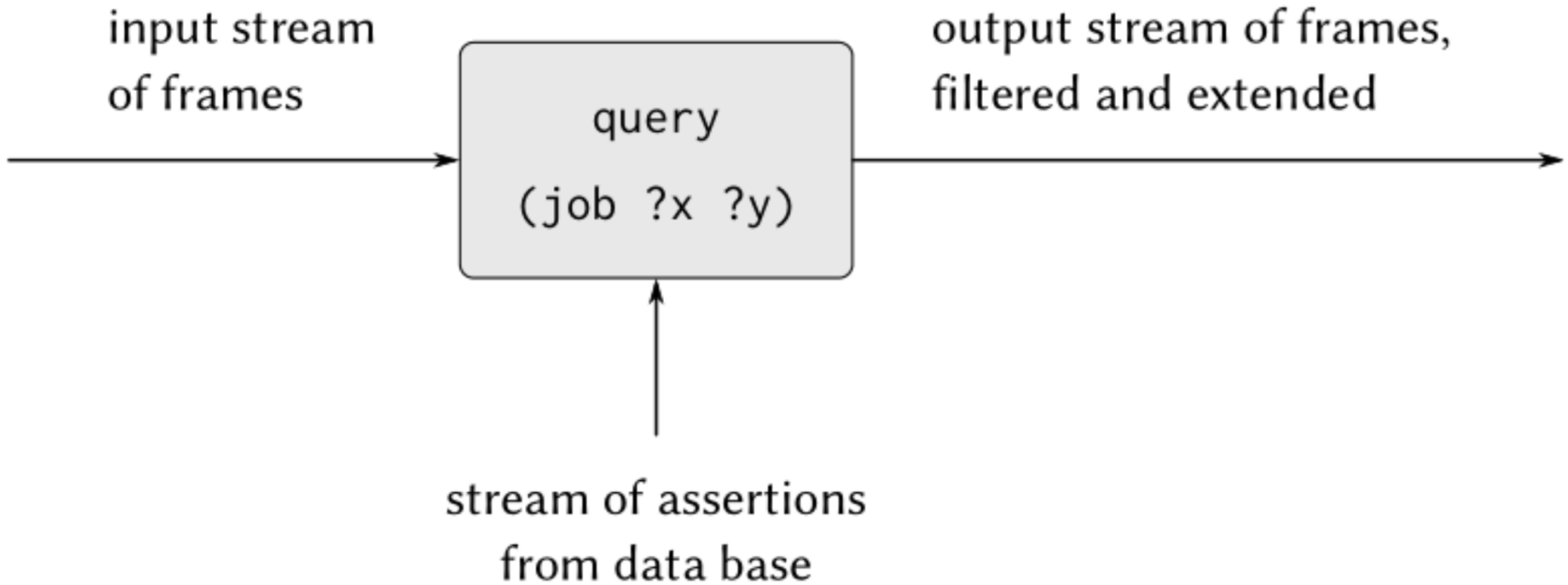


Figure 4.4: A query processes a stream of frames.

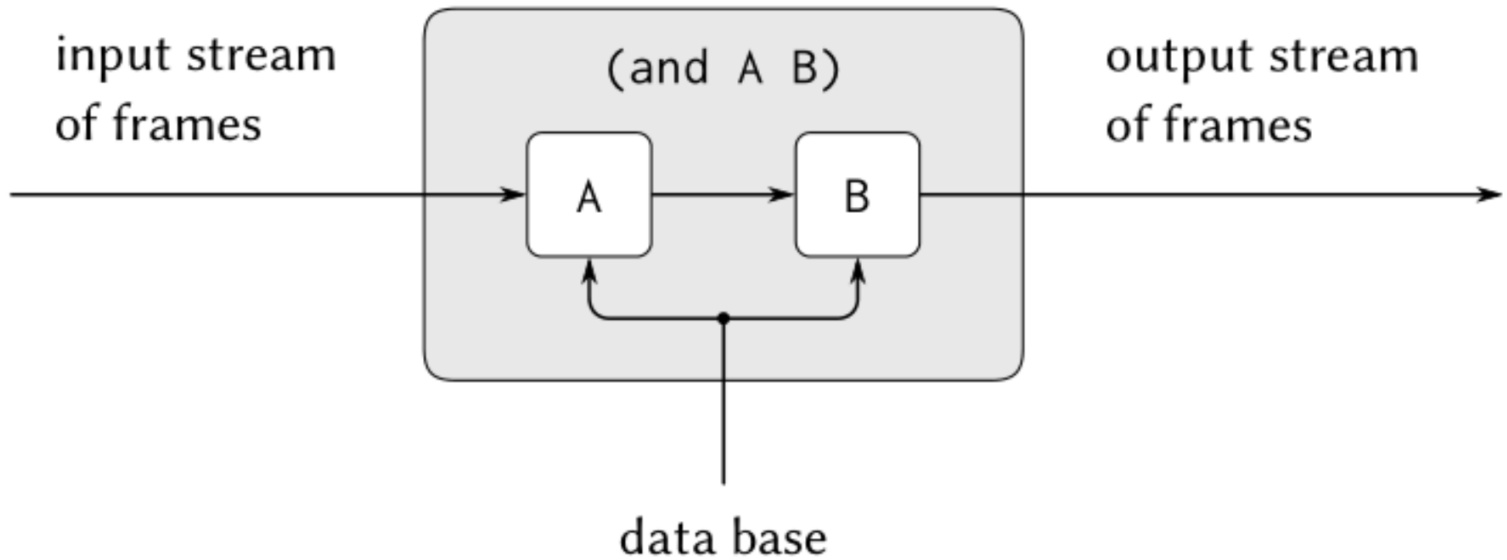


Figure 4.5: The and combination of two queries is produced by operating on the stream of frames in series.

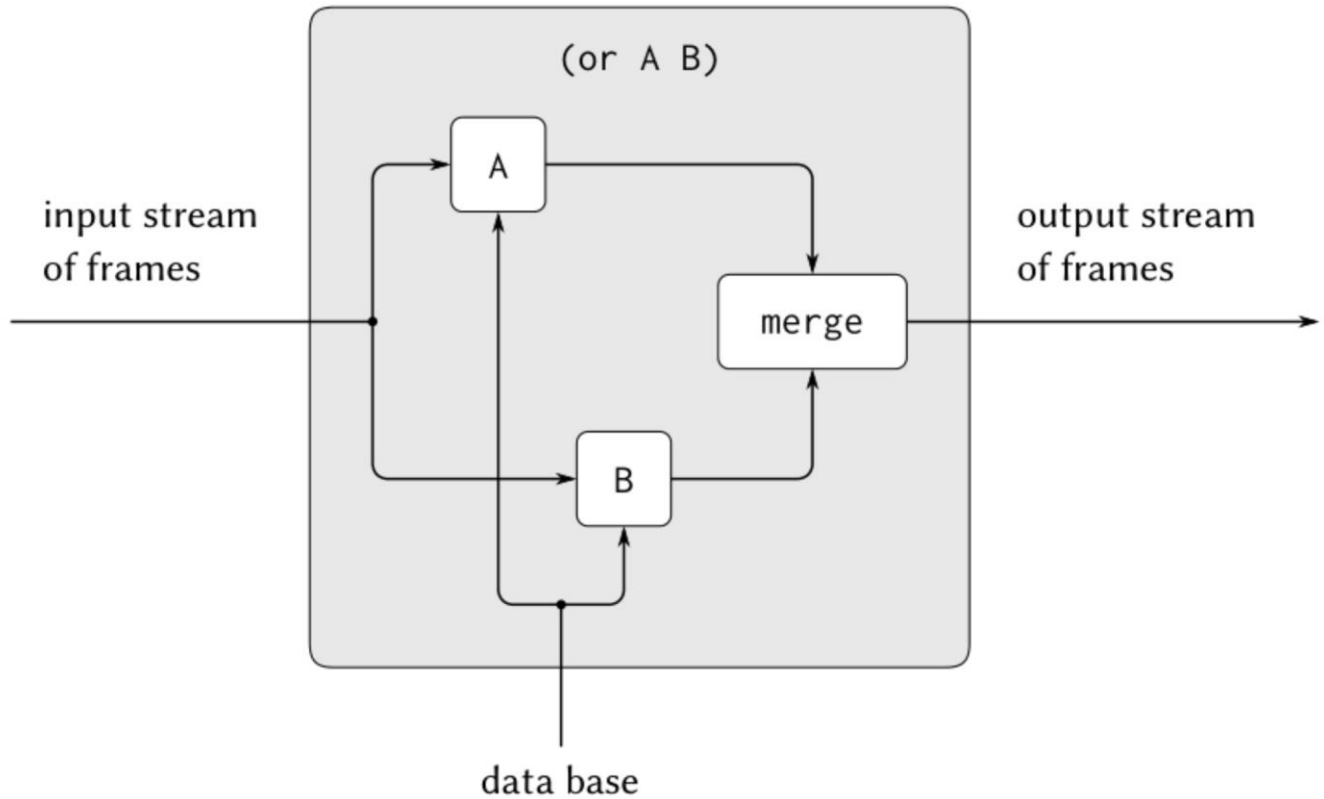


Figure 4.6: The `or` combination of two queries is produced by operating on the stream of frames in parallel and merging the results.

Unification

In order to handle rules in the query language, we must be able to find the rules whose conclusions match a given query pattern. Rule conclusions are like assertions except that they can contain variables, so we will need a generalization of pattern matching—called *unification*—in which both the "pattern" and the "datum" may contain variables.

A unifier takes two patterns, each containing constants and variables, and determines whether it is possible to assign values to the variables that will make the two patterns equal. If so, it returns a frame containing these bindings. For example, unifying `(?x a ?y)` and `(?y ?z a)` will specify a frame in which `?x`, `?y`, and `?z` must all be bound to `a`. On the other hand, unifying `(?x ?y a)` and `(?x b ?y)` will fail, because there is no value for `?y` that can make the two patterns equal. (For the second elements of the patterns to be equal, `?y` would have to be `b`; however, for the third elements to be equal, `?y` would have to be `a`.) The unifier used in the query system, like the pattern matcher, takes a frame as input and performs unifications that are consistent with this frame.

4.4 Logic Programming 594

4.4.1 Deductive Information Retrieval 599

4.4.2 How the Query System Works 615

4.4.3 Is Logic Programming Mathematical Logic? . . 627

4.4.4 Implementing the Query System 635

4.4.4.1 The Driver Loop and Instantiation . . 636

4.4.4.2 The Evaluator 638

4.4.4.3 Finding Assertions by Pattern Matching 642

4.4.4.4 Rules and Unification 645

4.4.4.5 Maintaining the Data Base 651

4.4.4.6 Stream Operations 654

4.4.4.7 Query Syntax Procedures 656

4.4.4.8 Frames and Bindings 659

There is also a much more serious way in which the `not` of the query language differs from the `not` of mathematical logic. In logic, we interpret the statement "not *P*" to mean that *P* is not true. In the query system, however, "not *P*" means that *P* is not deducible from the knowledge in the data base. For example, given the personnel data base of Section 4.4.1, the system would happily deduce all sorts of `not` statements, such as that Ben Bitdiddle is not a baseball fan, that it is not raining outside, and that 2 + 2 is not 4.[78] In other words, the `not` of logic programming languages reflects the so-called *closed world assumption* that all relevant information has been included in the data base.[79]

4.4 Logic Programming 594

4.4.1 Deductive Information Retrieval 599

4.4.2 How the Query System Works 615

4.4.3 Is Logic Programming Mathematical Logic? . . 627

4.4.4 Implementing the Query System 635

4.4.4.1 The Driver Loop and Instantiation . . 636

4.4.4.2 The Evaluator 638

4.4.4.3 Finding Assertions by Pattern Matching 642

4.4.4.4 Rules and Unification 645

4.4.4.5 Maintaining the Data Base 651

4.4.4.6 Stream Operations 654

4.4.4.7 Query Syntax Procedures 656

4.4.4.8 Frames and Bindings 659

```
;; 4.4.4.2 The Evaluator

(define (qeval query frame-stream)
  (let ((qproc (get (type query) 'qeval)))
    (if qproc
        (qproc (contents query) frame-stream)
        (simple-query query frame-stream))))
```

```
(put 'and 'qeval conjoin)
(put 'or  'qeval disjoin)
(put 'not 'qeval negate)

(put 'lisp-value 'qeval lisp-value)

(put 'always-true 'qeval always-true)
```

```
(define (simple-query query-pattern frame-stream)
  (stream-flatmap
   (lambda (frame)
     (stream-append-delayed
      (find-assertions query-pattern frame)
      (delay (apply-rules query-pattern frame))))
   frame-stream))
```

```
(define (conjoin conjuncts frame-stream)
  (if (empty-conjunction? conjuncts)
      frame-stream
      (conjoin (rest-conjuncts conjuncts)
               (qeval (first-conjunct conjuncts) frame-stream))))

(define (disjoin disjuncts frame-stream)
  (if (empty-disjunction? disjuncts)
      the-empty-stream
      (interleave-delayed
       (qeval (first-disjunct disjuncts) frame-stream)
       (delay (disjoin (rest-disjuncts disjuncts) frame-stream)))))
```

```
(define (negate operands frame-stream)
  (stream-flatmap
   (lambda (frame)
     (if (stream-null?
          (qeval (negated-query operands)
                 (singleton-stream frame)))
         (singleton-stream frame)
         the-empty-stream))
   frame-stream))
```

```
(define (add-rule-or-assertion! assertion)
  (if (rule? assertion)
      (add-rule! assertion)
      (add-assertion! assertion)))

(define (add-assertion! assertion)
  (store-assertion-in-index assertion)
  (let ((old-assertions THE-ASSERTIONS))
    (set! THE-ASSERTIONS
          (cons-stream assertion old-assertions))
    'ok))

(define (add-rule! rule)
  (store-rule-in-index rule)
  (let ((old-rules THE-RULES))
    (set! THE-RULES (cons-stream rule old-rules))
    'ok))
```

```
;; 4.4.4.9 Missing code

;; from page 521

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

;; from page 433

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

;; from former assignment

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

```
(define (add-to-data-base! assertions)
  (for-each add-rule-or-assertion! assertions))

(add-to-data-base! database-assertions)
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

"You should be drawing lessons on two levels. The first is to realize **just how different a language can be**. If you think that the jump from FORTRAN to LISP is a big deal, you haven't seen anything yet."

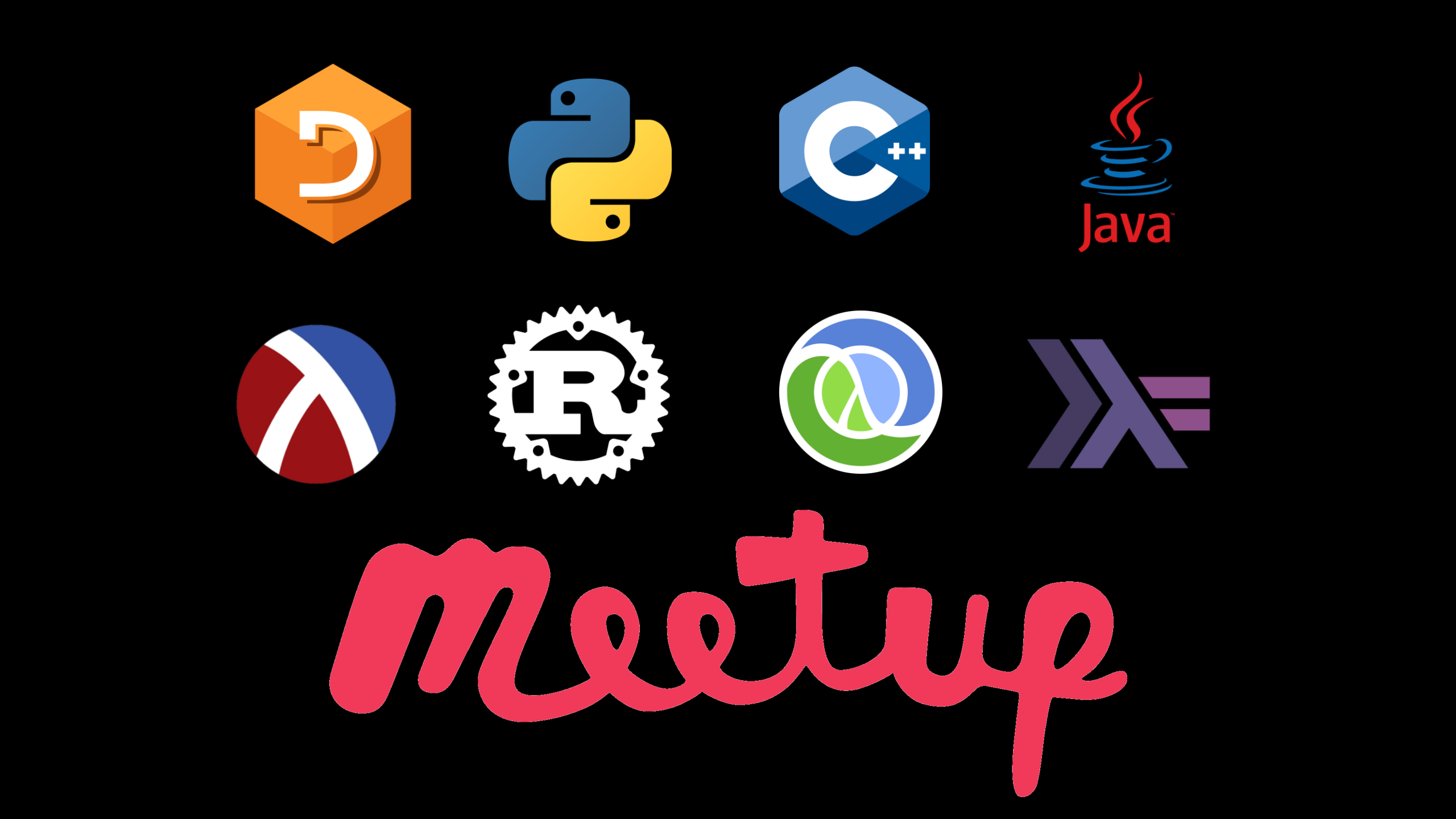
Gerald Sussman

Lecture 8A: Logic Programming

“This system works by **pattern matching**.”

Brian Harvey

L42 Logic Programming


Java
meetup