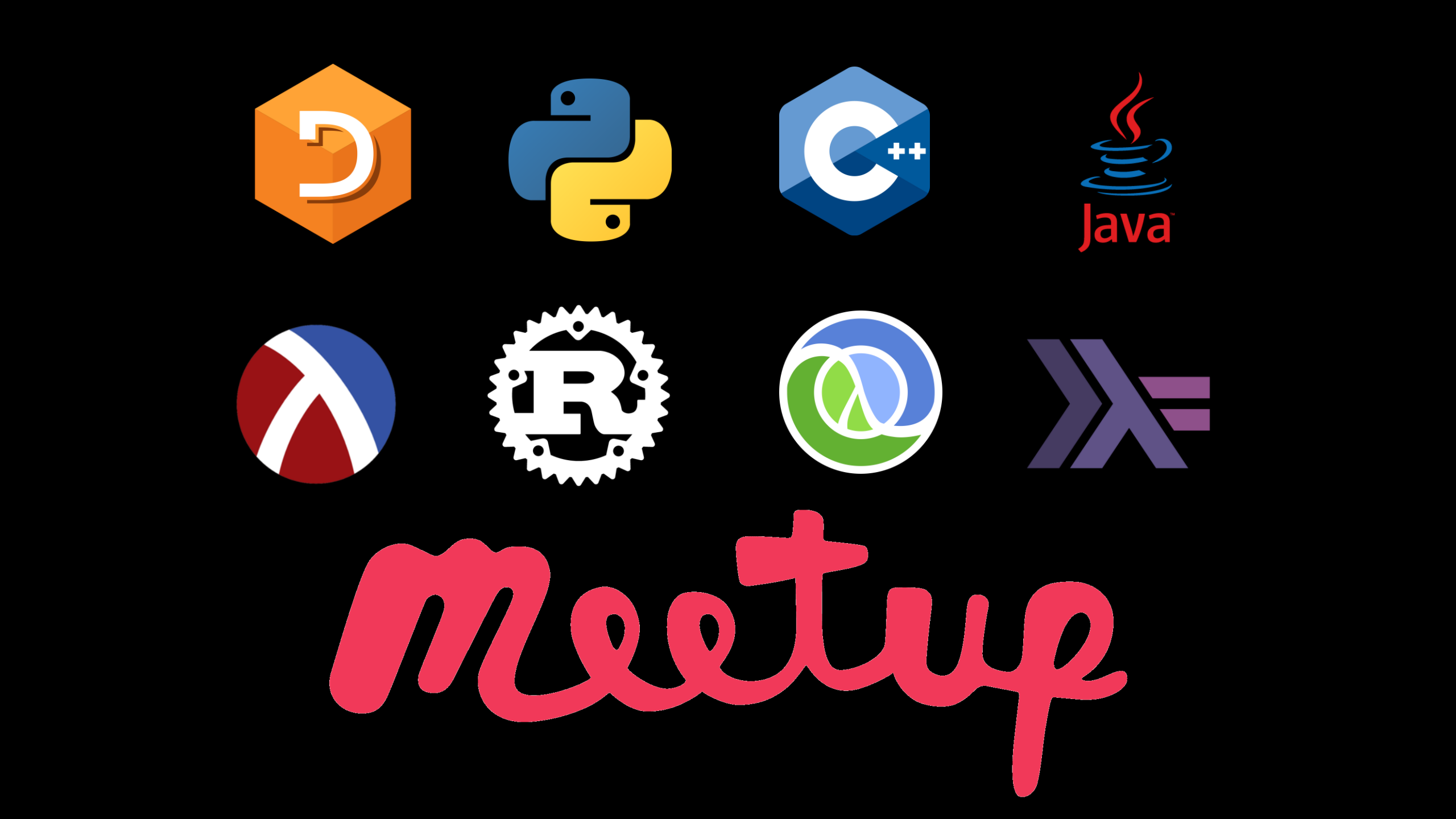
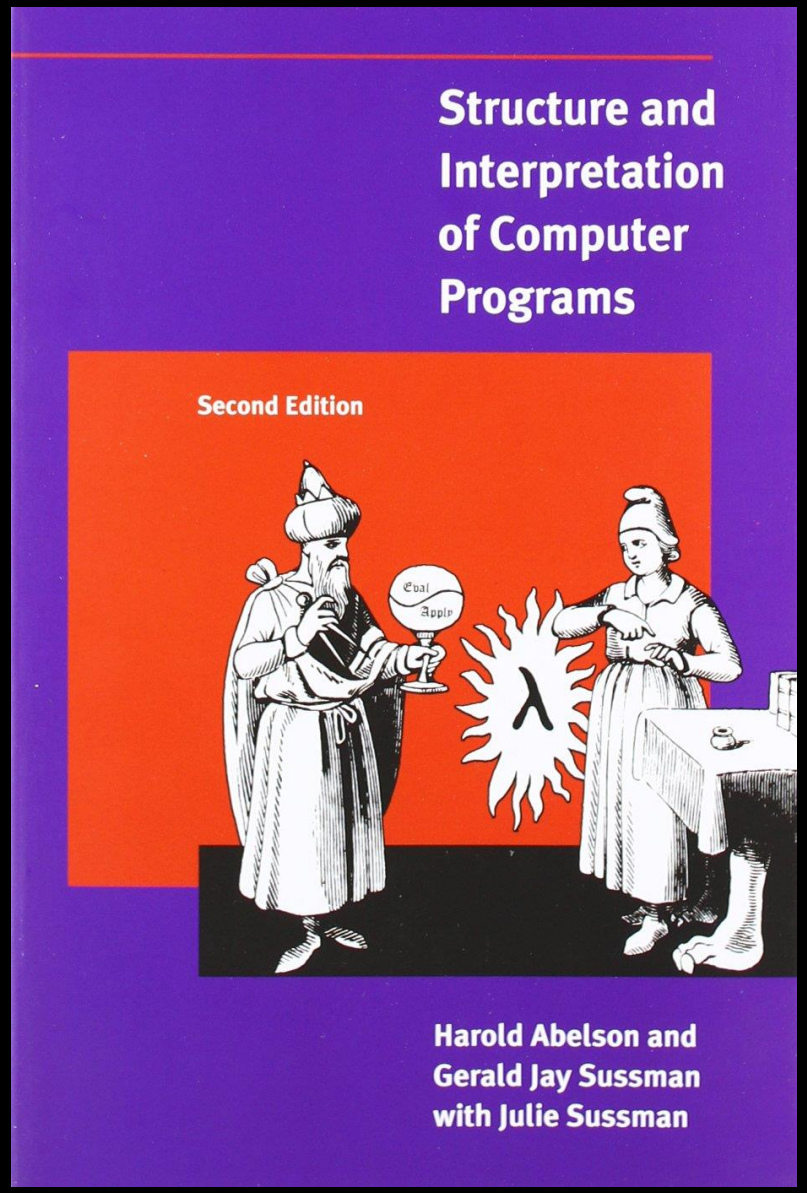

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 4.1

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

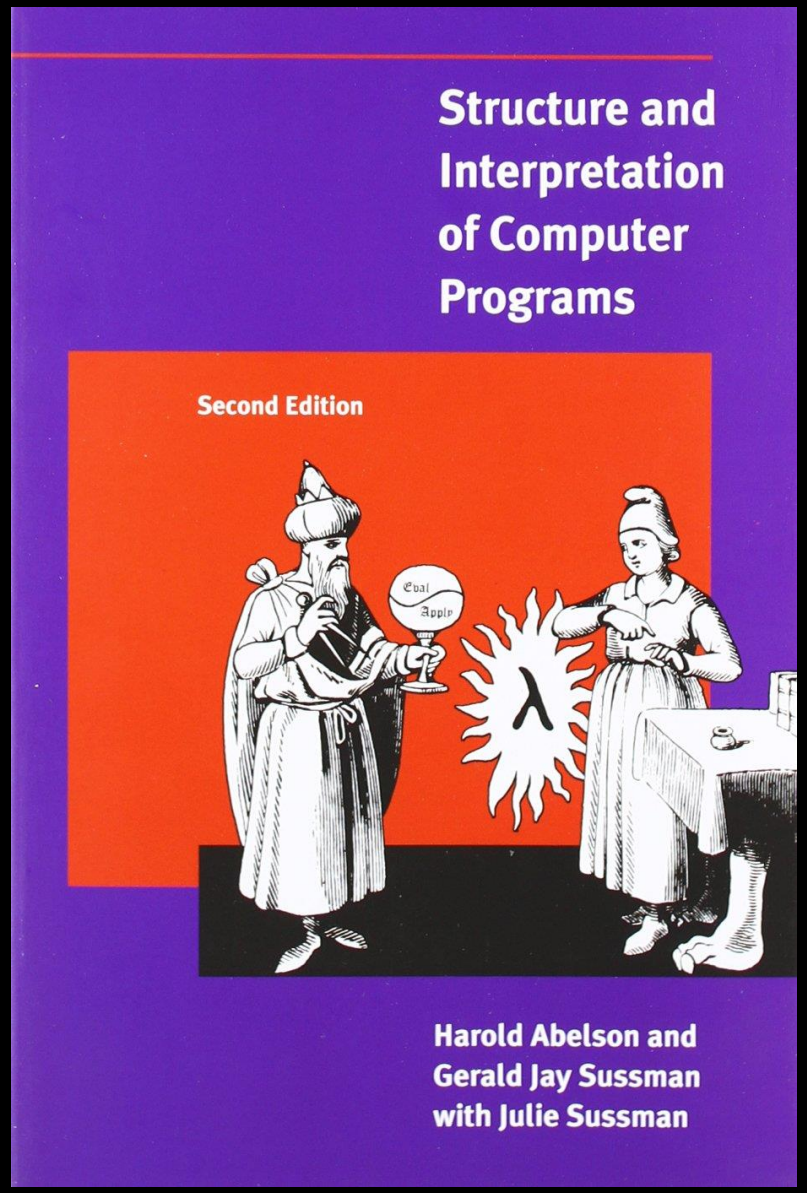
Following

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



Structure and Interpretation of Computer Programs

Chaper 4.1

“We are done the structure part and now we are starting on the interpretation part.”

Brian Harvey

L36, SICP

4.1 The Metacircular Evaluator

Our evaluator for Lisp will be implemented as a Lisp program. It may seem circular to think about evaluating Lisp programs using an evaluator that is itself implemented in Lisp. However, evaluation is a process, so it is appropriate to describe the evaluation process using Lisp, which, after all, is our tool for describing processes.[3] An evaluator that is written in the same language that it evaluates is said to be *metacircular*.

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help

Self-hosting (compilers)

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.

Find sources: "Self-hosting" compilers – news · newspapers · books · scholar · JSTOR *(April 2010) (Learn how and when to remove this template message)*

In computer programming, **self-hosting** is the use of a program as part of the toolchain or operating system that produces new versions of that same program—for example, a compiler that can compile its own source code. Self-hosting software is commonplace on personal computers and larger systems. Other programs that are typically self-hosting include kernels, assemblers, command-line interpreters and revision control software.

This evaluation cycle will be embodied by the interplay between the two critical procedures in the evaluator, `eval` and `apply`, which are described in Section 4.1.1 (see Figure 4.1).

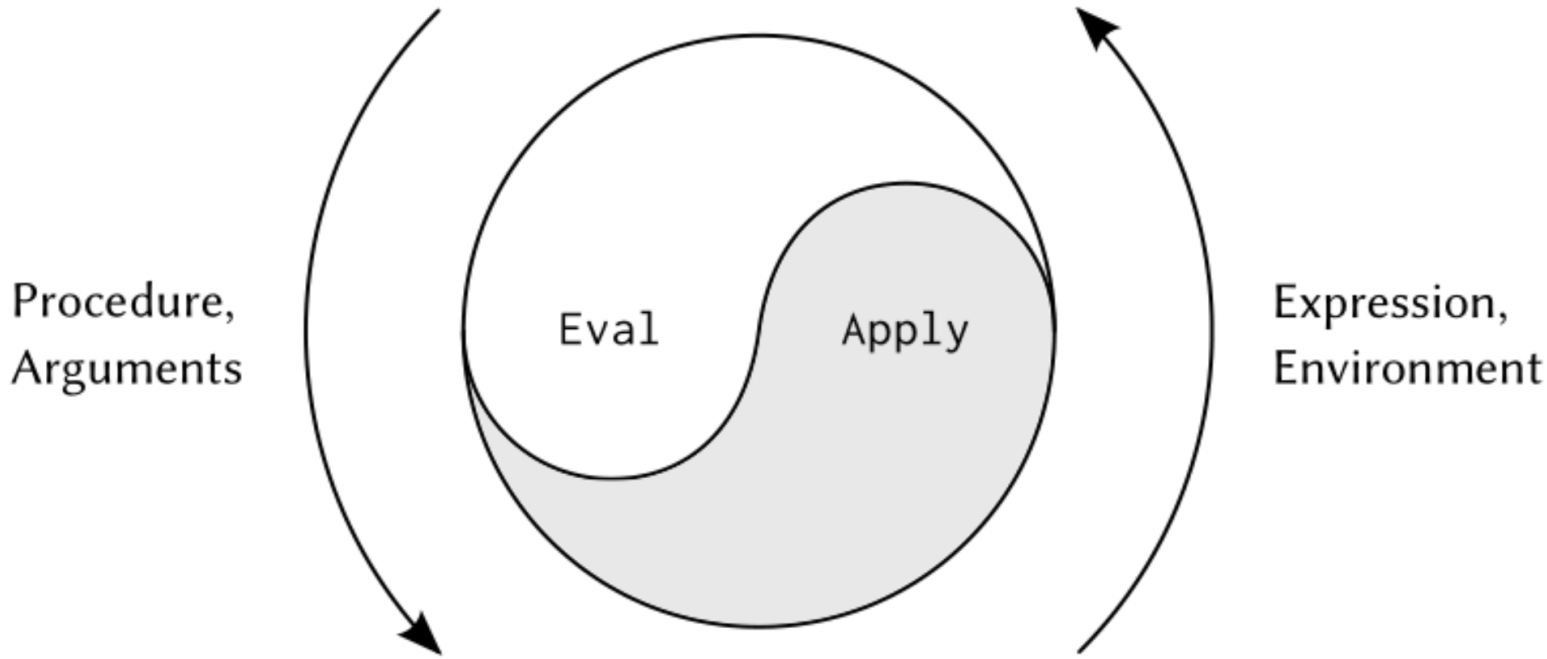


Figure 4.1: The `eval-apply` cycle exposes the essence of a computer language.

Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman


Eval
Appl

BEFORE WE LOOK AT THE CODE


```
(define apply-in-underlying-scheme apply) ; from footnote on page 520
```

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)                                ; #1 Literal
        ((variable?        exp) (lookup-variable-value exp env))    ; #2 Variable Reference
        ((quoted?          exp) (text-of-quotation exp))            ; #3 Special Form
        ((assignment?      exp) (eval-assignment exp env))          ; #3 Special Form
        ((definition?      exp) (eval-definition exp env))          ; #3 Special Form
        ((if?              exp) (eval-if exp env))                  ; #3 Special Form
        ((lambda?          exp) (make-procedure                     ; #3 Special Form
                                 (lambda-parameters exp)
                                 (lambda-body exp)
                                 env))
        ((begin?           exp) (eval-sequence                      ; #3 Special Form
                                 (begin-actions exp) env))
        ((cond?            exp) (eval (cond->if exp) env))          ; #3 Special Form
        ((and?             exp) (eval-and exp env))                 ; #3 Special Form
        ((or?              exp) (eval-or exp env))                  ; #3 Special Form
        ((let?             exp) (eval (let->combination exp) env))  ; #3 Special Form
        ((application?     exp) (my-apply (eval (operator exp) env) ; #4 Procedure Call
                                          (list-of-values
                                           (operands exp) env)))
        (else
         (error "Unknown expression type: FAIL" exp))))
```

```
(define (eval-working exp env)
  (cond ((self-evaluating? exp) exp)                                ; #1 Literal
        ;((variable?        exp) (lookup-variable-value exp env)) ; #2 Variable Reference
        ((quoted?          exp) (text-of-quotation exp))           ; #1 Literal ??
        ;((assignment?      exp) (eval-assignment exp env))       ; #3 Special Form
        ;((definition?      exp) (eval-definition exp env))       ; #3 Special Form
        ;((if?              exp) (eval-if exp env))               ; #3 Special Form
        ;((lambda?          exp) (make-procedure                  ; #3 Special Form
        ;                         (lambda-parameters exp)
        ;                         (lambda-body exp)
        ;                         env))
        ((begin?           exp) (eval-sequence                   ; #3 Special Form
                                  (begin-actions exp) env))
        ((cond?            exp) (eval (cond->if exp) env))       ; #3 Special Form
        ((application?     exp) (apply (eval (operator exp) env)  ; #4 Procedure Call
                                       (list-of-values
                                        (operands exp) env)))
        (else
         (error "Unknown expression type: FAIL" exp))))
```

```
(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type: APPLY" procedure))))
```

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

The use of `true?` in `eval-if` highlights the issue of the connection between an implemented language and an implementation language. The `if-predicate` is evaluated in the language being implemented and thus yields a value in that language. The interpreter predicate `true?` translates that value into a value that can be tested by the `if` in the implementation language: The metacircular representation of truth might not be the same as that of the underlying Scheme.[6]

[6]In this case, the language being implemented and the implementation language are the same. Contemplation of the meaning of `true?` here yields expansion of consciousness without the abuse of substance.

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)
         (eval (first-exp exps) env))
        (else
         (eval (first-exp exps) env)
         (eval-sequence (rest-exps exps) env))))
```

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  'ok)
```

Exercise 4.1: Notice that we cannot tell whether the metacircular evaluator evaluates operands from left to right or from right to left. Its evaluation order is inherited from the underlying Lisp: If the arguments to `cons` in `list-of-values` are evaluated from left to right, then `list-of-values` will evaluate operands from left to right; and if the arguments to `cons` are evaluated from right to left, then `list-of-values` will evaluate operands from right to left.

Write a version of `list-of-values` that evaluates operands from left to right regardless of the order of evaluation in the underlying Lisp. Also write a version of `list-of-values` that evaluates operands from right to left.

```
;; Exercise 4.1 (page 500-501)

(define (list-of-values-left-to-right exps env)
  (if (no-operands? exps)
      '()
      (let ((fst (eval (first-operand exps) env)))
        (cons fst
              (list-of-values (rest-operands exps) env)))))
```

```
(define (self-evaluating? exp)
  (cond ((number?  exp) true)
        ((string?  exp) true)
        ;((boolean? exp) true)
        (else false)))

(define (variable? exp) (symbol? exp))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

```
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)      ; formal parameters
                   (cddr exp))))    ; body
```

```scheme
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

```
(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```

```
(define (application? exp) (pair? exp))
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

```
(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp) (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last: COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

Exercise 4.2: Louis Reasoner plans to reorder the `cond` clauses in `eval` so that the clause for procedure applications appears before the clause for assignments. He argues that this will make the interpreter more efficient: Since programs usually contain more applications than assignments, definitions, and so on, his modified `eval` will usually check fewer clauses than the original `eval` before identifying the type of an expression.

a. What is wrong with Louis's plan? (Hint: What will Louis's evaluator do with the expression `(define x 3)`?)

```
;; Excercise 4.2 (page 507-8)

;; a) It will try to apply the operator "define" to the operands "x 3" and fail
;;    because "define" is a special form not a procedure
```

This is where we should cover
4.4, 4.6 & 4.9

```
(define (eval-working exp env)
  (cond ((self-evaluating? exp) exp)                                ; #1 Literal
        ;((variable?        exp) (lookup-variable-value exp env)) ; #2 Variable Reference
        ((quoted?          exp) (text-of-quotation exp))          ; #1 Literal ??
        ;((assignment?      exp) (eval-assignment exp env))       ; #3 Special Form
        ;((definition?      exp) (eval-definition exp env))       ; #3 Special Form
        ;((if?              exp) (eval-if exp env))               ; #3 Special Form
        ;((lambda?          exp) (make-procedure                  ; #3 Special Form
        ;                         (lambda-parameters exp)
        ;                         (lambda-body exp)
        ;                         env))
        ((begin?           exp) (eval-sequence                   ; #3 Special Form
                                  (begin-actions exp) env))
        ((cond?            exp) (eval (cond->if exp) env))       ; #3 Special Form
        ((application?     exp) (apply (eval (operator exp) env)  ; #4 Procedure Call
                                       (list-of-values
                                        (operands exp) env)))
        (else
         (error "Unknown expression type: FAIL" exp))))
```

The global environment also includes bindings for the symbols `true` and `false`, so that they can be used as variables in expressions to be evaluated.

```
(define (true? x) (not (eq? x false))) ; enables: eval-if; if? clause in eval
(define (false? x) (eq? x false))
```

```
(define (make-procedure parameters body env) ; enables: lambda? clause in eval
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))  ; at this point need to switch to #sicp
  (set-cdr! frame (cons val (cdr frame))))
```

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

```
(define (lookup-variable-value var env) ; enables: variable? clause of eval
  (define (env-loop env)
    (define (scan vars vals) ; :(
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

```
(define (set-variable-value! var val env) ; enables: eval-assignment;
  (define (env-loop env)                  ;          assignment? clause of eval
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable: SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

```
(define (define-variable! var val env) ; enables: eval-definition
  (let ((frame (first-frame env)))     ;          definition? clause of eval
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame) (frame-values frame))))
```

```
(define (primitive-procedure? proc) ; helps enable apply
  (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        ;;⟨ more primitives ⟩
        ))
```

```
(define (primitive-procedure-names)
  (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

Representing procedures

To handle primitives, we assume that we have available the following procedures:

- `(apply-primitive-procedure` ⟨*proc*⟩ ⟨*args*⟩`)`

  applies the given primitive procedure to the argument values in the list ⟨*args*⟩ and returns the result of the application.

- `(primitive-procedure?` ⟨*proc*⟩`)`

  tests whether ⟨*proc*⟩ is a primitive procedure.

These mechanisms for handling primitives are further described in Section 4.1.4.

```
(define input-prompt  ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

```
;(driver-loop) <- to kick things off
```

Exercise 4.4: Recall the definitions of the special forms `and` and `or` from Chapter 1:

- `and`: The expressions are evaluated from left to right. If any expression evaluates to false, false is returned; any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then true is returned.
- `or`: The expressions are evaluated from left to right. If any expression evaluates to a true value, that value is returned; any remaining expressions are not evaluated. If all expressions evaluate to false, or if there are no expressions, then false is returned.

Install `and` and `or` as new special forms for the evaluator by defining appropriate syntax procedures and evaluation procedures `eval-and` and `eval-or`. Alternatively, show how to implement `and` and `or` as derived expressions.

```
(define (and? exp) (tagged-list? exp 'and))
(define (or? exp)  (tagged-list? exp 'or))
```

```scheme
(define (eval-or tag-and-exps env)
  (define (iter exps)
    (cond ((null? exps) false)
          ((true? (eval (car exps) env)) true)
          (else (iter (cdr exps)))))
  (iter (cdr tag-and-exps)))

;; Tests
(eval '(or true true) the-global-environment)   ; #t
(eval '(or false true) the-global-environment)  ; #t
(eval '(or false false) the-global-environment) ; #f
```

```scheme
(define (eval-and tag-and-exps env)
  (define (iter exps)
    (cond ((null? exps) true)
          ((false? (eval (car exps) env)) false)
          (else (iter (cdr exps)))))
  (iter (cdr tag-and-exps)))

;; Tests
(eval '(and false true) the-global-environment) ; #f
(eval '(and true true) the-global-environment)  ; #t
```

Exercise 4.6: `let` expressions are derived expressions, because

```
(let ((⟨var1⟩ ⟨exp1⟩) ... (⟨varn⟩ ⟨expn⟩))
  ⟨body⟩)
```

is equivalent to

```
((lambda (⟨var1⟩ ... ⟨varn⟩)
   ⟨body⟩)
 ⟨exp1⟩
 ...
 ⟨expn⟩)
```

Implement a syntactic transformation `let->combination` that reduces evaluating `let` expressions to evaluating combinations of the type shown above, and add the appropriate clause to `eval` to handle `let` expressions.

```
;; Exercise 4.6 (page 509-10)

;  (let ((a b)
;        (c d)) ...) ; ->

;  (lambda (a c) (...) (b d)))

(define (let? exp) (tagged-list? exp 'let))

(define (let->combination exp)
  (let* ((bindings       (cadr exp))
         (binding-names  (map car bindings))
         (binding-values (map cadr bindings))
         (body     (cddr exp)))
    (cons (make-lambda binding-names
                       body)
          binding-values)))
```

```
;; Test
(eval '(define (add x y)
         (let ((a x) (b y)) (+ a b))) the-global-environment) ; ok
(eval '(add 1 2) the-global-environment) ; 3
```

Exercise 4.9: Many languages support a variety of iteration constructs, such as `do`, `for`, `while`, and `until`. In Scheme, iterative processes can be expressed in terms of ordinary procedure calls, so special iteration constructs provide no essential gain in computational power. On the other hand, such constructs are often convenient. Design some iteration constructs, give examples of their use, and show how to implement them as derived expressions.

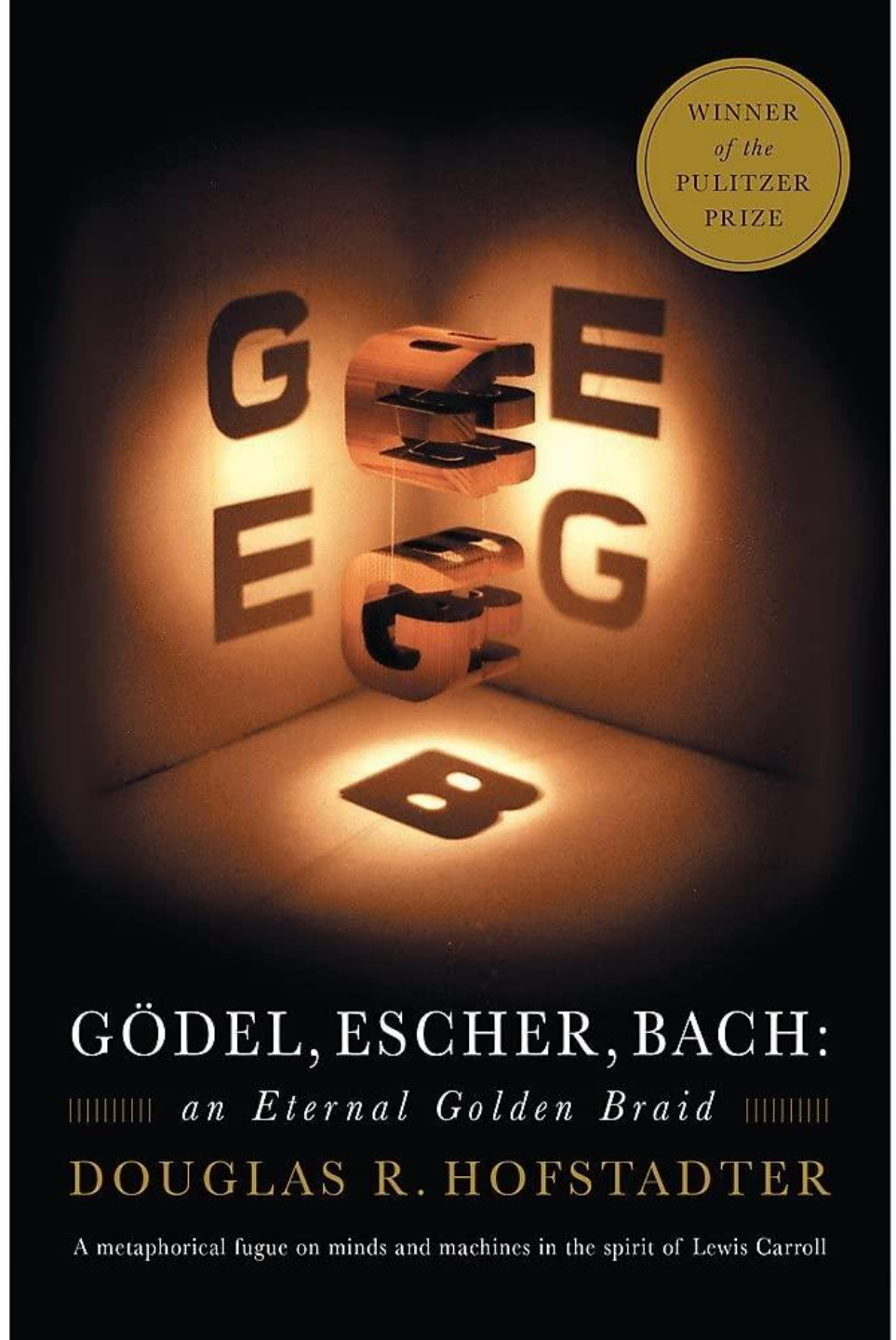
```
;; test
(eval '(define (squares-less-than val)
         (let ((i 0))
           (while (< (* i i) val)
                  (println (* i i))
                  (set! i (+ i 1))))) the-global-environment)

(eval '(squares-less-than 1000) the-global-environment)
; 0
; 1
; 4
; 9
; 16
; 25
; 36
; 49
; 64
; 81 ...
```

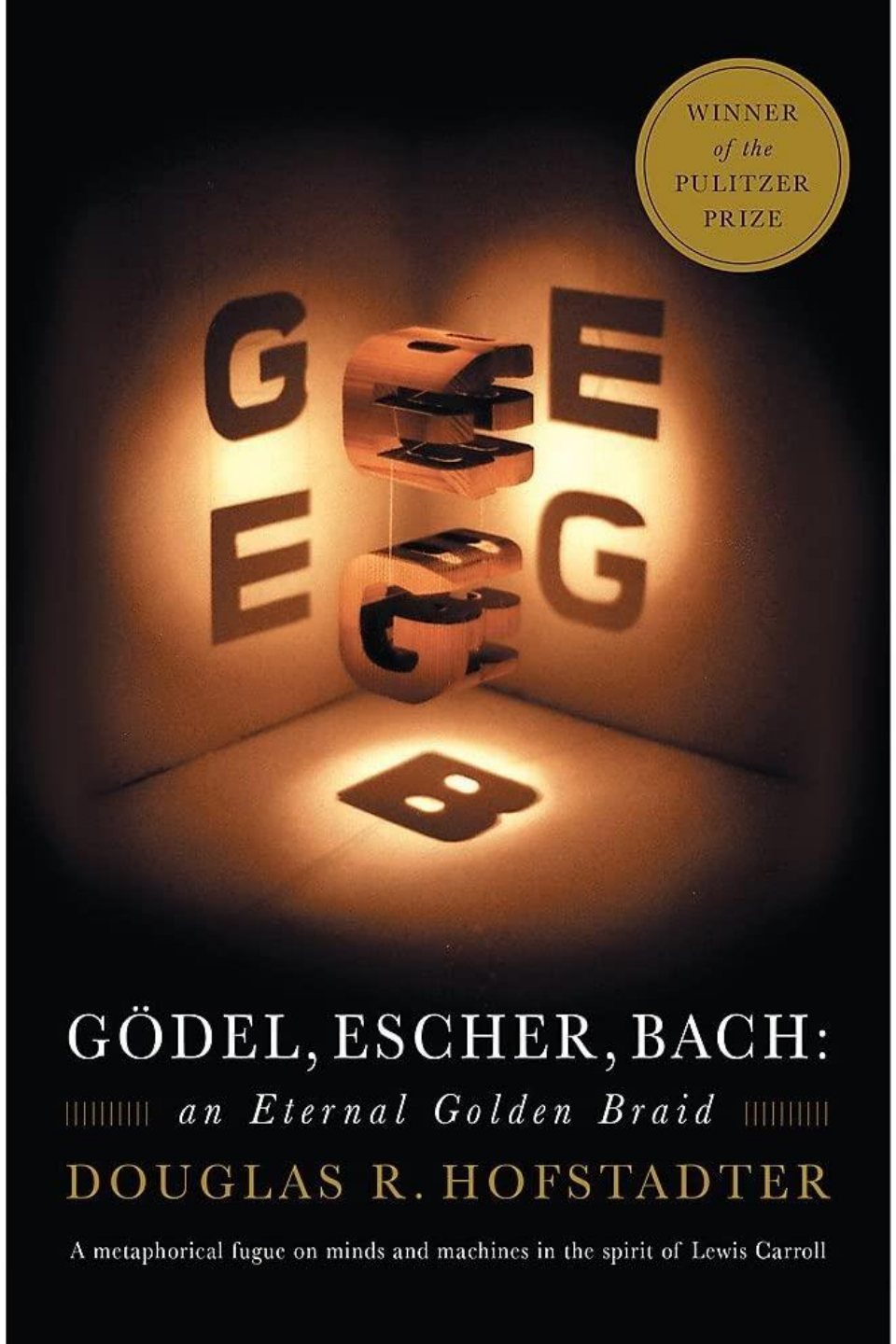
```
;; Exercise 4.9 (page 511)

;; implementing while
; (while <predicate> <body>)

(define (while? exp) (tagged-list? exp 'while))

(define (while->combination exp)
  (let ((predicate (cadr exp))
        (body      (cddr exp)))
    (sequence->exp
     (list (list 'define
                 (list 'while-iter)
                 (make-if predicate
                          (sequence->exp
                           (append body (list (list 'while-iter))))
                          'true))
           (list 'while-iter)))))
```

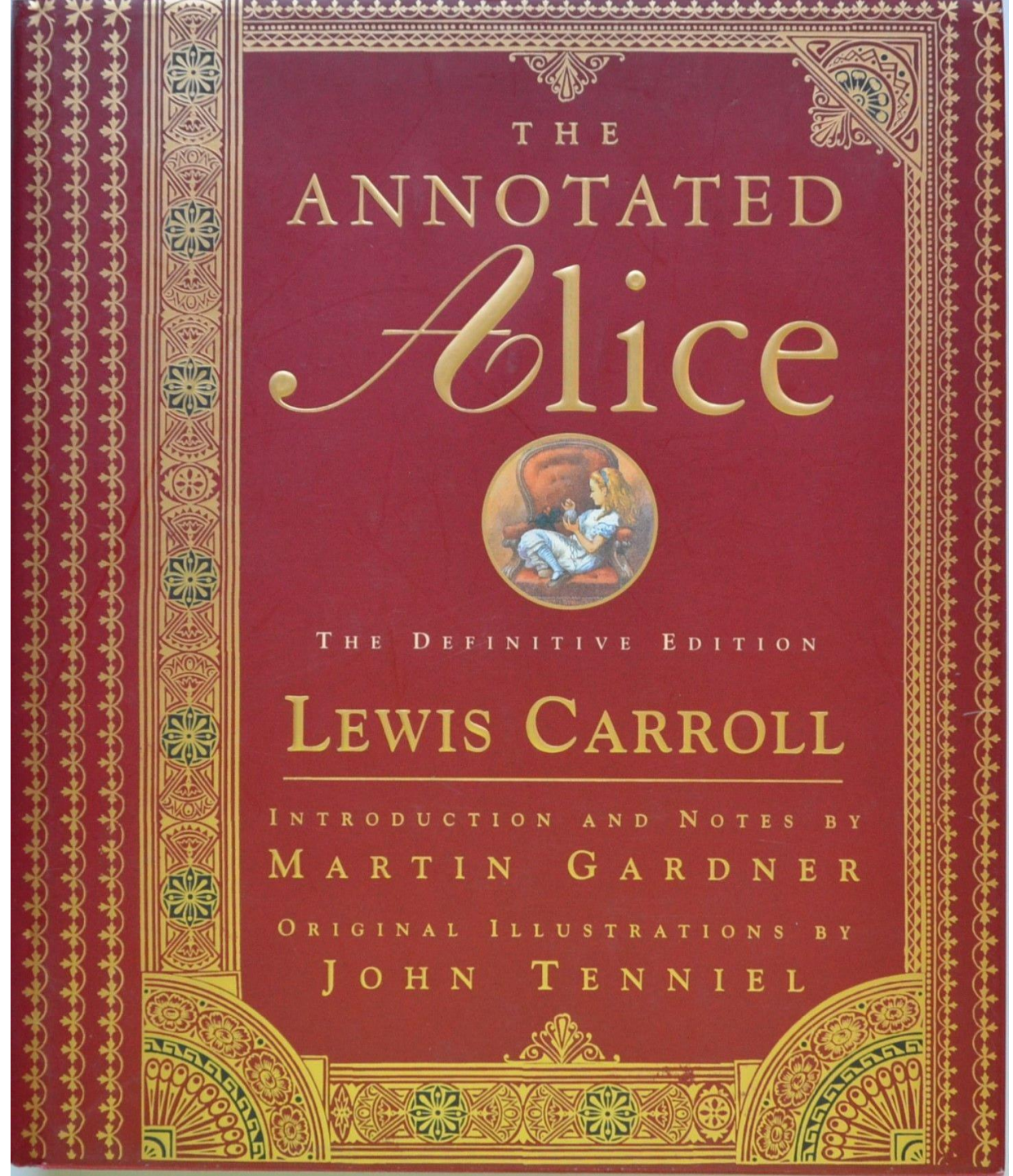
[19]The fact that the machines are described in Lisp is inessential. If we give our evaluator a Lisp program that behaves as an evaluator for some other language, say C, the Lisp evaluator will emulate the C evaluator, which in turn can emulate any machine described as a C program. Similarly, writing a Lisp evaluator in C produces a C program that can execute any Lisp program. The deep idea here is that any evaluator can emulate any other. Thus, the notion of "what can in principle be computed" (ignoring practicalities of time and memory required) is independent of the language or the computer, and instead reflects an underlying notion of *computability*. This was first demonstrated in a clear way by Alan M. Turing (1912-1954), whose 1936 paper laid the foundations for theoretical computer science. In the paper, Turing presented a simple computational model—now known as a *Turing machine*—and argued that any "effective process" can be formulated as a program for such a machine. (This argument is known

as the *Church-Turing thesis.*) Turing then implemented a universal machine, i.e., a Turing machine that behaves as an evaluator for Turing-machine programs. He used this framework to demonstrate that there are well-posed problems that cannot be computed by Turing machines (see Exercise 4.15), and so by implication cannot be formulated as "effective processes." Turing went on to make fundamental contributions to practical computer science as well. For example, he invented the idea of structuring programs using general-purpose subroutines. See Hodges 1983 for a biography of Turing.

[20]Some people find it counterintuitive that an evaluator, which is implemented by a relatively simple procedure, can emulate programs that are more complex than the evaluator itself. The existence of a universal evaluator machine is a deep and wonderful property of computation. *Recursion theory*, a branch of mathematical logic, is concerned with logical limits of computation. Douglas Hofstadter's beautiful book *Gödel, Escher, Bach* explores some of these ideas (Hofstadter 1979).


WINNER
of the
PULITZER
PRIZE
G E
E G
B
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll


WINNER
of the
PULITZER
PRIZE
G E
E G
B
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll


THE
ANNOTATED
Alice
THE DEFINITIVE EDITION
LEWIS CARROLL
INTRODUCTION AND NOTES BY
MARTIN GARDNER
ORIGINAL ILLUSTRATIONS BY
JOHN TENNIEL

Exercise 4.15: Given a one-argument procedure `p` and an object `a`, `p` is said to "halt" on `a` if evaluating the expression `(p a)` returns a value (as opposed to terminating with an error message or running forever). Show that it is impossible to write a procedure `halts?` that correctly determines whether `p` halts on `a` for any procedure `p` and object `a`. Use the following reasoning: If you had such a procedure `halts?`, you could implement the following program:

```
(define (run-forever) (run-forever))
(define (try p)
  (if (halts? p p) (run-forever) 'halted))
```

Now consider evaluating the expression `(try try)` and show that any possible outcome (either halting or running forever) violates the intended behavior of `halts?`.[23]

[23] Although we stipulated that `halts?` is given a procedure object, notice that this reasoning still applies even if `halts?` can gain access to the procedure's text and its environment. This is Turing's celebrated *Halting Theorem*, which gave the first clear example of a *non-computable* problem, i.e., a well-posed task that cannot be carried out as a computational procedure.

```
(define (f x)
  (define (even? n) (if (= n 0) true  (odd?  (- n 1))))
  (define (odd? n)  (if (= n 0) false (even? (- n 1))))
  ⟨rest of body of f⟩)
```

[27] This example illustrates a programming trick for formulating recursive procedures without using `define`. The most general trick of this sort is the *Y operator*, which can be used to give a "pure λ-calculus" implementation of recursion. (See Stoy 1977 for details on the λ-calculus, and Gabriel 1988 for an exposition of the *Y* operator in Scheme.)

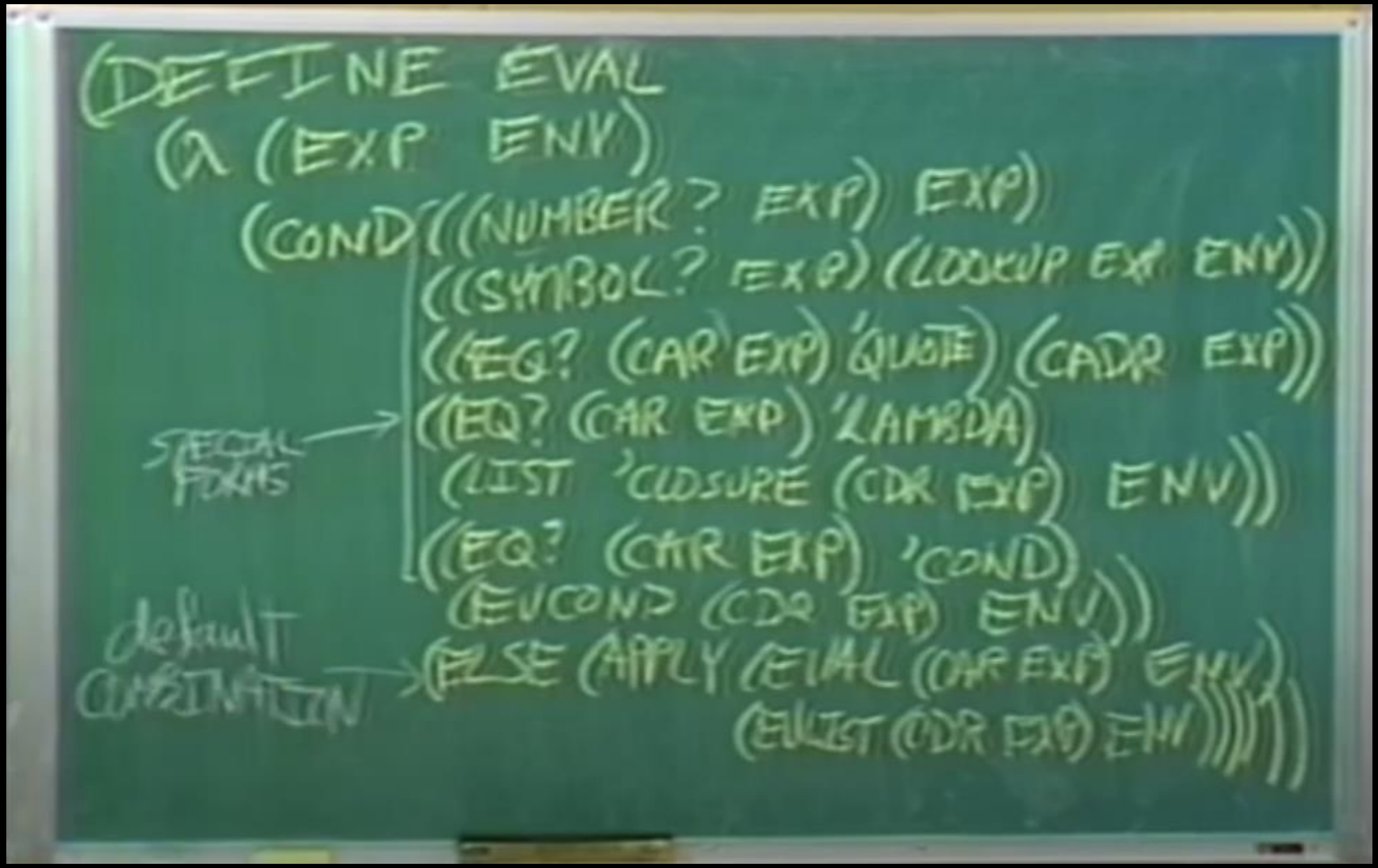
The evaluator implemented above is simple, but it is very inefficient, because the syntactic analysis of expressions is interleaved with their execution.

```
(cond ((self-evaluating? exp) (analyze-self-evaluating exp))
      ((quoted? exp) (analyze-quoted exp))
      ((variable? exp) (analyze-variable exp))
      ((assignment? exp) (analyze-assignment exp))
      ((definition? exp) (analyze-definition exp))
      ((if? exp) (analyze-if exp))
      ((lambda? exp) (analyze-lambda exp))
      ((begin? exp) (analyze-sequence (begin-actions exp)))
      ((cond? exp) (analyze (cond->if exp)))
      ((application? exp) (analyze-application exp))
      (else (error "Unknown expression type: ANALYZE" exp))))
```

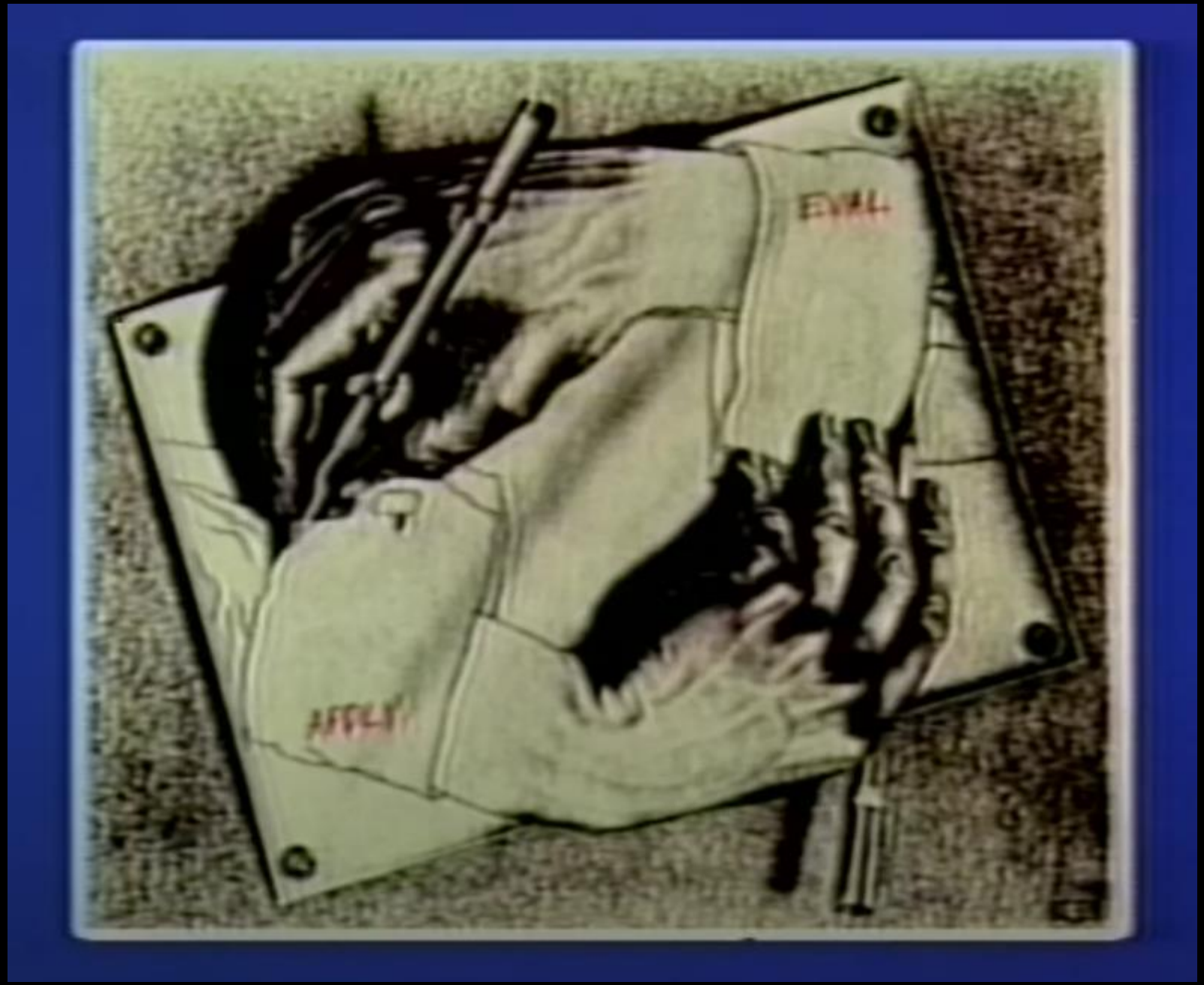
Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman


(DEFINE EVAL
(λ (EXP ENV)
(COND ((NUMBER? EXP) EXP)
((SYMBOL? EXP) (LOOKUP EXP ENV))
((EQ? (CAR EXP) 'QUOTE) (CADR EXP))
((EQ? (CAR EXP) 'LAMBDA)
(LIST 'CLOSURE (CDR EXP) ENV))
((EQ? (CAR EXP) 'COND)
(EVCOND (CDR EXP) ENV)))
(ELSE (APPLY (EVAL (CAR EXP) ENV)
(EVLIST (CDR EXP) ENV)))))
SPECIAL FORMS
default COMBINATION

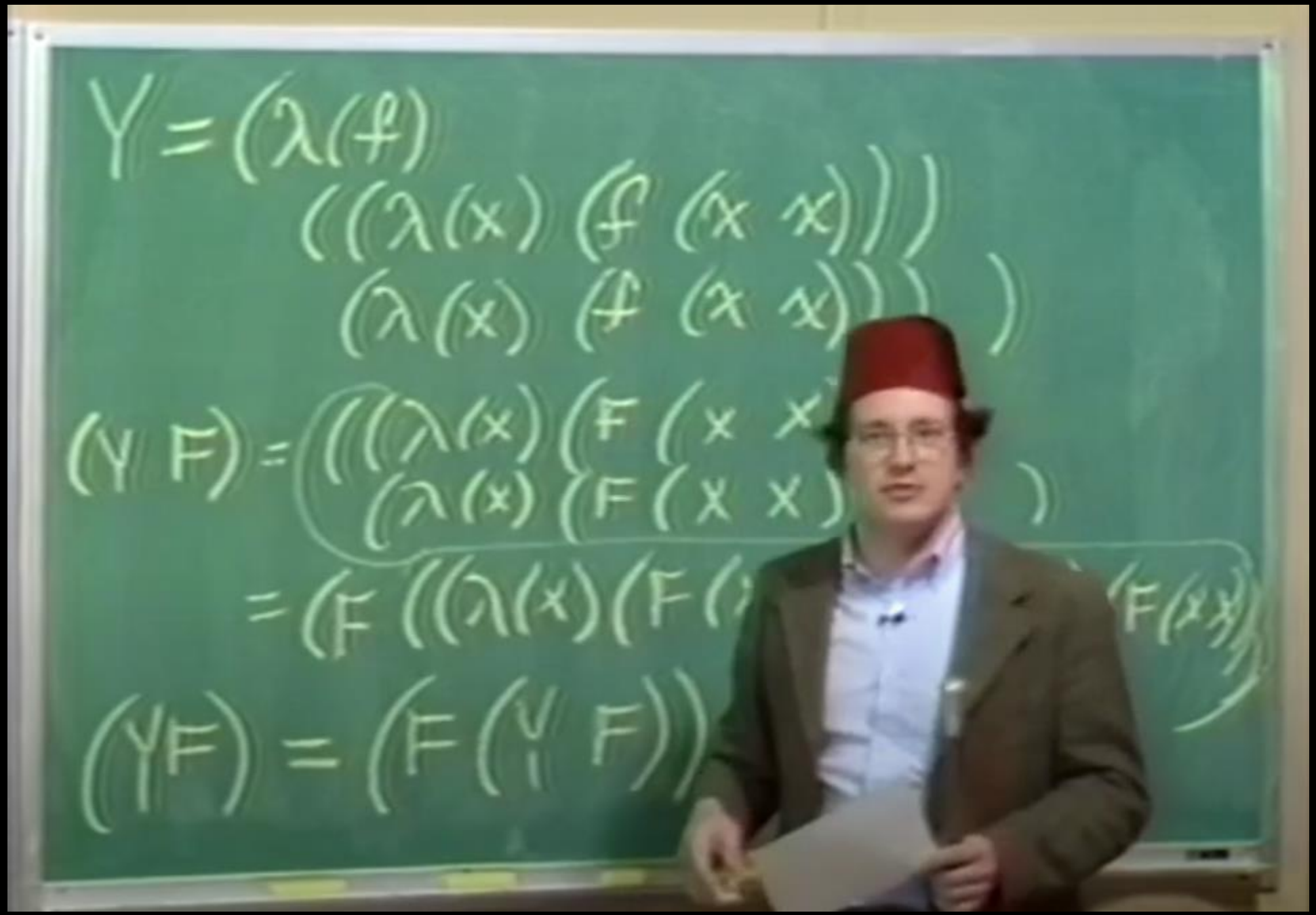

EQ?
VAL


EVIL
AFGLA!

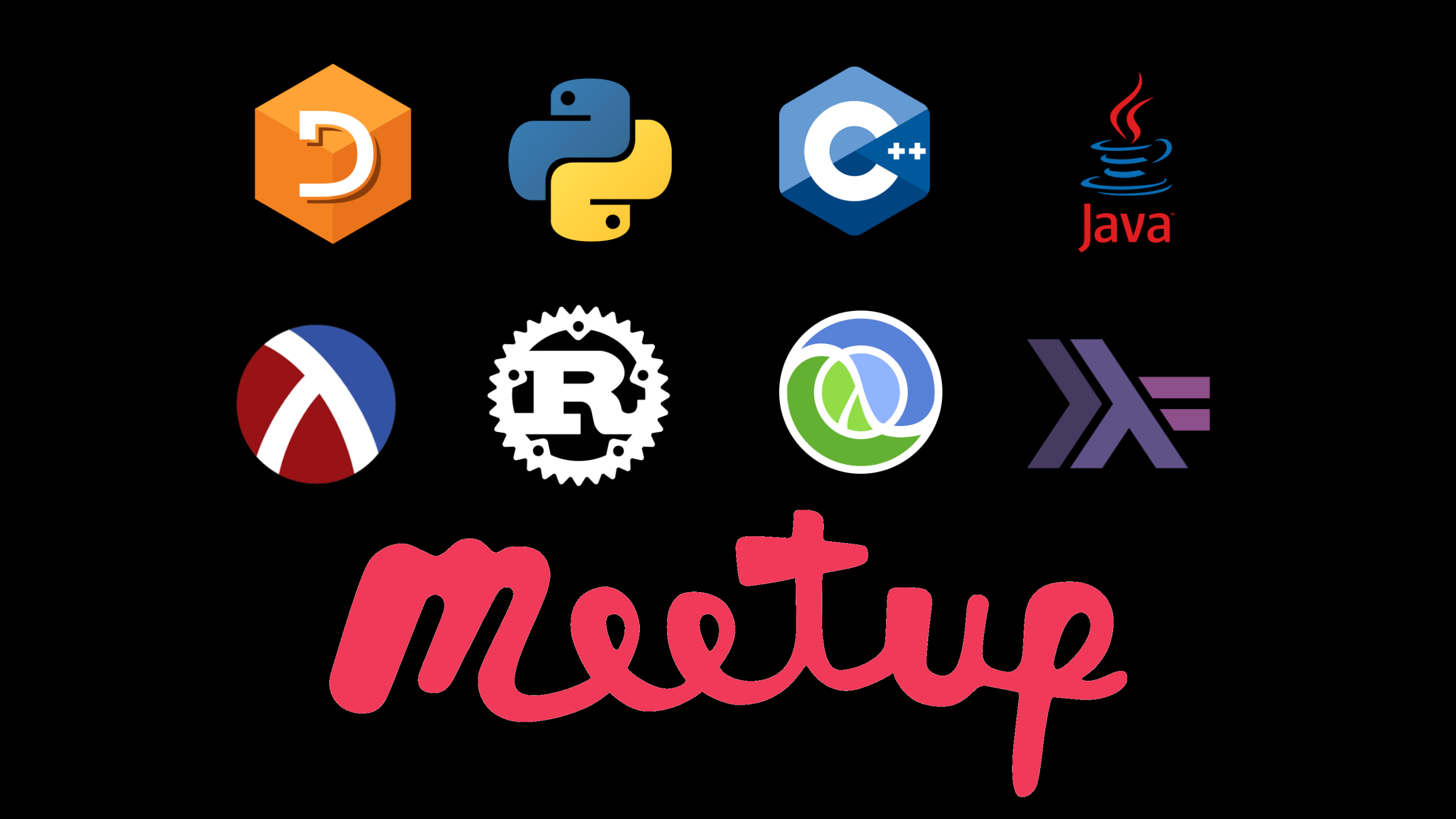
"I'll call it **Y**. This is called Curry's Paradoxical Combinator Y after a fellow by the name of Curry who was a logician of the 1930s."

Gerald Sussman

Lecture 7A, SICP


Y = (λ(f)
((λ(x) (f (x x)))
(λ(x) (f (x x)))))
(Y F) = (((λ(x) (F (x x)))
(λ(x) (F (x x)))))
= (F ((λ(x) (F (
F(x x)))
(Y F) = (F (Y F))


MASSACHUSETTS
GRAND
WIZARD
(YF) = (F(YF))
λ
GJS
INSTITUTE OF TECHNOLOGY


Java
meetup