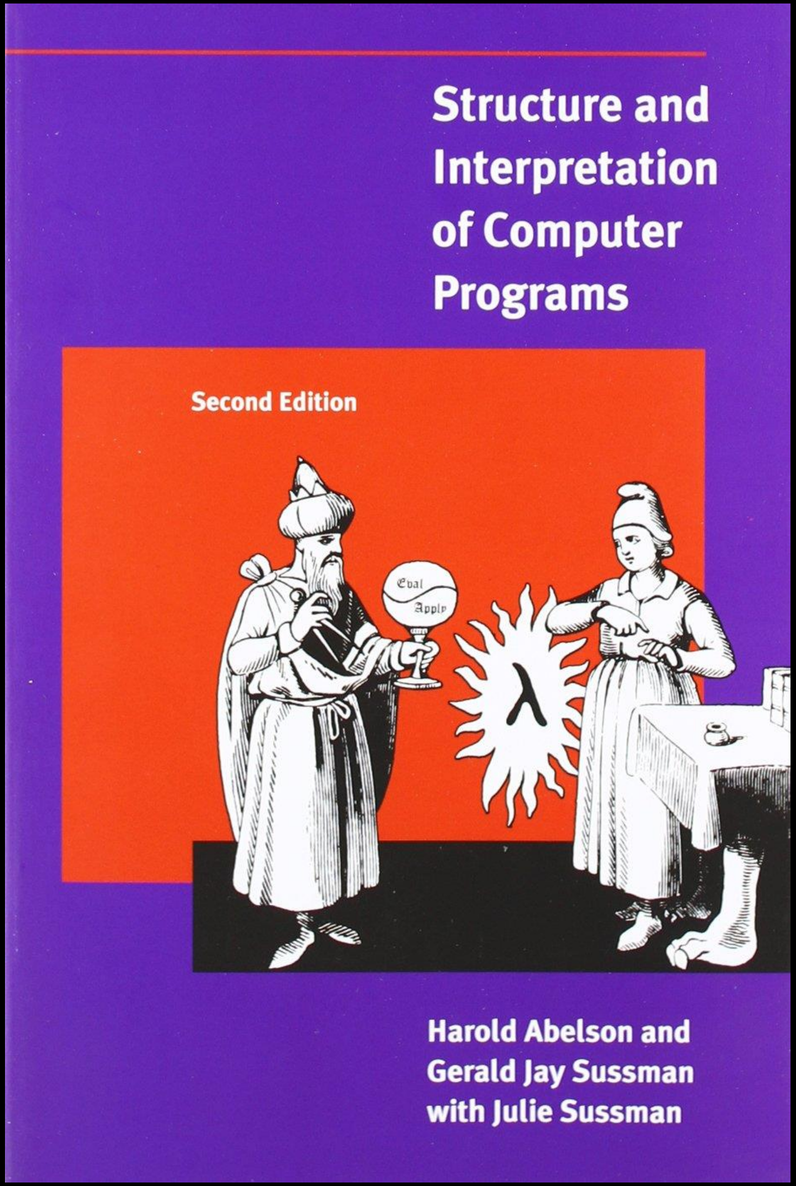
Java

Meetup


Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and Gerald Jay Sussman with Julie Sussman

Structure and Interpretation of Computer Programs

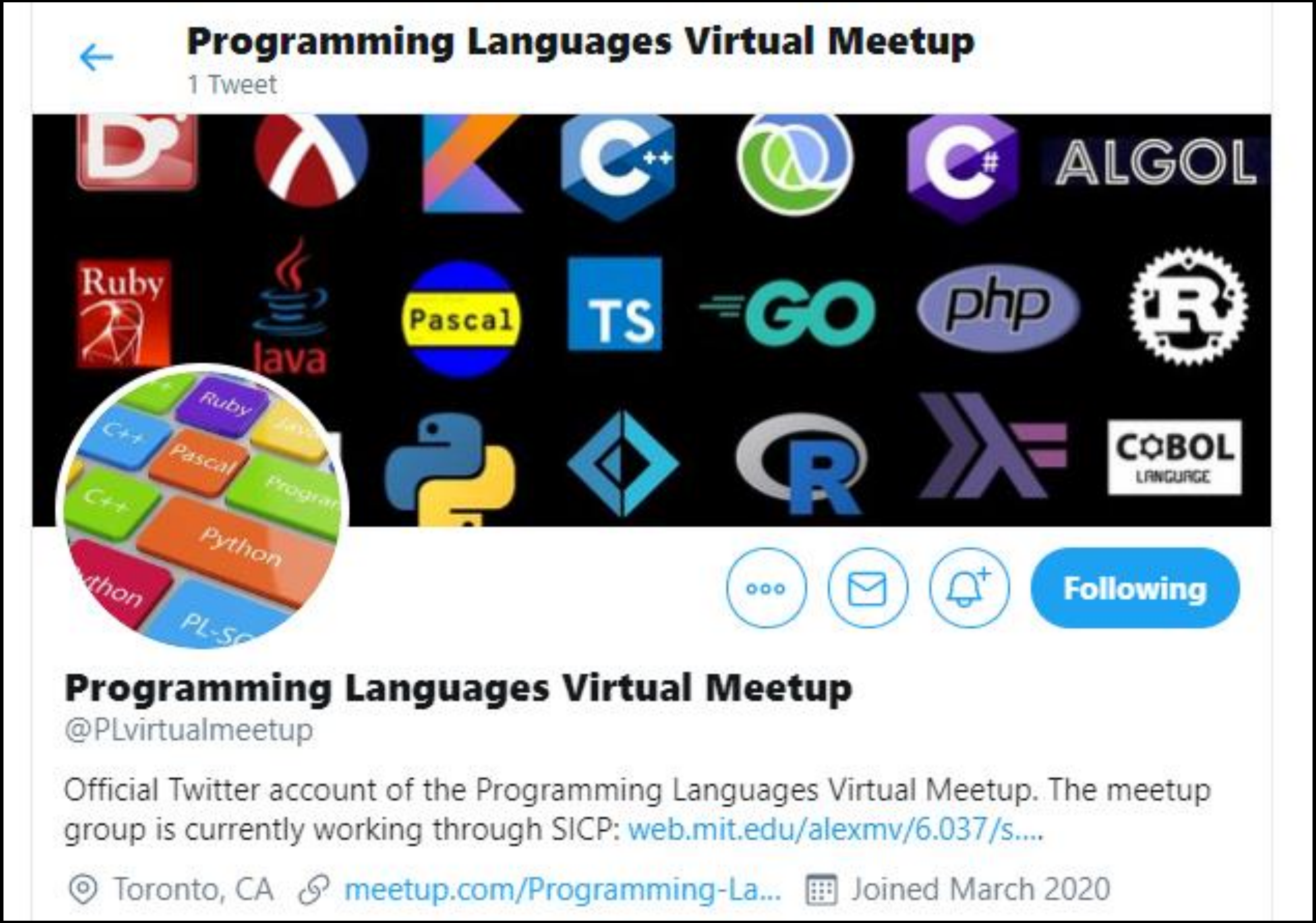
Chaper 4.2

Before we start …

Friendly Environment Policy

Berlin Code of Conduct

DISCORD


Programming Languages Virtual Meetup

1 Tweet

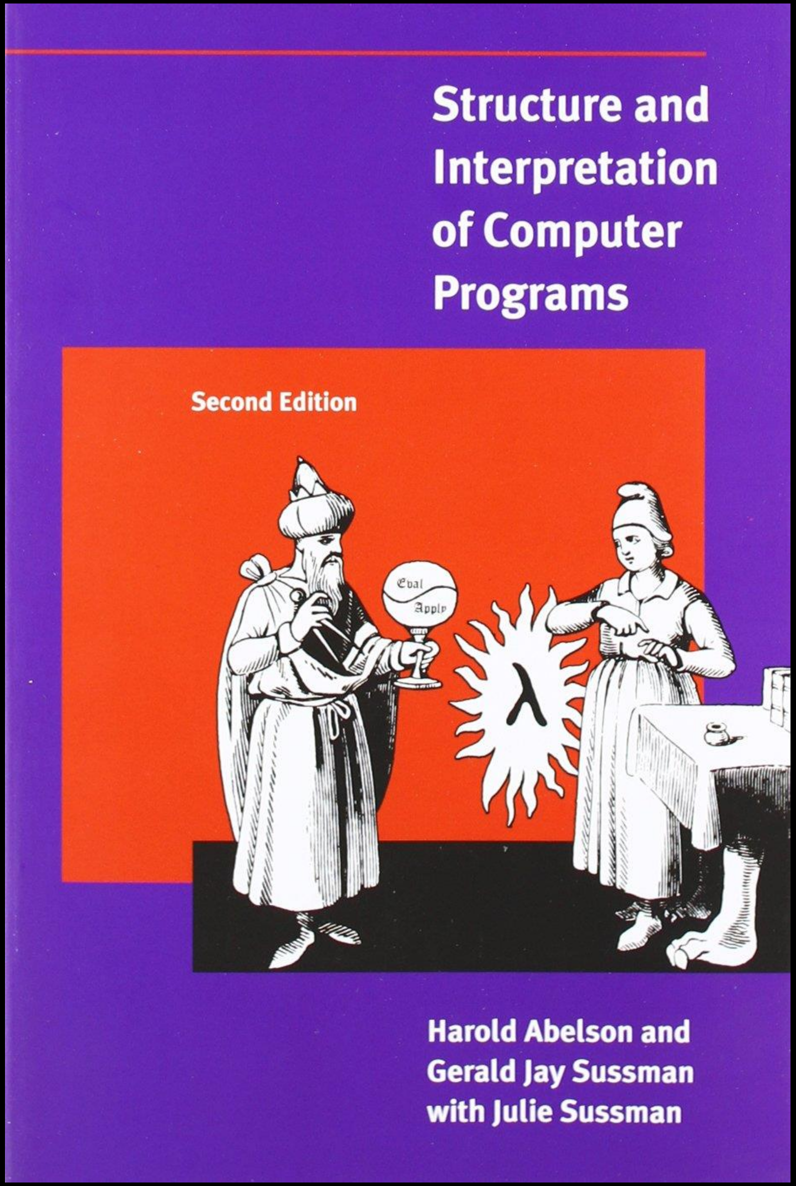
Following

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
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

Structure and Interpretation of Computer Programs

Chaper 4.2

4.2.1 Normal Order and Applicative Order

In Section 1.1, where we began our discussion of models of evaluation, we noted that Scheme is an *applicative-order* language, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied. In contrast, *normal-order* languages delay evaluation of procedure arguments until the actual argument values are needed. Delaying evaluation of procedure arguments until the last possible moment (e.g., until they are required by a primitive operation) is called *lazy evaluation.*[32]

An example that exploits lazy evaluation is the definition of a procedure `unless`

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))
```

that can be used in expressions such as

```
(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0") 0))
```

[32]The difference between the “lazy” terminology and the “normal-order” terminology is somewhat fuzzy. Generally, “lazy” refers to the mechanisms of particular evaluators, while “normal-order” refers to the semantics of languages, independent of any particular evaluation strategy. But this is not a hard-and-fast distinction, and the two terminologies are often used interchangeably.

If the body of a procedure is entered before an argument has been evaluated we say that the procedure is *non-strict* in that argument. If the argument is evaluated before the body of the procedure is entered we say that the procedure is *strict* in that argument.[33]

[33]The “strict” versus “non-strict” terminology means essentially the same thing as “applicative-order” versus “normal-order,” except that it refers to individual procedures and arguments rather than to the language as a whole. At a conference on programming languages you might hear someone say, “The normal-order language Hassle has certain strict primitives. Other procedures take their arguments by lazy evaluation.”

Applicative order	Normal order
?	Lazy
Strict	Non-strict

Applicative order	Normal order
Eager	Lazy
Strict	Non-strict

Exercise 4.25: Suppose that (in ordinary applicative-order Scheme) we define `unless` as shown above and then define

`factorial` in terms of `unless` as

```
(define (factorial n)
  (unless (= n 1)
          (* n (factorial (- n 1)))
          1))
```

What happens if we attempt to evaluate `(factorial 5)`? Will our definitions work in a normal-order language?

```
;; Exercise 4.25 (543-4)

;; Applicative order will run forever - because base case will never be hit "alone"
;;     - both arguments will always be evaluated.
;; Normal order will not have this problem and will work fine.
```

The delayed arguments are not evaluated; instead, they are transformed into objects called *thunks*.[34]

[34]The word *thunk* was invented by an informal working group that was discussing the implementation of call-by-name in Algol 60. They observed that most of the analysis of ("thinking about") the expression could be done at compile time; thus, at run time, the expression would already have been "thunk" about (Ingerman et al. 1960).

Modifying the evaluator

The main difference between the lazy evaluator and the one in Section 4.1 is in the handling of procedure applications in `eval` and `apply`.

The `application?` clause of `eval` becomes

```
((application? exp)
 (apply (actual-value (operator exp) env)
        (operands exp)
        env))
```

```
(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))          ; #1 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))))
```

```
((application?     exp) (my-apply (eval (operator exp) env) ; #4 Procedure Call
                                   (list-of-values
                                    (operands exp) env)))
```

```
((application?     exp) (my-apply (actual-value            ; #4 Procedure Call
                                     (operator exp) env)
                                    (operands exp) env))
```

```
;; actual-value - Add

(define (actual-value exp env)
  (force-it (eval exp env)))
```

```
;; apply - Before

(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))))
```

```scheme
;; apply - After

(define (my-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
          (list-of-arg-values arguments env)))  ; changed
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; changed
           (procedure-environment procedure))))
        (else (error "Unknown procedure type: APPLY"
                     procedure))))
```

```
;; list-of-values - Removed

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

```
;; list-of-* - Added

(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps)
                          env)
            (list-of-arg-values (rest-operands exps)
                                env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps)
                      env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

```
;; eval-if - Before

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

```
;; eval-if - After

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

```
;; driver-loop - Before

(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 - After

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

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

```
;; additions (from Representing thunks)

;(define (force-it obj)
;  (if (thunk? obj)
;      (actual-value (thunk-exp obj) (thunk-env obj))
;      obj))

(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj)       (tagged-list? obj 'thunk))
(define (thunk-exp thunk)  (cadr thunk))
(define (thunk-env thunk)  (caddr thunk))

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))
```

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result) ; replace exp with its value
           (set-cdr! (cdr obj) '())    ; forget unneeded env
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

Exercise 4.27: Suppose we type in the following definitions to the lazy evaluator:

```
(define count 0)
(define (id x) (set! count (+ count 1)) x)
```

Give the missing values in the following sequence of interactions, and explain your answers.[38]

```
(define w (id (id 10)))
;;; L-Eval input:
count
;;; L-Eval value:
⟨response⟩
;;; L-Eval input:
w
;;; L-Eval value:
⟨response⟩
;;; L-Eval input:
count
;;; L-Eval value:
⟨response⟩
```

```
;; Exercise 4.27 (page 551)

;;; L-Eval input:
(define count 0)
;;; L-Eval value:
ok

;;; L-Eval input:
(define (id x) (set! count (+ count 1)) x)
;;; L-Eval value:
ok

;;; L-Eval input:
(define w (id (id 10)))
;;; L-Eval value:
ok

;;; L-Eval input:
count
;;; L-Eval value:
1

;;; L-Eval input:
w
;;; L-Eval value:
10

;;; L-Eval input:
count
;;; L-Eval value:
2
```

```
;; w becomes a thunk and only one of the calls to id occurs, setting it to 1.
;; Only after evaluating w does it set the counter to 2.
```

Exercise 4.31: The approach taken in this section is somewhat unpleasant, because it makes an incompatible change to Scheme. It might be nicer to implement lazy evaluation as an *upward-compatible extension*, that is, so that ordinary Scheme programs will work as before. We can do this by extending the syntax of procedure declarations to let the user control whether or not arguments are to be delayed. While we're at it, we may as well also give the user the choice between delaying with and without memoization. For example, the definition

```
(define (f a (b lazy) c (d lazy-memo))
  ...)
```

would define `f` to be a procedure of four arguments, where the first and third arguments are evaluated when the procedure is called, the second argument is delayed, and the fourth argument is both delayed and memoized. Thus, ordinary procedure definitions will produce the same behavior as ordinary Scheme, while adding the `lazy-memo` declaration to each parameter of every compound procedure will produce the behavior of the lazy evaluator defined in this section. Design and implement the changes required to produce such an extension to Scheme. You will have to implement new syntax procedures to handle the new syntax for `define`. You must also arrange for `eval` or `apply` to determine when arguments are to be delayed, and to force or delay arguments accordingly, and you must arrange for forcing to memoize or not, as appropriate.

With lazy evaluation, streams and lists can be identical, so there is no need for special forms or for separate list and stream operations. All we need to do is to arrange matters so that `cons` is non-strict. One way to accomplish this is to extend the lazy evaluator to allow for non-strict primitives, and to implement `cons` as one of these. An easier way is to recall (Section 2.1.3) that there is no fundamental need to implement `cons` as a primitive at all. Instead, we can represent pairs as procedures:[40]

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```

Exercise 4.33: Ben Bitdiddle tests the lazy list implementation given above by evaluating the expression:

```
(car '(a b c))
```

To his surprise, this produces an error. After some thought, he realizes that the "lists" obtained by reading in quoted expressions are different from the lists manipulated by the new definitions of `cons`, `car`, and `cdr`. Modify the evaluator's treatment of quoted expressions so that quoted lists typed at the driver loop will produce true lazy lists.

```
;; Exercise 4.33 (page 558)

(eval '(define (cons x y) (lambda (m) (m x y))) the-global-environment)
(eval '(define (car z) (z (lambda (p q) p))) the-global-environment)
(eval '(define (cdr z) (z (lambda (p q) q))) the-global-environment)

(eval '(car '(1 2 3)) the-global-environment)
;; . . Unknown procedure type: APPLY (1 2 3)

;; text-of-quotation - Before
(define (text-of-quotation exp) (cadr exp))
```

```
; text-of-quotation - After
(define (text-of-quotation exp)
  (define (quotation->cons exp)
    (if (pair? exp)
        (list 'cons
              (quotation->cons (car exp))
              (quotation->cons (cdr exp)))
        exp))
  (let ((text (cadr exp)))
    (if (pair? text)
        (eval (quotation->cons text) the-global-environment)
        text)))
```

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

Java

Meetup