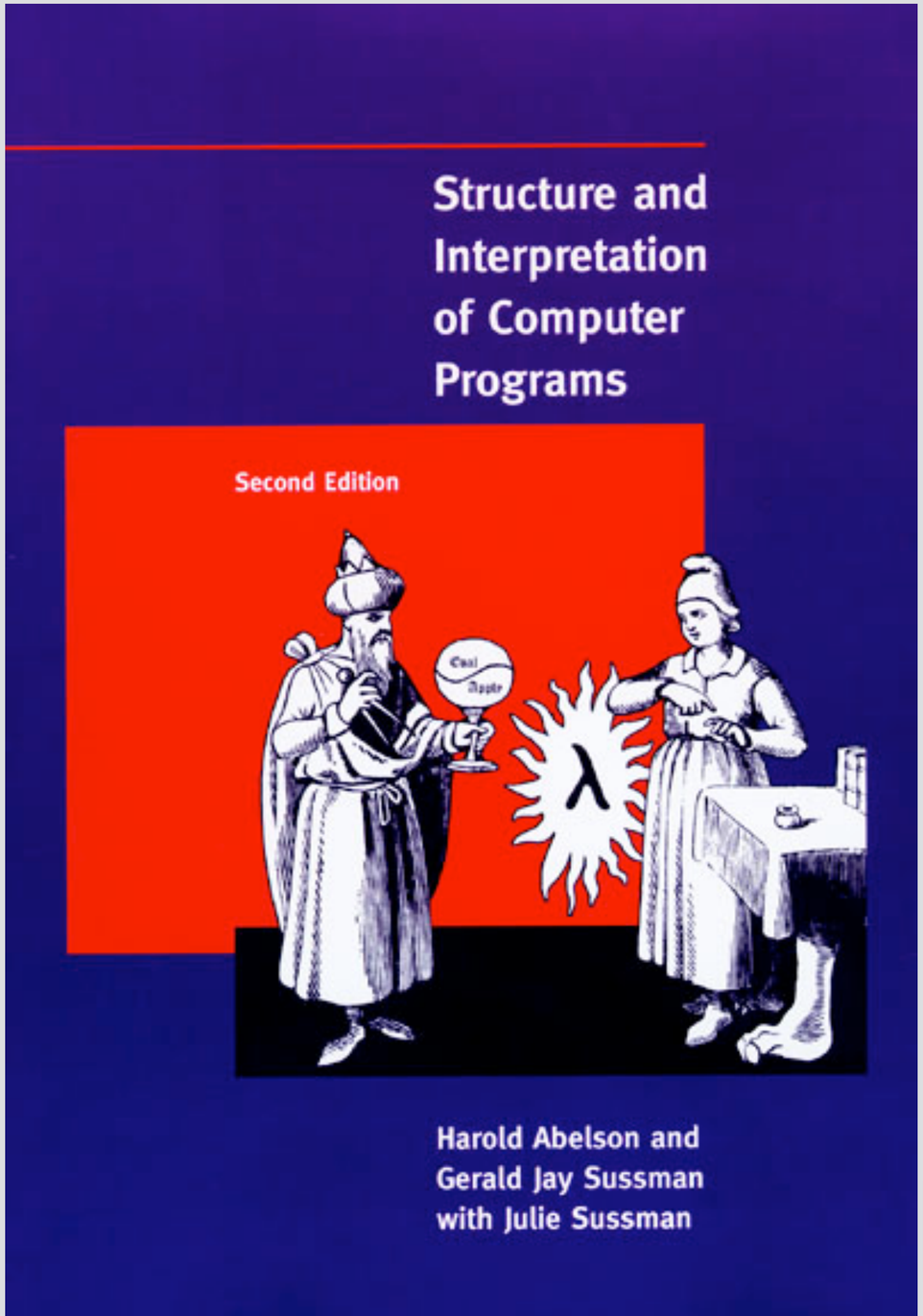
Chapter 5

Unfolding the Mystery: Scheme Evaluators (part II)

Every computer scientist should go meta at least once in their life (Dave Thomas)

What if Scheme is not enough?

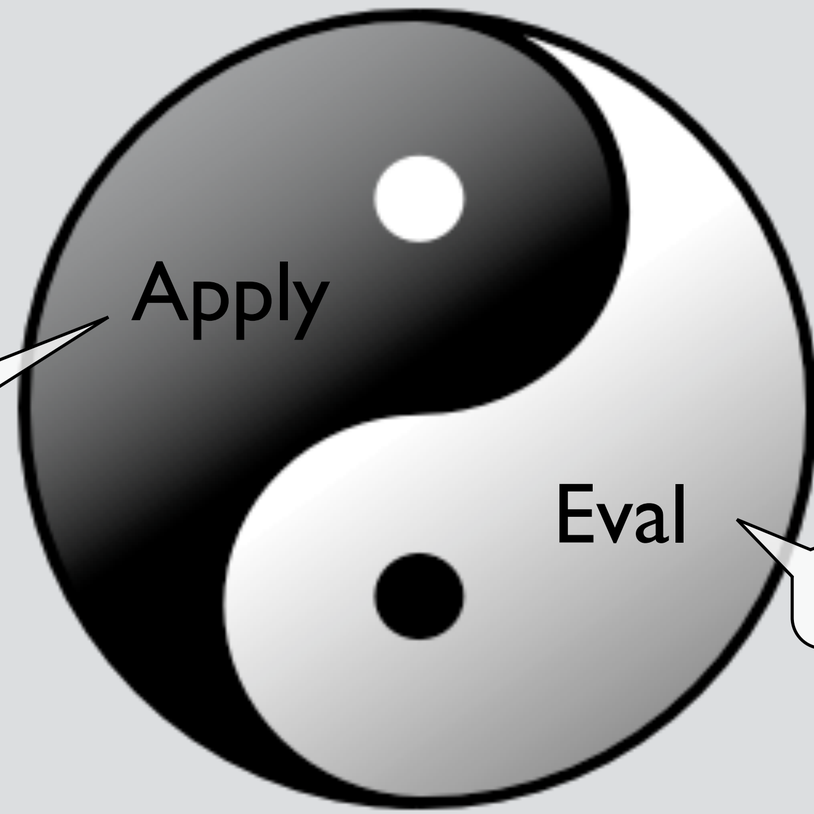


Metalinguistic abstraction -- establishing new languages -- plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.

The interpreter is just another program

The Interpreter is just another program

written in Scheme!



procedure + arguments

```
(define (apply proc args)
  ...)
```

expression + environment

```
(define (eval exp env)
  ...)
```

Keep the environment model in mind

The Scheme Evaluator: eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

The Scheme Evaluator: apply

```
(define (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))))
```

Drawback of MC-eval

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

Deciding that "the body is an if-expression" is done n times

Separate the analysis of the syntax from the actual execution

Separate Syntax Analysis & Execution

```
(define (eval exp env)
  ((analyze exp) env))
```

analyze returns an execution procedure (λ (env) ...)

```
(define (analyze exp)
  (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))))
```

Same case analysis as before, except that helper perform only analysis, not full evaluation

First-class Values

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))
```

lambda that returns the quoted value

lambda that returns a fresh procedure

Variables

All of these return lambdas that do the actual work

```
(define (analyze-variable exp)
  (lambda (env) (lookup-variable-value exp env)))

(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env)
      (set-variable-value! var (vproc env) env)
      'ok)))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env)
      (define-variable! var (vproc env) env)
      'ok)))
```

Sequence & Selection

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

All of these return lambdas that do the actual work

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

lambda-glue

glue together list of lambdas

Procedure Application

All of these return lambdas that do the actual work

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      (execute-application (fproc env)
                           (map (lambda (aproc) (aproc env))
                                aprocs)))))
(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

Obtain actual procedure

Obtain actual arguments

procedure body already analyzed

Drawback of MC-evaluators so far

Control flow of the MC-evalled program piggy-backs on control flow of underlying Scheme evaluator

Not a good means to explain advanced control operators such as non-deterministic choice, backtracking, coroutines, generators, threads, …

“Current Continuations”

Not in the SICP-book

The “Mother of All Control Flow” that can be used to implement any control flow mechanism

GOTO on Steroids

convention in many Schemes

```
Welcome to DrRacket, version 6.10.1 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> call-with-current-continuation
#<procedure:call-with-current-continuation>
> (eq? call/cc call-with-current-continuation)
#t
>
```

∀ Scheme expression ∃ continuation

I'm waiting for a value and I will compare that value to n and then I will do an if-test and ....

```
(letrec ((fac
          (lambda (n)
            (if (= n 0)
              1
              (* n (fac (- n 1)))))))
  (fac 10))
```

I'm waiting for a value and I will print that value on the screen (because I'm the REPL) and then I will wait for new input and ...

I'm waiting for a value and I will use that value to call fac recursively and when it returns, I will multiply the result with n and ...

"Give me the value of the expression and I will to the rest of the computation"

The Continuation of an Expression

The continuation of an expression is a lambda with 1 parameter (the value of that expression) that contains the rest of the computation to be done.

Scheme always knows about "the current" continuation

```
(call/cc
  (lambda (cont)
    ... expression ...))
```

call/cc immediately calls its argument lambda with the current continuation

Silly Example

```
((( ... (( ...
  ((( ... ((( .... (call/cc
                      (lambda (ignore)
                         (+ 3 4))))))
        ))   ) )) .. ))
```

call/cc expects a lambda of one parameter

which will be its own continuation (= everything outside that callcc)

and it just evaluates the body of that lambda

Let us now use the continuation itself

```
> (define *global* '())
> (+ 5 (call/cc
        (lambda (my-future)
          (set! *global* my-future)
          4)))
9
> (*global* 20)
25
> (*global* 30)
35
>
```

The continuation expects the value of the expression

~ goto the past

Call it with a new value over and over again

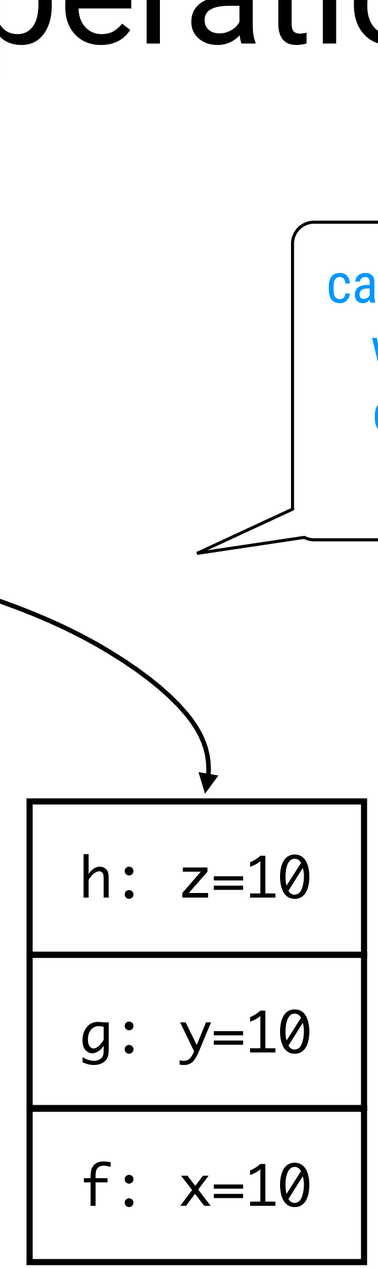
Explained Operationally

```
v

h(z) { …
       v = callcc
      }
g(y) { …
       h(y)
       …}
f(x) { …
       g(x)
       …}

f(10)
```

calling v from within any context will replace the stack of that context by f and unwind this computation



First Example: Goto

```
(define (onetwothree)
    (let ((start (label)))
      (display "one")
      (newline)
      (display "two")
      (newline)
      (display "three")
      (newline)
      (goto start)))
```

Goto: Implementation

```
(define (label)
  (call/cc (lambda (cont) cont)))

(define (goto cont)
  (cont cont))
```

```
(define (onetwothree)
    (let ((start (label)))
      (display "one")
      (newline)
      (display "two")
      (newline)
      (display "three")
      (newline)
      (goto start)))
```

Second Example: Coroutines

```
(define p1
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 1 ")
       (display input)
       (newline)
       (set! input (transfer p2 (+ input 1)))
       (loop))
     (loop))))
```

```
(define p2
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 2 ")
       (display input)
       (newline)
       (set! input (transfer p3 (+ input 1)))
       (loop))
     (loop))))
```

```
(define p3
  (new-process
   (lambda (input)
     (define (loop)
       (display "Tick 3 ")
       (display input)
       (newline)
       (set! input (transfer p1 (+ input 1)))
       (loop))
     (loop))))
```

```
(transfer p1 0)
```

Initiate the computation in the REPL

Coroutines: First Implementation

```
(define *current-process* #f)

(define (new-process proc)
  (define cont proc)
  (define (resume input)
    (cont input))
  (define (suspend saved)
    (set! cont saved))
  (lambda (msg)
    (cond ((eq? msg 'suspend) suspend)
          ((eq? msg 'resume) resume)
          (else (error)))))

(define (suspend-process process saved)
  ((process 'suspend) saved))

(define (resume-process process input)
  ((process 'resume) input))
```

```
(define (transfer process input)
  (call/cc (lambda (saved)
             (when *current-process*
               (suspend-process *current-process* saved))
             (set! *current-process* process)
             (resume-process process input))))
```

Coroutines: Second Implementation

```
(define (new-process proc)
  (cons 'process proc))

(define *current-process* (new-process #f))

(define (transfer process input)
  (call/cc
   (lambda (saved)
     (set-cdr! *current-process* saved)
     (set! *current-process* process)
     ((cdr process) input))))
```

Third Example: Exceptions

```
(define (divide x y)
  (if (= y 0)
    (throw 'division-by-zero)
    (div x y)))

(define (logarithm x y)
  (if (or (<= x 0)
          (<= y 0))
    (throw 'negative-log)
    (divide (log x) (log y))))
```

```
(define (do-it x y)
  (try-catch
   (lambda ()
     (try-catch
      (lambda ()
        (logarithm x y))
      (lambda (exception)
        (eq? exception 'division-by-zero))
      (lambda (exception)
        (display "x/0"))))
   (lambda (exception)
     (eq? exception 'negative-log))
   (lambda (exception)
     (display "log(-x)"))))
```

code to be tried

exception filter

exception handler

The Implementation

```
(define (throw exception)
   (error "No Exception Handler " exception))

 (define (try-catch
          try-lambda
          filter
          handler)
   (call-with-current-continuation
    (lambda (cont)
      (define keep throw)
      (set! throw (lambda (exception)
                    (set! throw keep)
                    (if (filter exception)
                      (cont (handler exception))
                      (throw exception))))
      (let ((result (try-lambda)))
        (set! throw keep)
        result))))
```

store the old handler

do try-block

restore the old handler

grab the continuation

the handler at this level restores the old handler

and jumps out of this level after running the exception handler code

or just rethrows the exception by invoking the old handler

Revisited: Drawback of MC-evals

It does not explain

call/cc

Towards a CPS-interpreter

CPS-Evaluator

Continuation-passing style

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

REPL continuation

extra parameter: continuation

eval = iterative process!

```
(define (cnteval exp env cnt)
  ((analyze exp) env cnt))
```

identical structure

First-class Value Expressions

Just pass the value over to the continuation

```
(define (analyze-self-evaluating exp)
  (lambda (env cnt)
    (cnt exp)))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env cnt)
      (cnt qval))))


(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env cnt)
      (cnt (make-procedure vars bproc env)))))
```

Control-flow Expressions

Continuation-passing style

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env cnt)
      (pproc env
             (lambda (pred-value)
               (if (true? pred-value)
                   (cproc env cnt)
                   (aproc env cnt)))))))
```

Compose continuations

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env cnt)
      (a env
         (lambda (a-value)
           (b env cnt)))))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

Variables

```
(define (analyze-variable exp)
  (lambda (env cnt)
    (cnt (lookup-variable-value exp env))))

(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env cnt)
      (vproc env
             (lambda (val)
               (define-variable! var val env)
               (cnt 'ok))))))

(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env cnt)
      (vproc env
             (lambda (val)
               (set-variable-value! var val env)
               (cnt 'ok))))))
```

Identical

Identical

Procedure Application (1/2)

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env cnt)
      (fproc env
             (lambda (proc)
               (get-args aprocs
                         env
                         (lambda (args)
                           (execute-application
                            proc args cnt))))))))

(define (get-args aprocs env cnt)
  (if (null? aprocs)
      (cnt '())
      ((car aprocs) env
                    (lambda (arg)
                      (get-args (cdr aprocs)
                                env
                                (lambda (args)
                                  (cnt (cons arg args)))))))
```

call more specific evaluation procedures

Iterative!

Procedure Application (2/2)

```
(define (execute-application proc args cnt)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args cnt))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))
          cnt))
        ((continuation? proc)
         ((continuation-continuation proc) (car args)))
        (else
         (error
          "Unknown procedure type -- EXECUTE-APPLICATION"
          proc))))
```

An application expression can be interpreted in 3 different ways

Ignore cnt: use the grabbed continuation

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

cadr

Primitives: Representation

```
(define (do-call-cc cont proc)
  ((procedure-body proc)
   (extend-environment (procedure-parameters proc)
                       (list (make-continuation cont))
                       (procedure-environment proc))
   cont))

(define primitive-procedures
  (let ((cc/ise-it (lambda (p)
                     (lambda args
                       (apply-in-underlying-scheme p (cdr args))))))
        (list (list 'car   (cc/ise-it car))
              (list 'cdr   (cc/ise-it cdr))
              (list 'cons  (cc/ise-it cons))
              (list '+     (cc/ise-it +))
              (list '-     (cc/ise-it -))
              (list 'eq?   (cc/ise-it eq?))
              (list 'member    (cc/ise-it member))
              (list 'memq  (cc/ise-it memq))
              (list 'call/cc do-call-cc))))
```

list of 1 parameter

"lift" p to the cps-style

Continuations: Representation

A continuation is 1-parameter lambda

```
(define (make-continuation cont)
  (list 'continuation cont))

(define (continuation? c)
  (tagged-list? c 'continuation))

(define (continuation-continuation p) (cadr p))
```

Chapter 5