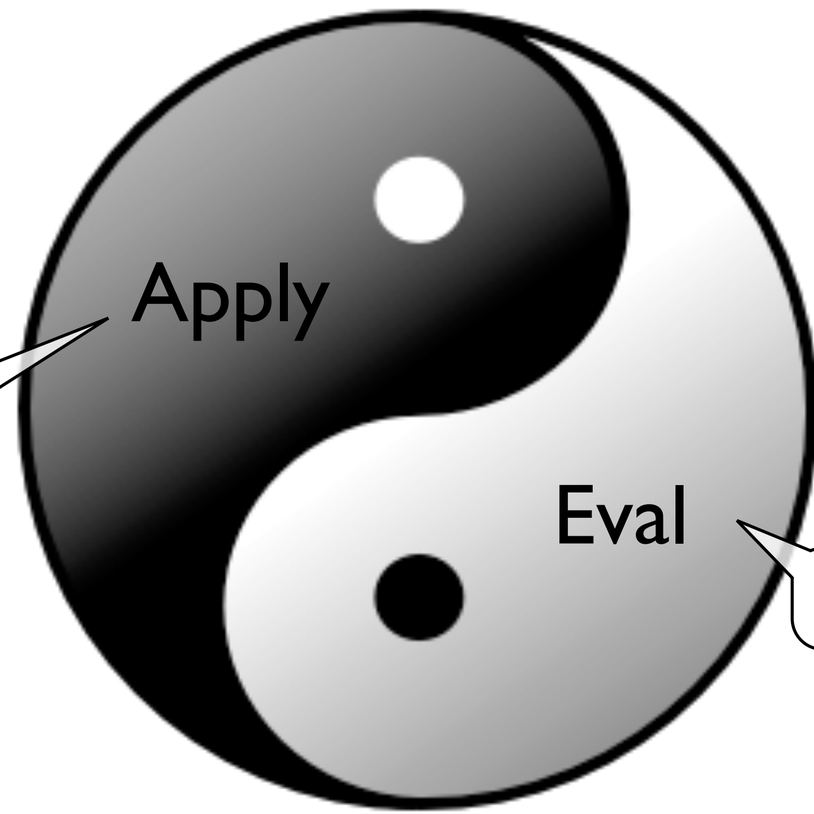
Chapter 4

Unfolding the Mystery: Scheme Evaluators

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

The Interpreter is just another program

written in Scheme!



procedure + arguments

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

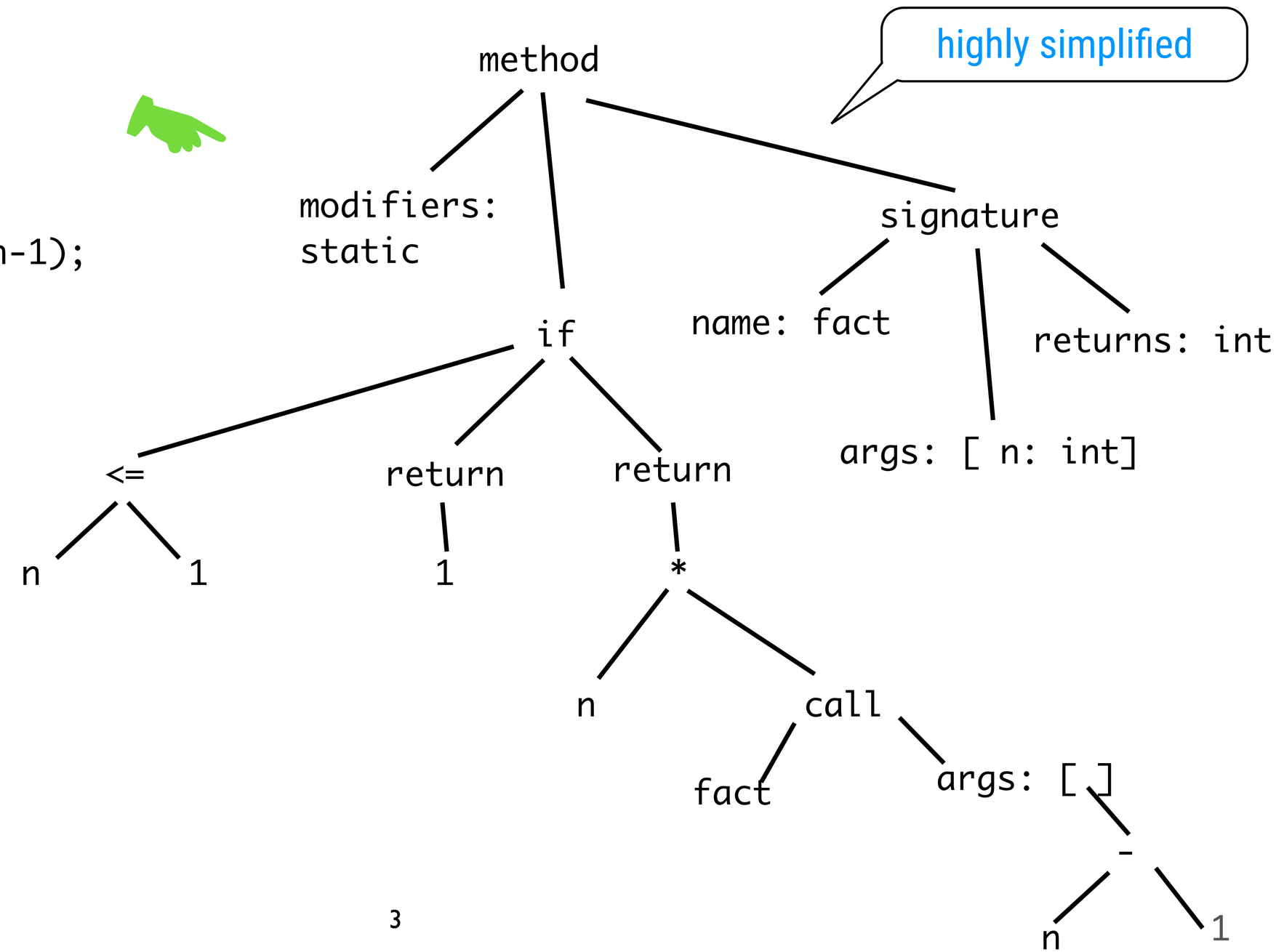
expression + environment

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

Keep the environment model in mind

Concrete vs. Abstract Syntax

```
static int fact(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return n * fact(n-1);
    }
```



Concrete vs. Abstract Syntax

Same example in Scheme

Turn program into data structure

```
(define (fact n)
  (if (<= n 1)
      1
      (* n (fact (- n 1)))))
```

```
‘(define (fact n)
   (if (<= n 1)
       1
       (* n (fact (- n 1)))))
```

The Scheme evaluator is just a list processing program!

The Evaluator as a Scheme Program

```
(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)
```

We will not focus on read

Go into an infinite loop

IO

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

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

(define (announce-output string)
  (newline) (display string) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

Don't show the environment

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))))
```

call more specific evaluation procedures

special forms are special

syntactic sugar

apply is the regular case

Representing Expressions (1)

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        (else #f)))

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

Simple expressions

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

Compound expressions

First example

(quote exp)

```
(define (quoted? exp)
  (tagged-list? exp 'quote))

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

Accessor

Representing Expressions (2)

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

(set! var val)

```
(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
```

(define var val)

(define (proc ...) val)

Syntactic sugar

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

Representing Expressions (3)

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

(lambda pars body)

(if pred cons alt)

```
(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))
```

Evaluating cond
syntactic sugar (later)

Representing Expressions (4)

```
(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))
```

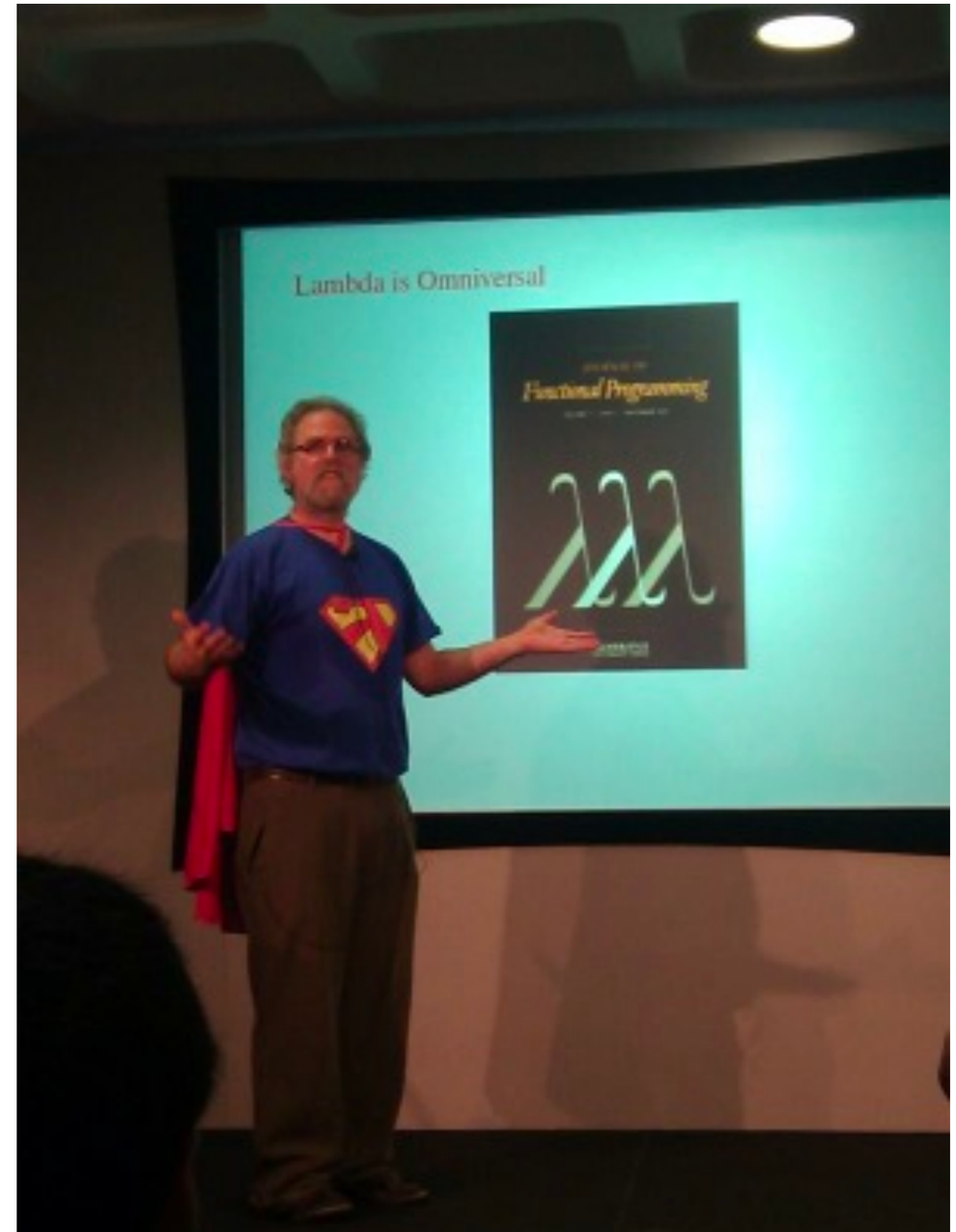
Evaluating cond
syntactic sugar (later)

The Evaluator Parts

```
(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)

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

(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))))
```

Representation of Procedures

```
(define (make-procedure parameters body env)
  (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))
```



The Scheme Evaluator: apply

already evaluated!

```
(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))))
```

body

local environment

Remember...

```
(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))))
```

Syntactic sugar

Syntactic Sugar

```
(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))))))
```

Environments are Lists of Frames

```
(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)))
  (set-cdr! frame (cons val (cdr frame))))

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

Variables with their corresponding values

Lists of Frames

Lookup Variable Value

Mutual recursive search process

```
(define (lookup-variable-value var env)
  (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))
```

lookup in current frame

lookup in next frame

Ended by a read

Set Variable Value

Mutual recursive search process

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (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))
```

lookup in current frame

lookup in next frame

Ended by a write

Adding Identifiers to Environment

add or change binding in first frame

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (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))))
```

Remember

```
(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))))
```

Primitive Procedures

```
(define (primitive-procedure? proc)
  (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?)
        ))
(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))
```

Built-in apply

```
(define apply-in-underlying-scheme apply)
```

The Global Environment

```
(define (setup-environment)
  (let ((initial-env  (extend-environment (primitive-procedure-names)
                                          (primitive-procedure-objects)
                                          the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))

(define the-global-environment (setup-environment))
```

A Variation on the Interpreter

```
(define (try a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0))
```

Applicative-Order Evaluation

```
Welcome to DrRacket, version 6.8 [3m].
Language: racket, with debugging; memory limit: 4096 MB.
> (define (try a b)
    (if (= a 0) 1 b))
> (try 0 (/ 1 0))
/: division by zero
>
```

Normal-Order Evaluation

```
(try 0 (/ 1 0))   ⟹ 1
```

Delay arguments into "thunks"

Example (More: Haskell!)

```
(define (list-ref items n)
  ...)

(define (map proc items)
  ...)
```

Chapter 2

```
(define (add-lists list1 list2)
  (cond ((null? list1) list2)
        ((null? list2) list1)
        (else (cons (+ (car list1) (car list2))
                    (add-lists (cdr list1) (cdr list2))))))
```

No force/delay!

```
(define ones (cons 1 ones))
(define integers (cons 1 (add-lists ones integers)))
```

No special stream procedures needed

Procedure Application: Arguments

unevaluated operand expressions!

Extra parameter

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

Applicative-order evaluation

```
((compound-procedure? procedure)
        (eval-sequence
          (procedure-body procedure)
          (extend-environment
            (procedure-parameters procedure)
            arguments
            (procedure-environment procedure)))
```

Normal-order evaluation

Interpreter with Lazy Evaluation

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

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

force evaluation of the procedure

don't evaluate operands

```
((application? exp)
 (apply (eval (operator exp) env)
        (list-of-values (operands exp) env)))
```

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

force evaluation of the condition

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

REPL prints values

```
(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))
```

force evaluation of value to be printed

Normal Order vs. Applicative Order

Applicative-order evaluation

```
(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))))
```

instead of `eval`

Normal-order evaluation

```
(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))))
```

later

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

Thunks: Naive Version

Representing delayed computations

inefficient → memoize...

```
(define (thunk? obj)
  (tagged-list? obj 'thunk))

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))
```

Actually a parameterless lambda

```
(define (delay-it exp env)
  (list 'thunk exp env))

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

force until non-thunk value

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

Thunks: Improved Version

```
(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)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

Call the thunk

Memoize result

Help GC

Return Memoized Value

Chapter 4