6.001 SICP
Variations on a Scheme

- Scheme Evaluator – A Grand Tour
  - Making the environment model concrete
  - Defining eval defines the language
  - Provides mechanism for unwinding abstractions

- Techniques for language design:
  - Interpretation: eval/apply
  - Semantics vs. syntax
  - Syntactic transformations

- Beyond Scheme – designing language variants
  - Lexical scoping vs. Dynamic scoping

Building up a language...

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

Stages of an interpreter

Lexical analyzer
Parser
Evaluator
Environment
Printer

"(average 4 (+ 5 5))"

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

Building up a language...

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

Meval

(meval exp env)

(define (meval 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) (meval (cond->if exp env))
                    (eval (application exp)
                          (meval (meval (operator exp) env)
                                 (list-of-values (operands exp) env)))
                    (else (error "Unknown expression type -- EVAL" exp))))

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop

The Core Evaluator

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
Basic Semantics: m-eval & m-apply

- primitive expressions
  - self-evaluating, quoted
- variables and the environment
  - variable definition, lookup, and assignment
- conditionals
  - if, cond
- procedure creation
- sequences
  - Begin
- procedure application

Pieces of Eval&Apply

(define (meval 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) (mapply (meval (operator exp) env)
                                    (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

Pieces of Eval&Apply

(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env))))

Side comment – procedure body

- The procedure body is a sequence of one or more expressions:
  (define (foo x)
    (do-something (+ x 1))
    (* x 5))
- In m-apply, we eval-sequence the procedure body.

Pieces of Eval&Apply

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
(define (meval 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) (mapply (meval (operator exp) env)
                                    (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (meval (assignment-value exp) env) env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (meval (definition-value exp) env) env))

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

(define (tagged-list? Exp tag)
  (and (pair? Exp) (eq? (car exp) tag)))

(define (if? exp) (tagged-list? exp 'if))
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (application? exp) (tagged-list? exp 'application))
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
Example – Changing Syntax

- Suppose you wanted a "verbose" application syntax, i.e.,

  instead of

  
  <proc> <arg1> <arg2> . . .

  USE

  (CALL <proc> ARGS <arg1> <arg2> ...)

- Changes – only in the syntax routines!

  (define (application? exp) (tagged-list? exp 'CALL))
  (define (operator app) (cadr app))
  (define (operands app) (cdddr app))

Implementing "Syntactic Sugar"

- Idea:
  – Easy way to add alternative/convenient syntax
  – Implement a simpler "core" in the evaluator

- "let" as sugared procedure application:

  (let ((<name1> <val1>)
        (<name2> <val2>))
       <body>)

Detect and Transform the Alternative Syntax

(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp)
         (text-of-quotation exp))
        ...
        ((let? exp)
         (m-eval (let->combination exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env)
                  (list-of-values
                   (operands exp) env)))
        (else (error "Unknown expression" exp))))

Let Syntax Transformation

(from
  (let ((x 23)
       (y 15))
     (dosomething x y))

(to
  (lambda (x y) (dosomething x y))
     23 15)

Let Syntax Transformation

(define (let? exp) (tagged-list? exp 'let))
(define (let-bound-variables let-exp) (map car (cadr let-exp)))
(define (let-values let-exp) (map cadr (cadr let-exp)))
(define (let-body let-exp) (sequence->exp (cddr let-exp)))
(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-values let-exp))
        (body (let-body let-exp)))
  (cons (list 'lambda names body) values)))

Defining Procedures

(define foo (lambda (x) <body>))
(define (foo x) <body>)

- Semantic implementation – just another define:

  (define (eval-definition exp env)
    (define-variable! (definition-variable exp)
      (m-eval (definition-value exp) env)
      env))

- Syntactic transformation:

  (define (definition-value exp)
    (if (symbol? (cadr exp))
        (caddr exp)
      (make-lambda (cdadr exp) ;formal params
                    (cddr exp))) ;body

NOTE: only manipulates list structure, returning new list structure that acts as an expression
How the Environment Works

- Abstractly – in our environment diagrams:
  - Environment diagrams:
    - E1
      - x: 10
      - plus: (procedure ...)
    - E2
      - x: 10
      - plus: (procedure ...)
    - E3
      - x: 4
      - y: 5
- Concretely – our implementation (as in SICP)

Extending the Environment

- (extend-environment '(x y) (list 4 5) E2)

"Scanning" the environment

- Look for a variable in the environment...
  - Look for a variable in a frame...
    - Loop through the list of vars and list of vals in parallel
    - Detect if the variable is found in the frame
  - If not found in frame (out of variables in the frame), look in enclosing environment

The Initial (Global) Environment

- setup-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))

Read-Eval-Print Loop

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

Variations on a Scheme

- More (not-so) stupid syntactic tricks
  - LetSeq:
    - (letseq ((x 4)
      (y (+ x 1))) . . . )
  - Infix notation:
    - (+(* 4 3)?) → (((4 * 3) + 7)
Diving in Deeper: Lexical Scope

- How does our evaluator achieve lexical scoping?
  - environment chaining
  - procedures that capture their lexical environment

- make-procedure:
  - stores away the evaluation environment of lambda
  - the "evaluation environment" is always the enclosing lexical scope

(define (foo x y) ...)
(define (bar l)
  (define (baz m) ...)
  ...)

Lexical Scope & Environment Diagram

(define (foo x y)
  (lambda (z) (+ x y z)))

GE

p: x y
body: ([+ x y z])
foo: E1 x: 1
y: 2
(+ x y z) | E2 => 6

Alternative Model: Dynamic Scoping

- Dynamic scope:
  - Look up free variables in the caller's environment rather than the surrounding lexical environment

- Example:

(define (pooh x)
  (bear 20))
(define (bear y)
  (+ x y))
(pooh 9)

Dynamic Scope & Environment Diagram

(define (pooh x)
  (bear 20))
(define (bear y)
  (+ x y))
(pooh 9)

GE

p: x
body: (bear 20)
pooh: E1 x: 9

A "Dynamic" Scheme

(define (m-eval exp env)
  (cond
   ((self-evaluating? exp) exp)
   ((variable? exp) (lookup-variable-value exp env)) ...
   ((lambda? exp)
    (make-procedure (lambda-parameters exp)
      (lambda-body exp)
      '*no-environment')) :CHANGE: no env ...)
   ((application? exp)
    (d-apply (m-eval (operator exp) env)
      (list-of-values (operands exp) env) env)
      :CHANGE: add env
    (else (error "Unknown expression -- M-EVAL" exp)))))
A "Dynamic" Scheme – d-apply

(define (d-apply procedure arguments calling-env)
  (cond ((primitive-procedure? procedure)
             (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
             (eval-sequence
              (procedure-body procedure)
              (extend-environment
               (procedure-parameters procedure)
               arguments
               calling-env)))
       (else (error "Unknown procedure" procedure))))

Summary

- Scheme Evaluator – Know it Inside & Out
- Techniques for language design:
  - Interpretation: eval/apply
  - Semantics vs. syntax
  - Syntactic transformations
- Able to design new language variants!
  - Lexical scoping vs. Dynamic scoping