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

Eval
exp & env
proc & args
Apply

• Core evaluator
  • eval: dispatch on expression type
  • apply: eval args then apply operator

How to describe Eval?

Meval

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

Mapply

(define (mapply 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))))

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

Pieces of Eval&Apply

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

Pieces of Eval&Apply

(defn (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)))))

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)))))

Pieces of Eval&Apply

(define (mapply 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)))))

Pieces of Eval&Apply

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

Pieces of Eval&Apply

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

Syntactic Abstraction

• Semantics
  • What the language means
  • Model of computation

• Syntax
  • Particulars of writing expressions
  • E.g. how to signal different expressions

• Separation of syntax and semantics:
  allows one to easily alter syntax
Basic Syntax
(define (tagged-list? Exp tag)
  (and (pair? Exp) (eq? (car exp) tag)))

• Routines to detect expressions
  (define (if? exp) (tagged-list? exp 'if))
  (define (lambda? exp) (tagged-list? exp 'lambda))

• Routines to get information out of expressions
  (define (operator app) (car app))
  (define (operands app) (cdr app))

• Routines to manipulate expressions
  (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:
  (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) (caddr app))

Implementing "Syntactic Sugar"
• Idea:
  • Implement a simple fundamental "core" in the evaluator
  • Easy way to add alternative/convenient syntax?
  • "let" as sugared procedure application:
    (let ([<name1> <val1>]
          [<name2> <val2>])
        <body>)

      ((lambda <name1> <name2>)   <body>)
      (cadr app)

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)
          (let->combination exp))
        ((application? exp)
          (m-apply (m-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else (error "Unknown expression" exp))))

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)))

NOTE: only manipulates list structure, returning new list structure that acts as an expression

Details of let syntax transformation
(let ((x 23)
      (y 15))
  (dosomething x y))

(x 23)
(y 15)
(dosomething x y)
Details of let syntax transformation

\[
\text{let}\ y\ x\ \text{do}\ \text{smthg}
\]

\[
\lambda\ y\ x\ \text{do}\ \text{smthg}
\]

(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)))

Named Procedures – Syntax vs. Semantics

\[
\text{define } (\text{foo } \text{param}) \text{ exp}
\]

- Semantic implementation – just another define:
  \[
  \text{define-variable}! \left(\text{definition-variable exp}\right)\left(\text{def-at}\text{-}\text{env exp env}\right)
  \]

- Syntactic transformation:
  \[
  \text{define } \left(\text{definition-value exp}\right)
  \text{(if symbol? (cadr exp)) \left(\text{define } \text{foo } \text{bars} \text{ exp}\right)}
  \text{(caddr exp)}
  \text{make-lambda (cadr exp) \text{formal params} (\text{cddr exp}) \text{body}}
  \]

How the Environment Works

- Abstractly – in our environment diagrams:

- Concretely – our implementation (as in SICP)

Extending the Environment

\[
\text{extend-environment } \left(\text{x y} \text{ (list 4 5)}\right) \text{ E2}
\]

- Abstractly

- Concretely

Implementing the environment

\[
\text{define } \left(\text{enclosing-environment env} \text{ (cdr env)}\right)
\text{define } \left(\text{first-frame env} \text{ (car env)}\right)
\text{define } \left(\text{the-empty-environment} \text{ '()}\right)
\text{define } \left(\text{make-frame variables values}\right)
\text{define } \left(\text{frame-variables frame}\right)
\text{define } \left(\text{frame-values frame}\right)
\text{define } \left(\text{add-binding-to-frame! var val frame}\right)
\text{define } \left(\text{extend-environment vars vals base-env}\right)
\text{define } \left(\text{length vars length vals}\right)
\text{define } \left(\text{if symbol? (cadr exp)}\right)
\text{define } \left(\text{error "Too many args supplied" vars vals}\right)
\text{define } \left(\text{error "Too few args supplied" vars vals}\right)
\]

"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
Scanning the environment (details)

(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 -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame))))
  (env-loop env))

The Initial (Global) Environment 4.

• 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))

• define initial variables we always want
• bind explicit set of "primitive procedures"
  • here: use underlying scheme
  • in other interpreters: assembly code, hardware, ....

Read-Eval-Print Loop 5.

(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)))
  (driver-loop))

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
  • why?
    • our semantic rules for procedure application!
    • "hang a new frame"
    • "bind parameters to actual args in new frame"
    • "evaluate body in this new environment"

Lexical Scope & Environment Diagram

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

GE

| foo:   | bar:
|-------|-------|
| p: x y
| body:
| k (x)
| (+ x y z)
| (x y z) | (x y z)  | x2 => 6

Will always evaluate (+ x y z) in a new environment inside the surrounding lexical environment.

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) => 29
Dynamic Scope & Environment Diagram

```
(define (pooh x)
  (bear 20))
```

Will evaluate `(+ x y)` in an environment that extends the caller's environment.

```
(define (bear y)
  (+ x y))
```

(`pooh 9`) Will evaluate `(+ x y)` in an environment that extends the caller's environment.

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))) ;CHANGE: use 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