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

The Core Evaluator

1. eval/apply core

How to describe Eval?

Meval

 primitives
 special forms
 application

How to describe?
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))))
```

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

Pieces of Eval&Apply

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

Basic Syntax

(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) (pair? exp))

define (operator app) (car app))
(define (operands app) (cdr app))

define (no-operands? args) (null? args))
(define (first-operands args) (car args))
(define (rest-operands args) (cdr args))
Example – Changing Syntax

• Suppose you wanted a "verbose" application syntax, i.e., instead of

  \((\text{proc} \ \text{arg1} \ \text{arg2} \ . \ . \ .)\)

  USE

  \((\text{CALL} \ \text{proc} \ \text{ARGS} \ \text{arg1} \ \text{arg2} \ . \ . \ .)\)

• 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 {\((\text{name1} \ \text{val1})\)
   \((\text{name2} \ \text{val2})\))
   \text{body}}\)

  \{(\lambda {\((\text{name1} \ \text{name2})\)} \text{body})
   \text{val1} \text{val2}\)

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)))

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))
Details of let syntax transformation

Defining Procedures

(let
  
  y x
dosomething
)

(defn foo (lambda [x] <body>))
(defn (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

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

How the Environment Works

• Abstractly – in our environment diagrams:
  3.

• Concretely – our implementation
  (as in SICP)

Extending the Environment

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

  Abstractly

  Concretely

"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)))

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

- **setup-environment**
  ```scheme```
  ```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))```
  ```end-define```
```scheme```
- define initial variables we always want
- bind explicit set of "primitive procedures"
  - here: use underlying scheme
  - in other interpreters: assembly code, hardware, ...

Variations on a Scheme

- More (not-so) stupid syntactic tricks
  - LetSeq: `(letseq ((x 4)`
    `(y (+ x 1))) . . . )`
  - Infix notation:
    `(+(* 4 3)7) ➔ ((4 * 3)+ 7)`
- Semantic variations
  - *Lexical vs dynamic scoping*
    - Lexical: defined by the text
    - Dynamic: defined by the runtime behavior

Diving in Deeper: 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)))
(define bar (foo 1 2))
```

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:

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

**Dynamic Scope & Environment Diagram**

```
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Environment</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pooh</td>
<td>x: 9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body:</td>
<td>(bear 20)</td>
</tr>
<tr>
<td>bear</td>
<td>p: y</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>body:</td>
<td>(+ x y)</td>
</tr>
</tbody>
</table>
```

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

**A "Dynamic" Scheme**

```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 (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**

```scheme
(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