

## 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

1/40

## Building up a language...

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

2/40

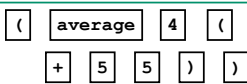
## Stages of an interpreter

input to each stage

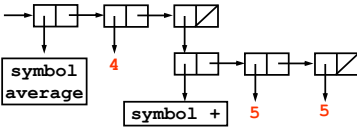
Lexical analyzer

"(average 4 (+ 5 5))"

Parser



Evaluator



Environment

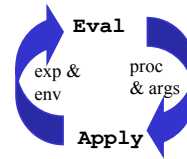
Printer

"7"

3/40

## The Core Evaluator

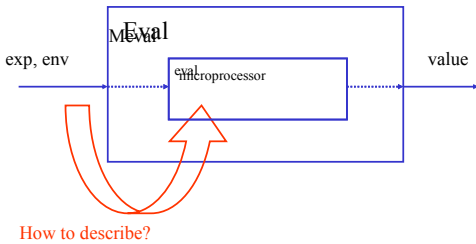
1. eval/apply core



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

4/40

## How to describe Eval?



5/40

## 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))))
```

6/40

## 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

7/40

## 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))))
```

8/40

## 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.

9/40

## 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))))
```

10/40

## 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)))
```

List structure

```
(eval-if (if (= n 0) 1 (* n (fact (- n 1)))) env)
```

```
(if (m-eval (= n 0) env)
    (m-eval 1 env)
    (m-eval (* n (fact (- n 1))) env))
```

11/40

## 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))))
```

12/40

## 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))))
```

13/40

## 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))))
```

14/40

## 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)))))

(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))))
```

15/40

## 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))))
```

16/40

## Pieces of Eval&Apply

```
(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))
```

17/40

## Syntactic Abstraction

2.

syntax  
procedures

- 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



18/40

## 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))
(define (application? exp) (pair? exp))
```

- 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))
```

19/40

## 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))
```

20/40

## 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>)
 <val1> <val2>)
```

21/40

## 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))))
```

22/40

## 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 (caddr let-exp)))
```

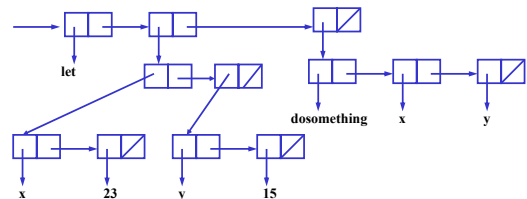
```
(define (let->combination let-exp)
  (let ((names (let-bound-variables let-exp))
        (values (let-values let-exp))
        (body (let-body let-exp)))
    (cons (list 'lambda names body)
          (cons values))))
```

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

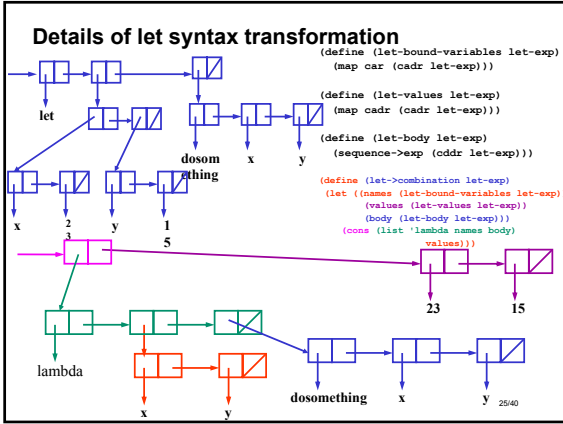
23/40

## Details of let syntax transformation

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



24/40



### Named Procedures – Syntax vs. Semantics

```

(define (foo <param>) <body>)

```

- Semantic implementation – just another define:
 

```

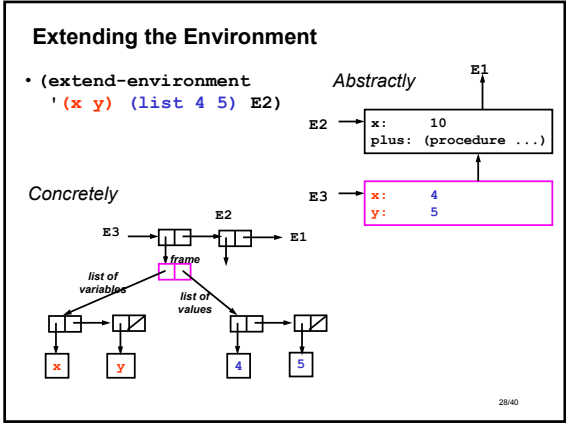
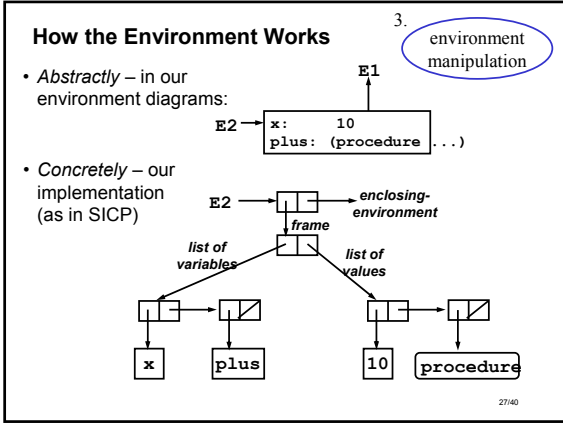
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))

```
- Syntactic transformation:
 

```

(define foo bar)
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (define (foo bar baz))
      (caddr exp)
      (make-lambda (caddr exp) ; formal params
                   (caddr exp)) ; body

```



### Implementing the environment

```

;; Implement environments as a list of frames; parent environment is
;; the cdr of the list. Each frame will be implemented as a list
;; of variables and a list of corresponding values.

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

(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 (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 args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

```

### "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)))
```

31/40

## The Initial (Global) Environment

4. primitives and initial env.

- 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, ....

32/40

## Read-Eval-Print Loop

5. 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)))
    (driver-loop)))
```

33/40

## 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*"

34/40

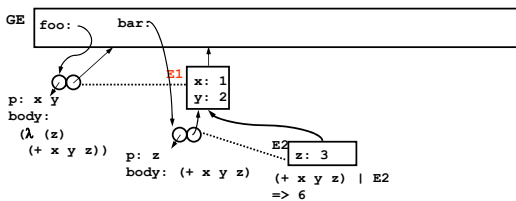
## Lexical Scope & Environment Diagram

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

(define bar (foo 1 2))

(bar 3)
```

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



35/40

## 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
```

36/40

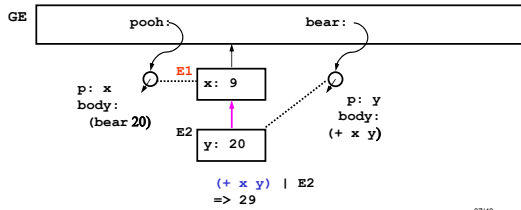
## Dynamic Scope & Environment Diagram

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

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

```
(pooh 9)
```

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



37/40

## 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))))
```

38/40

## 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))))
```

39/40

## 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

40/40