

6.184 Lecture 4

Interpretation

•Nelson Elhage <nelhage@mit.edu>

•Original material by Eric Grimson

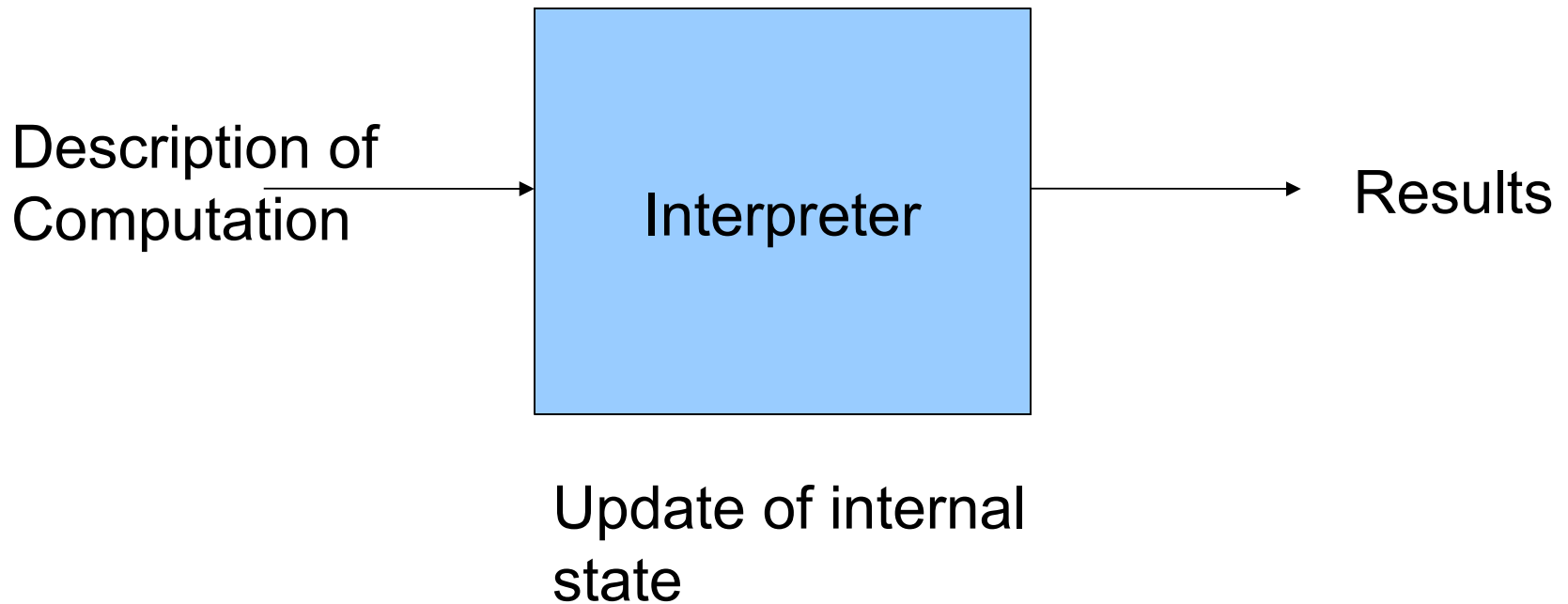
•Compiled by Mike Phillips <mpp@mit.edu>

•Tweaked by Ben Vandiver <benmv@mit.edu>

Interpretation

- Parts of an interpreter
- Arithmetic calculator
- Meta-circular Evaluator (Scheme-in-scheme!)
- A slight variation: dynamic scoping

What is an interpreter?



Why do we need an interpreter?

- Abstractions let us bury details and focus on use of modules to solve large systems
- We need a process to unwind abstractions at execution time to deduce meaning
- We have already seen such a process – [the Environment Model](#)
- Now want to describe that process as a procedure

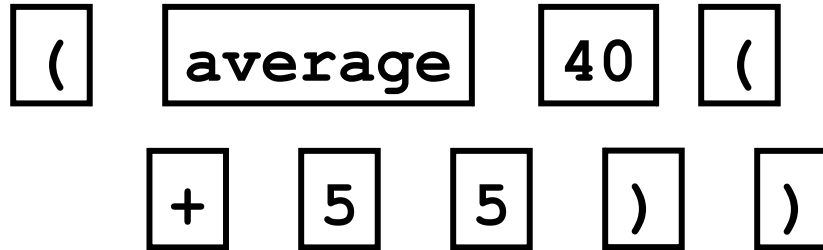
Stages of an interpreter

input to each stage

Lexical analyzer

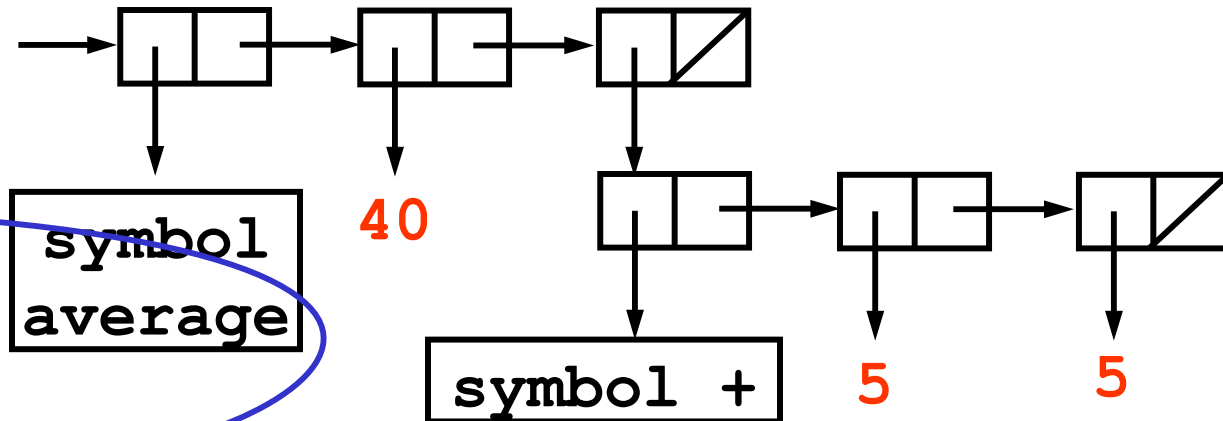
"(average 40 (+ 5 5))"

Parser



Environment

Evaluator



Printer

25

"25"

Role of each part of the interpreter

- Lexical analyzer
 - break up input string into "words" called tokens
- Parser
 - convert linear sequence of tokens to a tree
 - like diagramming sentences in elementary school
 - also convert self-evaluating tokens to their internal values
 - e.g., `#f` is converted to the internal false value
- Evaluator
 - follow language rules to convert parse tree to a value
 - read and modify the `environment` as needed
- Printer
 - convert value to human-readable output string

Our interpreters

- Only write evaluator and environment
 - Use Scheme's **reader** for lexical analysis and parsing
 - Use Scheme's **printer** for output
 - To do this, our language must resemble Scheme
- Start with interpreter for simple arithmetic expressions

Arithmetic calculator

Want to evaluate arithmetic expressions of two arguments, like:

```
(plus* 24 (plus* 5 6))
```

Arithmetic calculator

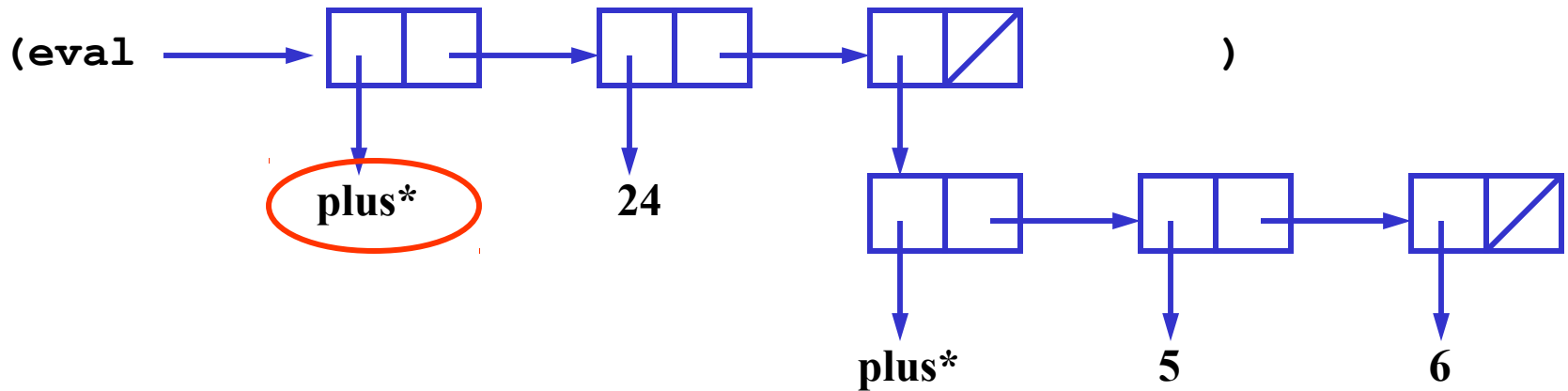
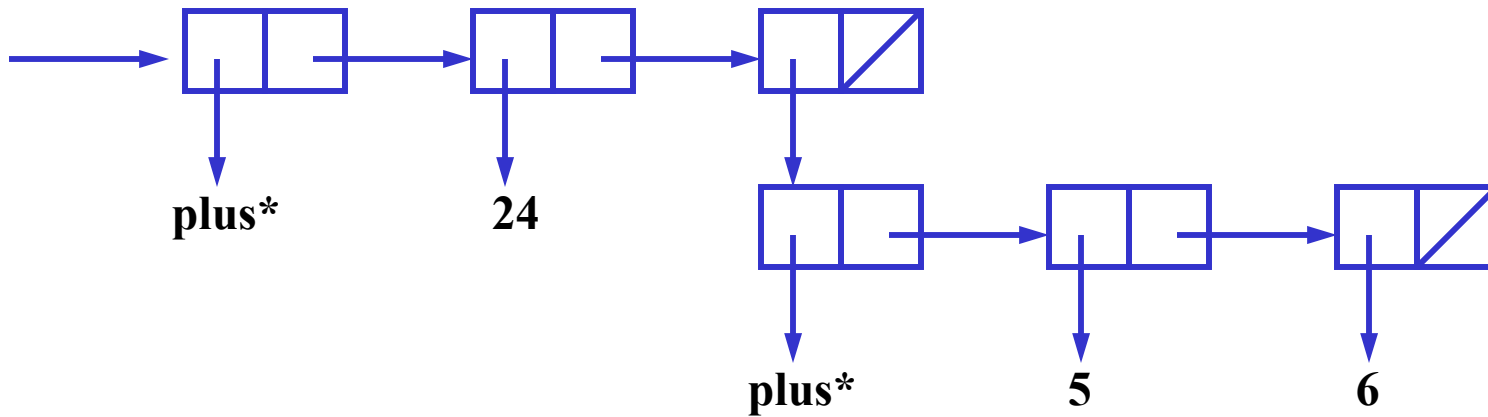
```
(define (tag-check e sym) (and (pair? e) (eq? (car e) sym)))  
(define (sum? e) (tag-check e 'plus*))
```

```
(define (eval exp)  
  (cond  
    ((number? exp) exp)  
    ((sum? exp) (eval-sum exp))  
    (else  
     (error "unknown expression " exp))))
```

```
(define (eval-sum exp)  
  (+ (eval (cadr exp)) (eval (caddr exp))))
```

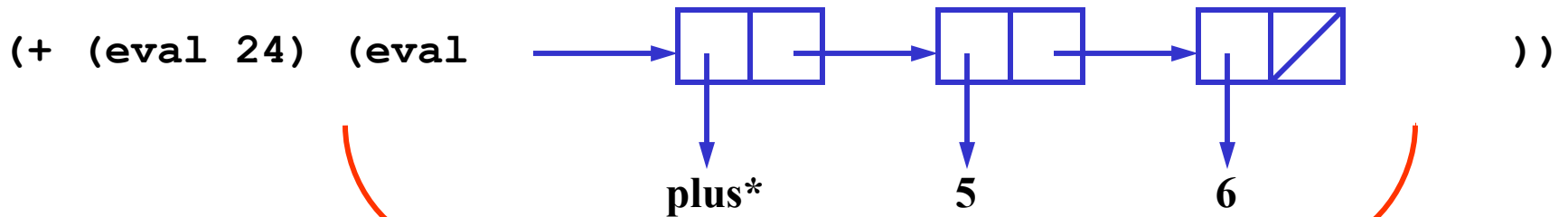
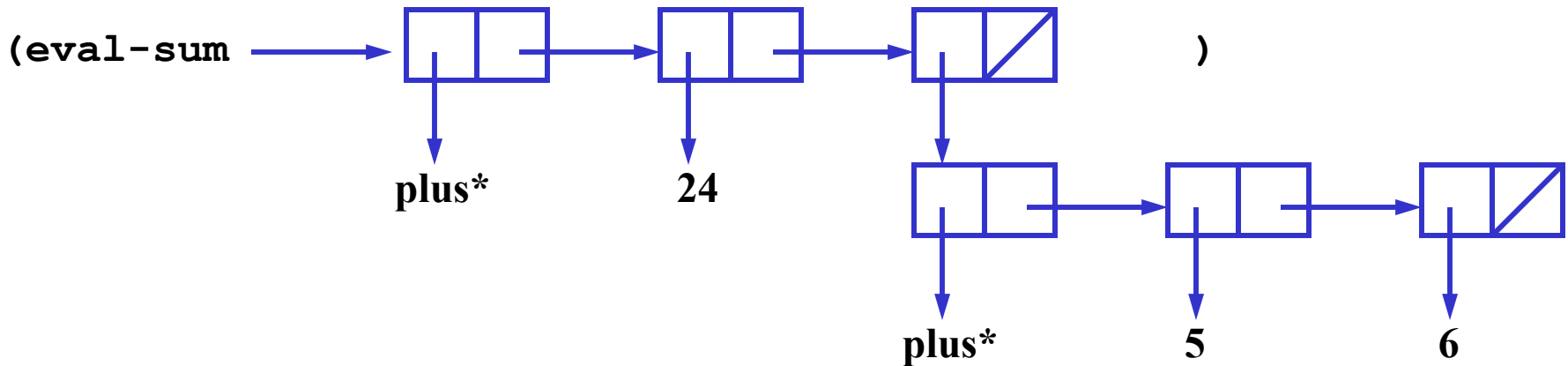
```
(eval '(plus* 24 (plus* 5 6)))
```

We are just walking through a tree ...



sum? checks the tag

We are just walking through a tree ...

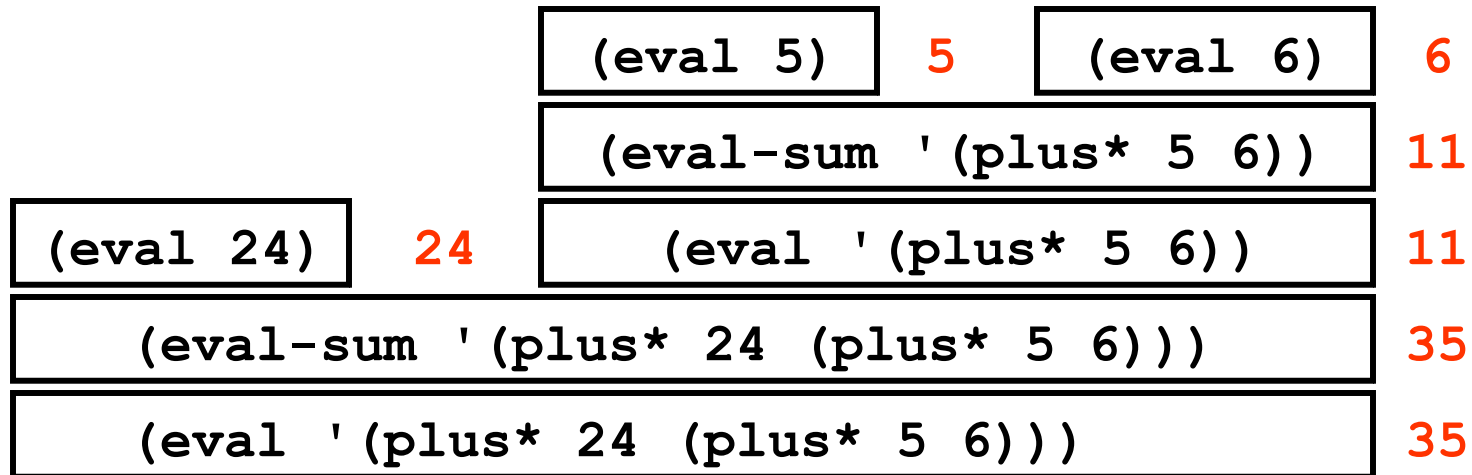


(+ (eval 5) (eval 6))

Arithmetic calculator

```
(plus* 24 (plus* 5 6))
```

- What are the argument and return values of `eval` each time it is called in the evaluation of this expression?



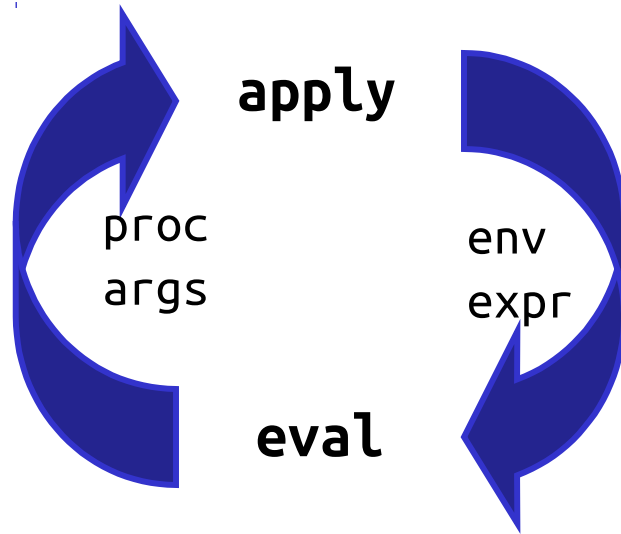
Things to observe

- `cond` determines the expression type
- No work to do on numbers
 - Scheme's reader has already done the work
 - It converts a sequence of characters like "24" to an internal binary representation of the number 24
 - ...self-evaluating!
- `eval-sum` recursively calls `eval` on both argument expressions

the Metacircular Evaluator

- And now a complete Scheme interpreter written in Scheme
- Why?
 - An interpreter makes things explicit
 - e.g., procedures and procedure application in the environment model
 - Provides a precise definition for what the Scheme language means
 - Describing a process in a computer language forces precision and completeness
 - Sets the foundation for exploring variants of Scheme
 - Today: lexical vs. dynamic scoping

The Core Evaluator



1. **eval/apply core**

```
(define (square x)
  (* x x))
```

```
(square 4)
```

```
x = 4
```

```
(* x x)
```

- Core evaluator
 - **eval**: evaluate expression by dispatching on type
 - **apply**: apply procedure to argument values by evaluating procedure body

Building up a language...

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

Metacircular evaluator (Scheme implemented in Scheme)

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

primitives

special forms

application

Pieces of Eval&Apply

```
(define (m-eval exp env)
  (cond ((self-evaluating? 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)
         (m-apply (m-eval (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)
  (map (lambda (exp) (m-eval exp env)) exps))
```

m-apply

```
(define (m-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))))
```

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 (m-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)
         (m-apply (m-eval (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)
                        (m-eval (assignment-value exp) env)
                        env))
```

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

Pieces of Eval&Apply

```
(define (m-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)
         (m-apply (m-eval (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

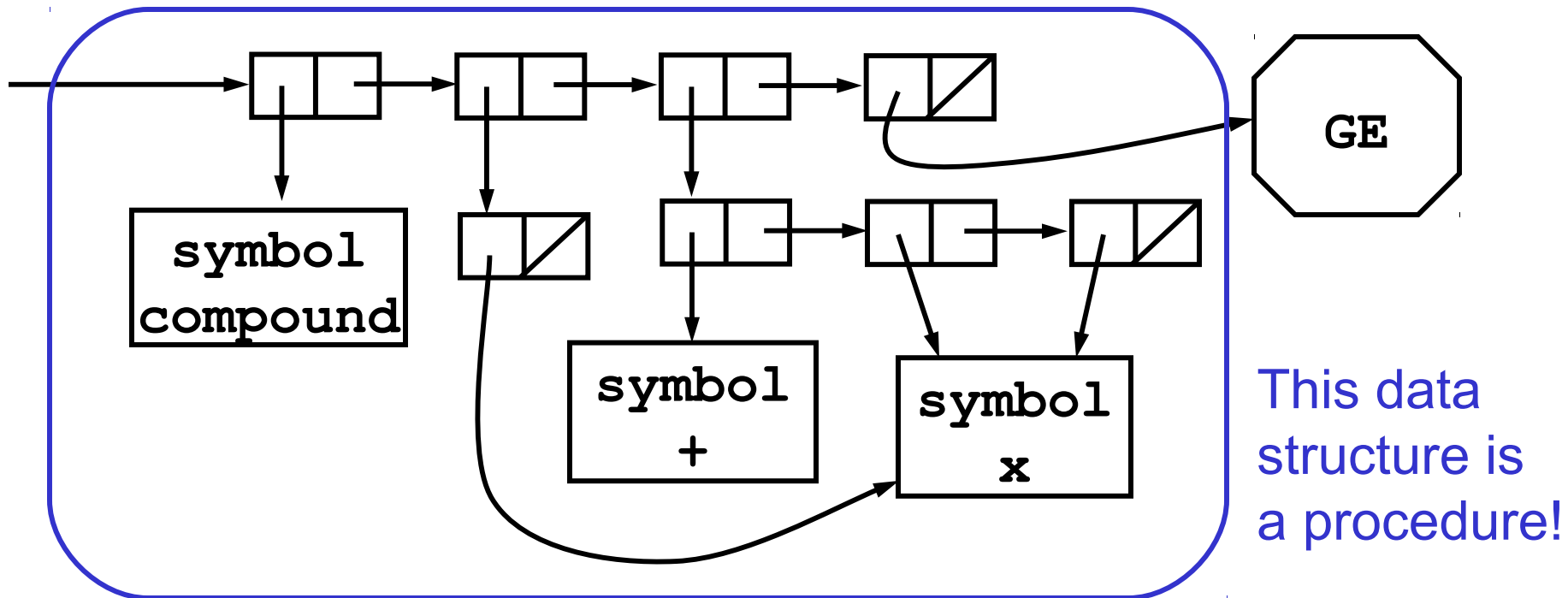
```
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if 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)
         (m-apply (m-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

Implementation of lambda

```
(eval '(lambda (x) (+ x x)) GE)
```

```
(make-procedure '(x) '(+ x x) GE)
```

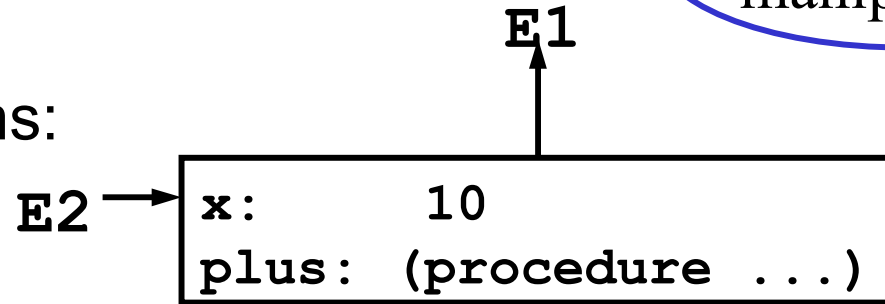
```
(list 'compound '(x) '(+ x x) GE)
```



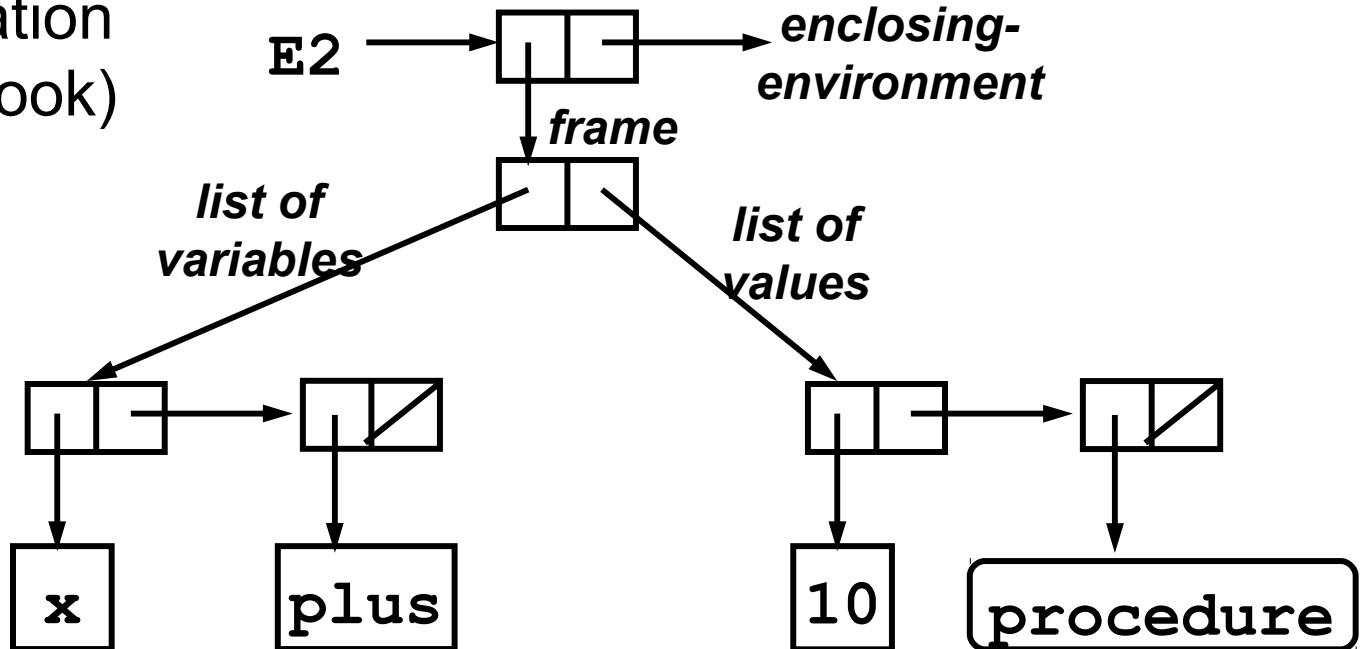
How the Environment Works

3. environment manipulation

- *Abstractly* – in our environment diagrams:



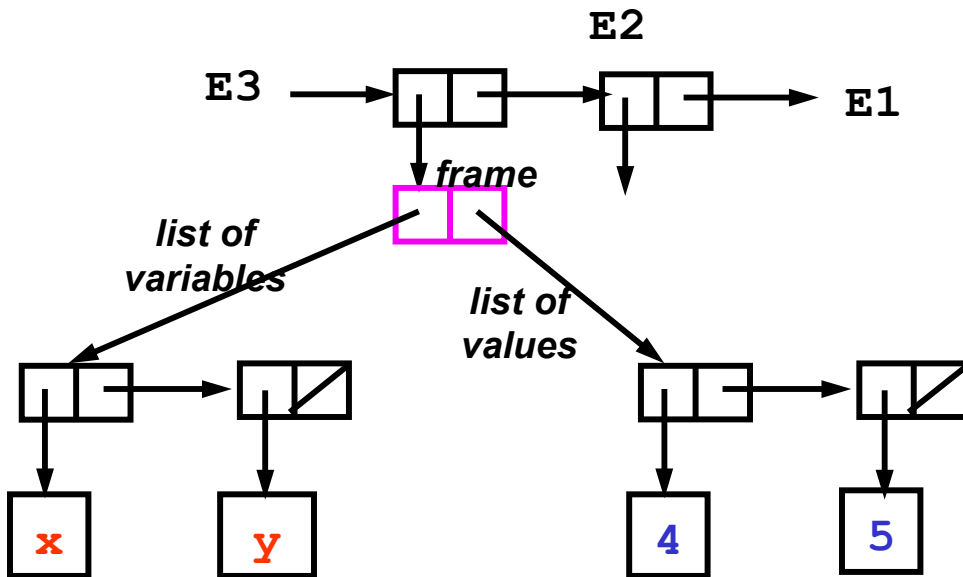
- *Concretely* – our implementation (as in textbook)



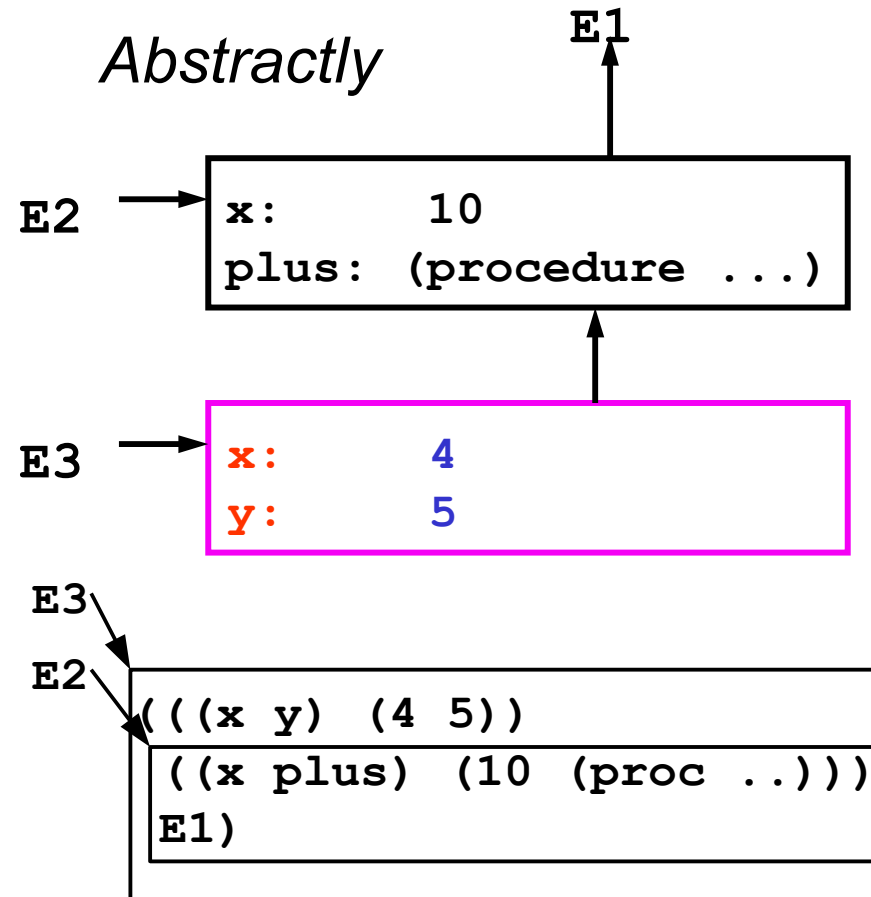
Extending the Environment

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

Concretely



Abstractly



"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** (i.e. we reached end of list of vars), 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. primitives and initial env.


- setup-environment

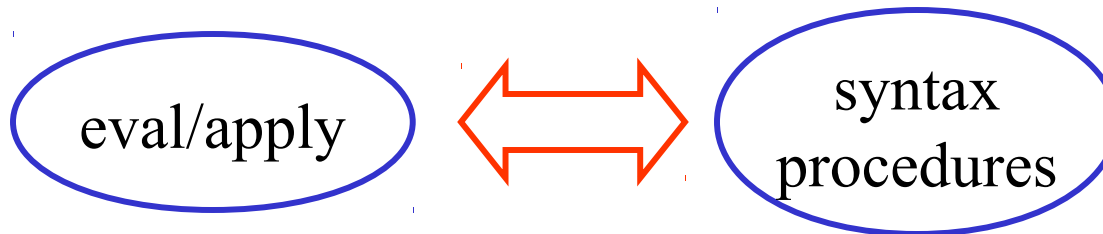
```
(define (setup-environment)
  (let ((initial-env (extend-environment
                      (primitive-procedure-names)
                      (primitive-procedure-objects)
                      the-empty-environment)))
    (define-variable! 'true #T initial-env)
    (define-variable! 'false #F initial-env)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

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

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

2. syntax
procedures



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))
```

Example – Changing Syntax

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

```
(<proc> <arg1> <arg2> . . .)
```

use

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

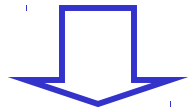
- Changes – **only in the syntax routines!**

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

Implementing "Syntactic Sugar"

- Idea:
 - Easy way to add alternative/convenient syntax
 - Allows us to implement a simpler "core" in the evaluator, and support the alternative syntax by translating it into core syntax
- "let" as sugared procedure application:

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



```
((lambda (<name1> <name2>)
  <body>)
  <val1> <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)))
```

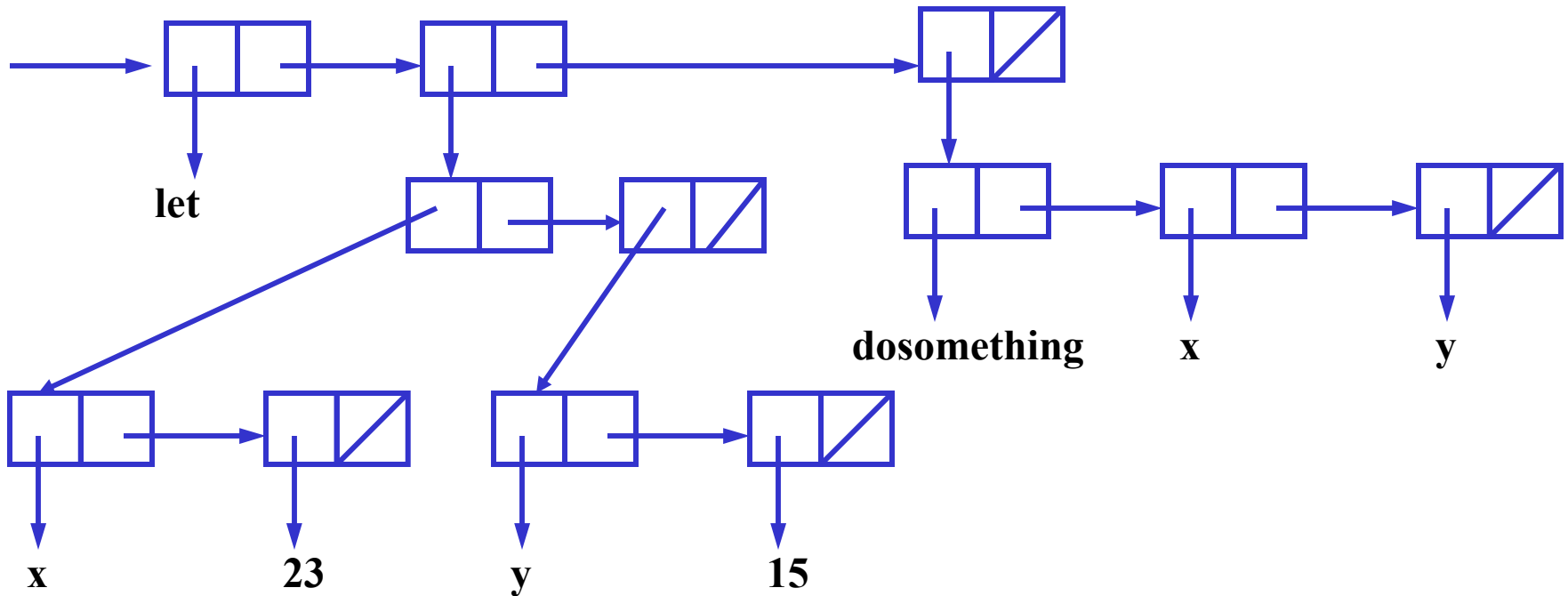
```
(define (let-body let-exp)  
  (cddr let-exp))
```

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

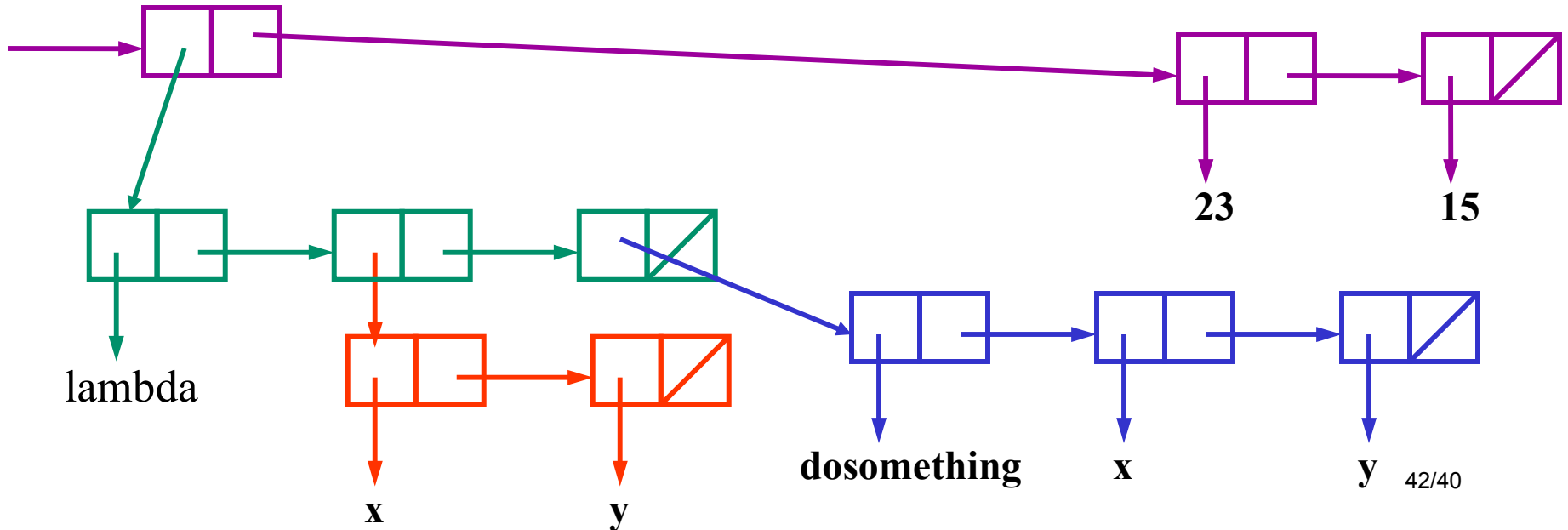
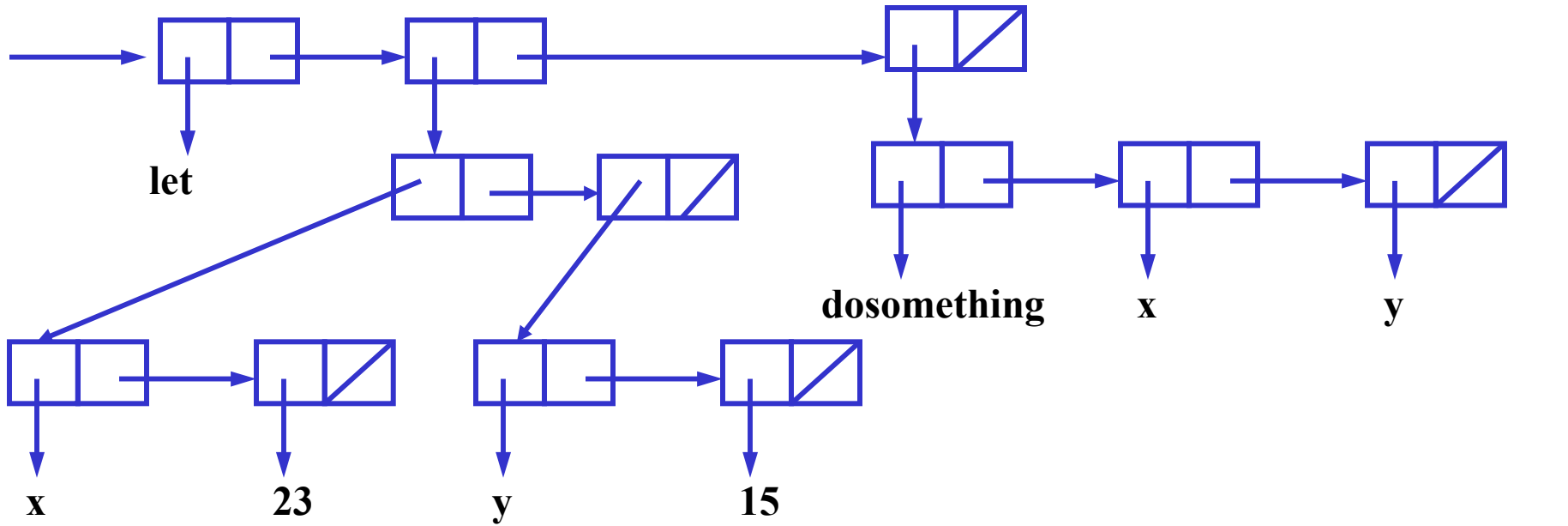
NOTE: only manipulates list structure, returning new list structure that acts as an expression

Details of let syntax transformation

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



Details of let syntax transformation



Defining Procedures

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

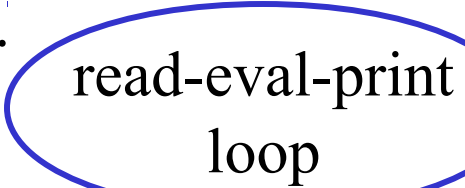
- Semantic implementation – just another define:

```
(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  
                   (caddr exp))) ;body
```

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)
      (display output)))
    (driver-loop))
```

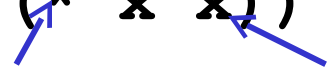
Variations on a Scheme

- More (not-so) stupid syntactic tricks
 - Let with sequencing
`(let* ((x 4)
 (y (+ x 1))) . . .)`
 - Infix notation
`((4 * 3) + 7)` instead of `(+ (* 4 3) 7)`
- Semantic variations
 - *Lexical vs dynamic* scoping
 - Lexical: defined by the program text
 - Dynamic: defined by the runtime behavior

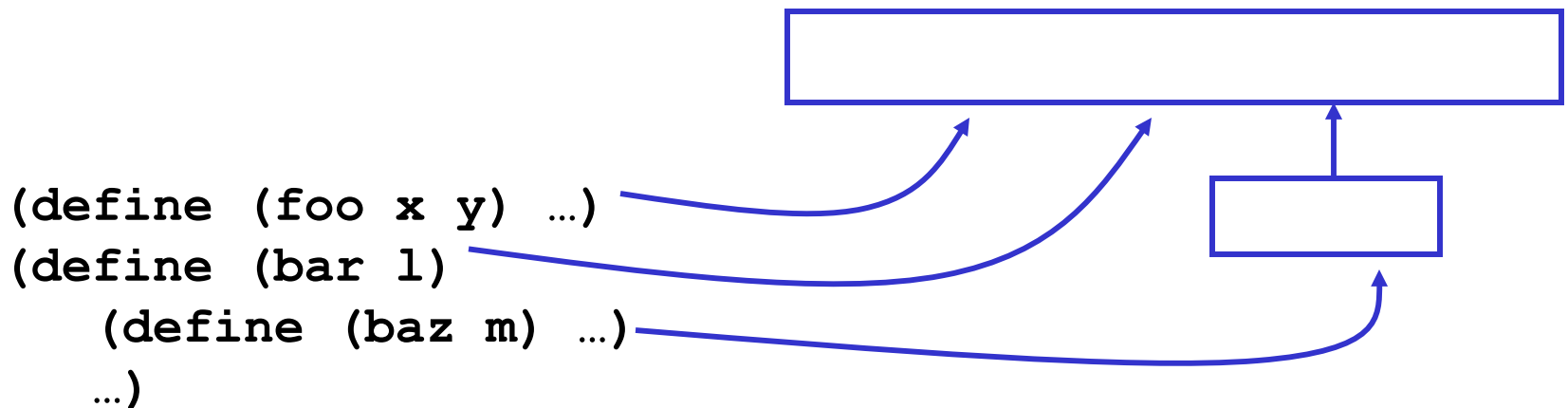
Diving in Deeper: Lexical Scope

- Scoping is about how **free variables** are looked up (as opposed to bound parameters)

`(lambda (x) (* x x))`
* is free x is bound

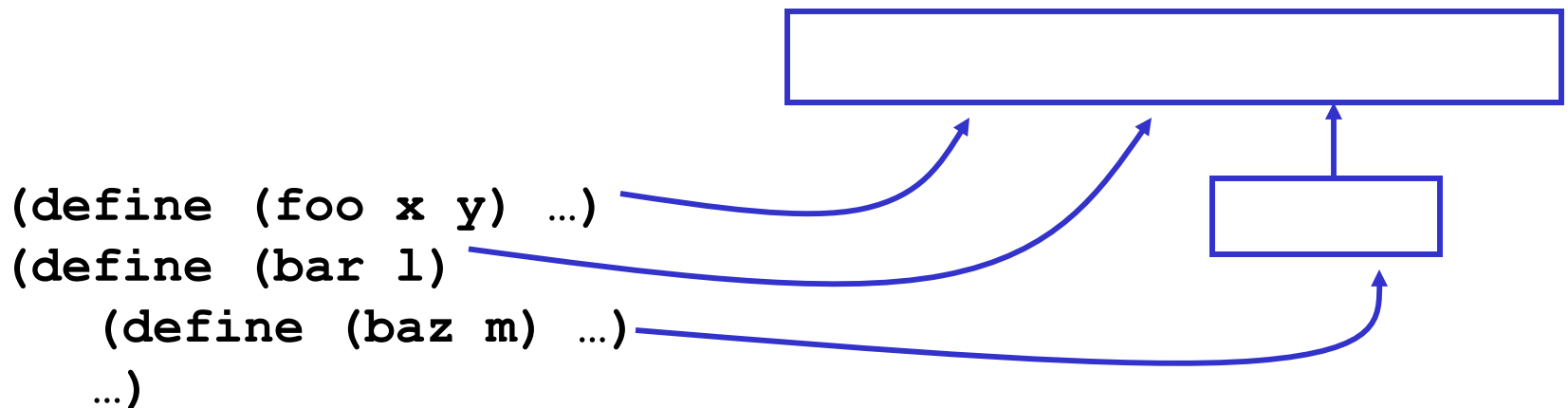


- How does our evaluator achieve lexical scoping?
 - environment chaining
 - procedures capture their enclosing **lexical** environment



Diving in Deeper: Lexical Scope

- Why is our language lexically scoped? Because of the semantic rules we use for procedure application:
 - “Drop a new frame”
 - “Bind parameters to actual args in the new frame”
 - “Link frame to the **environment in which the procedure was defined**” (i.e., the environment surrounding the procedure in the program text)
 - “Evaluate body in this new environment”



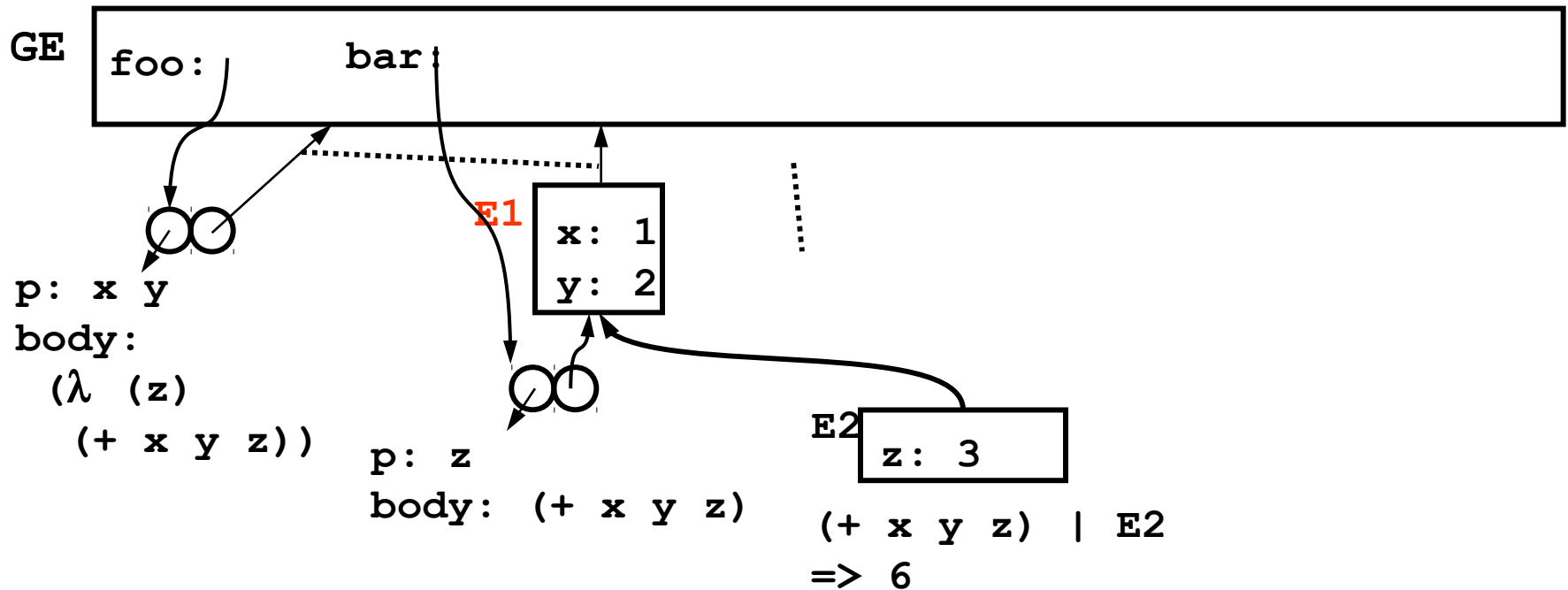
Lexical Scope & Environment Diagram

```
(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.



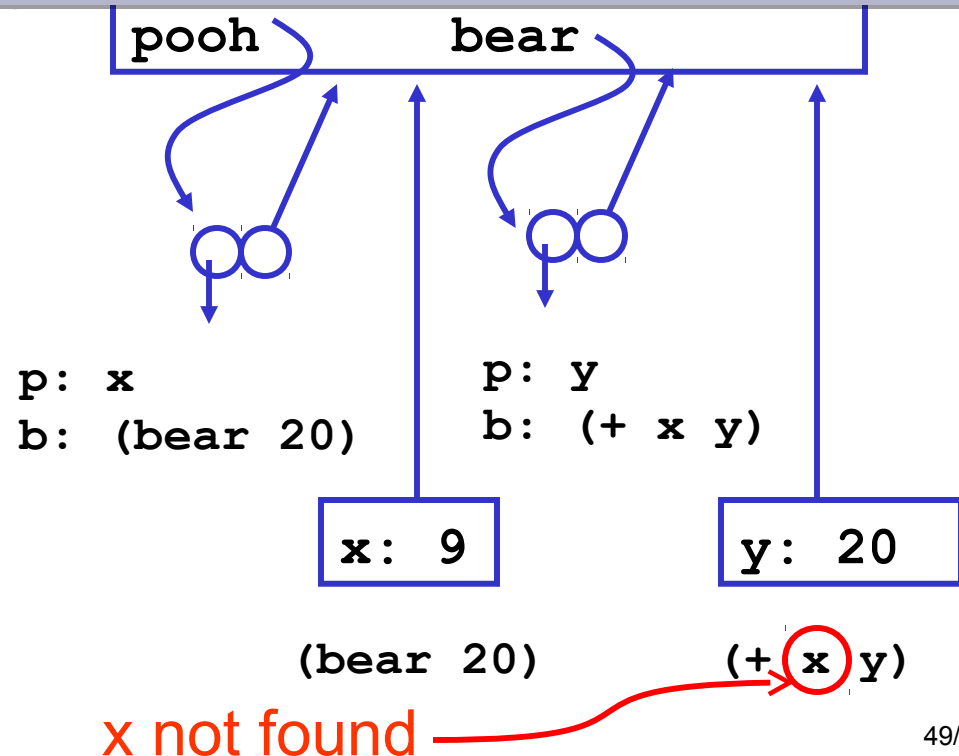
Alternative Model: Dynamic Scoping

- 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)
```

Suppose we use our usual environment model rules...



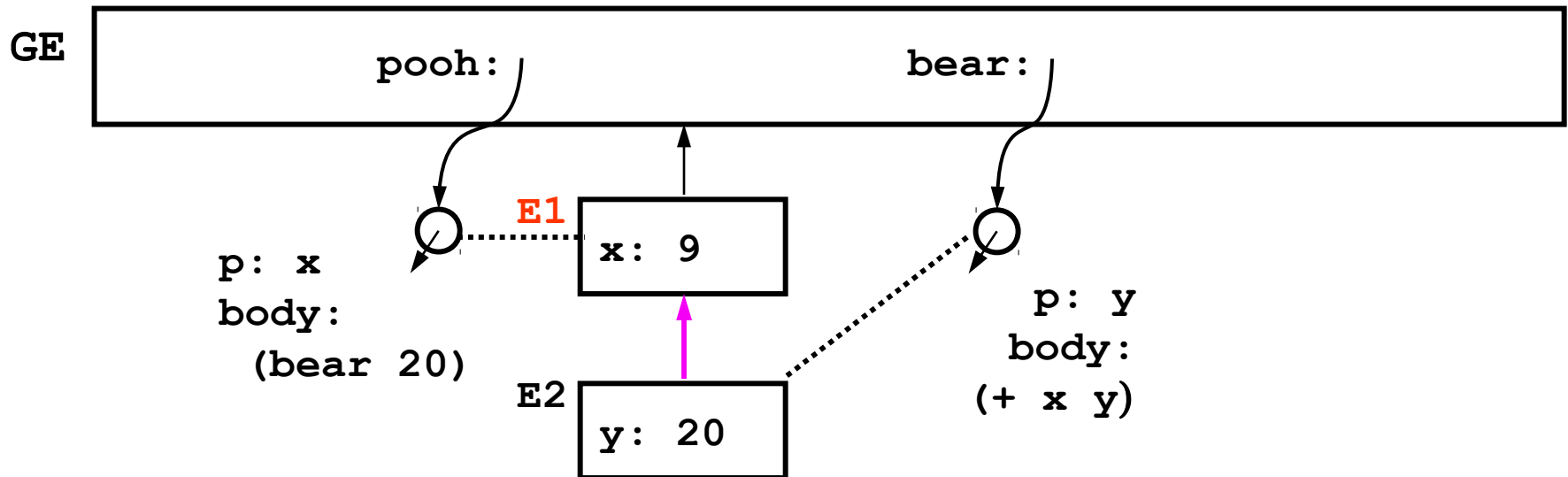
Dynamic Scope & Environment Diagram

```
(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**.



```
(+ x y) | E2  
=> 29
```

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))))
```

Evaluator 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