6.001 SICP
Environment Model

• A model for computation consistent with mutation
  – tells us where variable bindings live
  – tells us where bindings are changed

• A graphical model for how Scheme works
  – shows how lexical scoping (or block structure) is achieved

• A means to create and manipulate procedures with local state
Need for a New Model of Computation

- **Functional Programming** (up to now)
  - Every expression (almost) has a value
  - Procedures capture a *mapping* from values to values
    
    ```scheme
    (define (square x) (* x x)) ; number -> number
    ```
  - **Substitution Model** – expansions (by way of procedure applications) and reductions of expressions
    
    ```scheme
    (square 5)  
    ==> (* 5 5)  
    ==> 25
    ```

- **Imperative Programming** (with mutation)
  - Expressions can "do" something
  - have side effects
    ```scheme
    (define x 10)  
    x ==> 10
    ```
    ```scheme
    (set! x 20)  
    x ==> 20
    ```
    - different values, depends on WHEN evaluated!
Need for a New Model of Computation

• **Functional Programming** (up to now)
  – Every expression (almost) has a value
  – Procedures capture a *mapping* from values to values
    
    (define (square x) (* x x)) ; number → number
  – **Substitution Model** – expansions (by way of procedure applications) and reductions of expressions
    
    (square 5)
    => (* 5 5)
    => 25

• **Imperative Programming** (with *mutation*)
  – Expressions can “do” something – have side effects
    
    (define x 10)
    x => 10
    (set! x 20) – *changes* something...
    x => 20 – different values, depends on WHEN evaluated!
What the environment model is:

• A precise, completely mechanical description of:
  – name-rule  looking up the value of a variable
  – define-rule creating a new definition of a var
  – set!-rule  changing the value of a variable
  – lambda-rule creating a procedure
  – application applying a procedure

• Enables analysis of procedures with local/mutable state:
  – Example: make-counter

• Basis for implementing a scheme interpreter
  – for now: draw EM state with frames and pointers
  – later on: implement with code
Frame: a table of bindings

• Binding: a pairing of a name and a value

Example:  \( \text{x} \) is bound to 15 in frame A
\( \text{y} \) is bound to \((1 \ 2)\) in frame A

the value of the variable \( \text{x} \) in frame A is 15

![Diagram of frame A with bindings x: 15, y: (1 2)]
Environment

- Generally, an **environment** is a **sequence** of frames
  - Simplest example: the global environment (GE)
- All evaluation occurs **with respect to an environment**
  - Notation: `<exp> | <env>`

```
(define z 10) | GE ==> unspecified (side effect!)
z | GE ==> 10
(set! z 'apple) | GE ==> unspecified (side effect!)
z | GE ==> apple
```
Environment as a sequence of frames

- Environment E1 consists of frame B only
- Environment E2 consists of frames A and B
  - A frame may be shared by multiple environments

![Diagram showing environments and frames](image)
Environments & Lexical Scope (Block Structure)

```lisp
(let ((z 10))
  (let ((x 15)
        (y '(1 2)))
      …)
```

- **Local Variables**: `z`, `x`, `y`
- **Enclosing Environment**: `E1`
- **Enclosing Scope**: `E2`
- **Local Frame**: “inside” the enclosing environment

- **Environment `E1`**:
  - `z: 10`

- **Environment `E2`**:
  - `x: 15`
  - `y:`
Name-rule

• A name X evaluated in environment E gives the value of X in the first frame of E where X is bound.

\[
\begin{align*}
  z &|_{GE} \rightarrow 10 \\
  z &|_{E1} \rightarrow 10 \\
  x &|_{GE} \rightarrow 3
\end{align*}
\]

• In E1, the binding of x in frame A shadows the binding of x in B.

\[
\begin{align*}
  x &|_{E1} \rightarrow 15
\end{align*}
\]

This is a local frame.
Define-rule

• A define special form evaluated in environment $E$ creates or replaces a binding in the first frame of $E$

$$(\text{define } z \ 20) \ \mid \ GE \quad (\text{define } z \ 25) \ \mid \ E_1$$
Set!-rule

- A set! of variable $X$ evaluated in environment $E$ changes the binding of $X$ in the first frame of $E$ where $X$ is bound.

$(\text{set! } z \ 20) \mid_{GE}$

$(\text{set! } z \ x) \mid_{E_1}$

$x \mid_{E_1} \Rightarrow 15$
Your turn: evaluate the following in order

\[
\begin{align*}
(\text{+ } z \; 1) & \quad | \quad \text{E}_1 \\
\text{(set! } z \; (+ \; z \; 1)) & \quad | \quad \text{E}_1 \\
\text{(define } z \; (+ \; z \; 1)) & \quad | \quad \text{E}_1 \\
\text{(set! } y \; (+ \; z \; 1)) & \quad | \quad \text{GE} \\
\end{align*}
\]

\[\Rightarrow 11\]

(modify EM)

(modify EM)

(modify EM)

Error: unbound variable: y
Double bubble: how to draw a procedure

\[(\text{lambda } (x) \ (* \ x \ x))\]

A compound proc that squares its argument

Environment pointer

Code pointer

parameters: \( x \)
body: \( (* \ x \ x) \)
Lambda-rule

• A lambda special form evaluated in environment E creates a procedure whose environment pointer is E

\[
\text{(define square (lambda (x) (* x x)))} \mid _E^1
\]

Evaluating a lambda actually returns a pointer to the procedure object because the lambda was evaluated in E1

Environment pointer points to frame A

Parameters: x

Body: (* x x)
To apply a compound procedure P to arguments:

1. Create a new frame A
2. Make A into an environment E:
   A's enclosing environment pointer goes to the same frame as the environment pointer of P
3. In A, bind the parameters of P to the argument values
4. Evaluate the body of P with E as the current environment

You must memorize these four steps
\( (\text{square } 4) \mid \text{GE} \)

\[ \begin{align*}
\text{square} & \mid \text{GE} \Rightarrow \# [\text{proc}] \\
(* x x) & \mid \text{E1} \Rightarrow 16 \\
* & \mid \text{E1} \Rightarrow \# [\text{prim}] \\
x & \mid \text{E1} \Rightarrow 4
\end{align*} \]
Example: inc-square

\[
\begin{align*}
\text{inc-square:} & \quad \text{square:} \\
p: & \quad x \\
b: & \quad (+ 1 (\text{square} \ y)) \\
p: & \quad y \\
b: & \quad (* x x)
\end{align*}
\]

\[
\begin{align*}
\text{(define square} & \quad \text{(lambda} \ (x) \ (* x x))) \quad | \quad \text{GE} \\
\text{(define inc-square} & \quad \text{(lambda} \ (y) \ (+ 1 (\text{square} \ y)))) \quad | \quad \text{GE}
\end{align*}
\]
Example cont'd: \((\text{inc-square } 4) \mid \text{GE}\)

\[
\text{inc-square:}
\]

\[
\text{square:}
\]

\[
\text{GE} \quad \rightarrow \quad \text{inc-square:}
\]

\[
\text{GE} \quad \rightarrow \quad \text{square:}
\]

\[
p: \ x
\]
\[
b: \ (\ast \ x \ x)
\]

\[
p: \ y
\]
\[
b: \ (+ \ 1 \ (\text{square } y))
\]

\[
\text{inc-square} \mid \text{GE} \Rightarrow \#[\text{compound-proc} \ldots]
\]

\[
(+ \ 1 \ (\text{square } y)) \mid \text{E1}
\]

\[
+ \mid \text{E1} \Rightarrow \#[\text{prim}] \quad (\text{square } y) \mid \text{E1}
\]
Example cont'd: \((\text{square } y) \mid _E1\)

\[
\begin{align*}
\text{inc-square:} \\
\text{square:} \\
\end{align*}
\]

\[
\begin{align*}
p & : x \\
b & : (* x x) \\
\end{align*}
\]

\[
\begin{align*}
p & : y \\
b & : (+ 1 (\text{square } y)) \\
\end{align*}
\]

\[
\begin{align*}
\text{square} & \mid _E1 ==> #[\text{compound}] \\
y & \mid _E1 ==> 4 \\
(* x x) & \mid _E2 ==> 16 \\
(+ 1 16) & \mid _E1 ==> 17 \\
* & \mid _E2 ==> #[\text{prim}] \\
x & \mid _E2 ==> 4
\end{align*}
\]

3/18/2004
Lessons from the `inc-square` example

• Environment model (EM) doesn't show the complete state of the interpreter
  – missing the stack of pending operations

• The GE contains all standard bindings (`*`, `cons`, etc)
  – usually omitted from EM drawings

• Useful to link environment pointer of each frame to the procedure that created it
  – reminds us where that frame came from, and what next steps are… binding args and then evaluating proc body
Lexical Scoping and the EM – Key Ideas

• Local environments
  – “Inside” other environments in code text
  – Local frames pointing to enclosing environment

• Procedures remember their environments!
  – What matters is the surrounding environment at procedure creation time,
    • which will be the surrounding lexical environment,
  – NOT the environment that the procedure finally gets applied in

  – **Benefit**: if you can view/read the code, then you always know where the variable values are to be found
Lexical Scoping Example – sqrt

(define sqrt
  (lambda (x)
    (define good-enough?
      (lambda (guess)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve
      (lambda (guess)
        (average guess (/ x guess))))
    (define sqrt-iter
      (lambda (guess)
        (if (good-enough? guess)
            guess
            (sqrt-iter (improve guess))))))
(sqrt-iter 1))
sqrt Example

(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1))
sqrt Example

p: x
b: (define (good-enough? guess) …)
   (define (improve guess) …)
   (define (sqrt-iter? guess) …)
   (sqrt-iter 1)

E1 x: 2
good-enough?:
   improve:
sqrt-iter?:

E3 guess: 1
   (< …) |E3

E2 guess: 1
   (if (good…) …) |E2

E4 guess: 1
   (average …) |E4

(sqrt 2) |GE

(sqrt-iter 1) |E1
Environment Model

• A model for computation consistent with mutation
  – tells us where variable bindings live
  – tells us where bindings are changed

• A graphical model for how Scheme works
  – shows how lexical scoping (or block structure) is achieved

• A means to create and manipulate local state
Example: make-counter

• Counter: something which counts up from a number

(define make-counter
  (lambda (n)
    (lambda () (set! n (+ n 1))
      n ))))

(define ca (make-counter 0))
(ca) ==> 1
(ca) ==> 2
(define cb (make-counter 0))
(cb) ==> 1
(ca) ==> 3
(cb) ==> 2 ; ca and cb are independent
(define (make-counter n)
  (set! n (+ n 1))
  n)

(make-counter:)
(define ca (make-counter 0)) |GE

GE

make-counter:
ca:

p: n
b: (lambda ()
  (set! n (+ n 1))
  n)

E1

n: 0

E1

p: (set! n (+ n 1)) n

(lambda () (set! n (+ n 1)) n) |E1
(set! n (+ n 1)) ==\rightarrow 1

E1

E2

empty

1

(make-counter)

ca:

p: n

b: (lambda () (set! n (+ n 1)) n)

GE

3/18/2004
E3

\[ \text{make-counter:} \]

\[ \begin{align*}
\text{ca:} & \quad \text{GE} \\
\text{p: } & \quad n \\
\text{b:} & \quad (\lambda () \ (\text{set! } n \ (+ n 1)) \ n)
\end{align*} \]

\[ \begin{align*}
\text{p: } & \quad (\text{set! } n \ (+ n 1)) \ n \\
\text{empty} & \quad \text{GE}
\end{align*} \]
(define cb (make-counter 0))  |  GE

GE

make-counter:
ca:

GE

p: n
b: (lambda ()
  (set! n (+ n 1))
  n)

p: n
b: (set! n (+ n 1)) n

(lambda () (set! n (+ n 1)) n)  |  E4

3/18/2004
make-counter:

| p: n |
| b: (lambda () (set! n (+ n 1)) n) |

GE

(cba) | GE => 1

E1

n: 2

E4

n: 0

E5

p: n

b: (set! n (+ n 1)) n

3/18/2004
Capturing state in local frames & procedures

make-counter:
b: (lambda ()
  (set! n (+ n 1))
  n)

ca:
p: n
b: (lambda ()
  (set! n (+ n 1))
  n)

cb:
p: n
b: (set! n (+ n 1))

E1: n: 2

E4: n: 1

3/18/2004
Lessons Learned

• Environment diagrams get complicated very quickly
  – graphical tool to explain and reason using the environment model

• Environment Model:
  – implements block structure (lexical scoping)
  – shows where variables (bindings) are located
  – shows which values change as a result of mutation

• Implement objects with local state
  – a lambda captures the frame that was active when the lambda was evaluated
  – information hiding – expressions outside the environment do not have access to that local state
  – with environment model, see where local state changes