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
    ```
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
    
    (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: $x$ is bound to 15 in frame A
$y$ is bound to (1 2) in frame A

the value of the variable $x$ in frame A is 15

![Diagram of frame A with bindings x: 15 and 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: \(<\text{exp}>\mid_{\text{<env>}}\)

\[
\begin{array}{c}
\text{GE} \\
\rightarrow z: 10 \quad \text{apple}
\end{array}
\]

\[(\text{define } z \ 10) \mid_{\text{GE}} \Rightarrow \text{unspecified} \quad \text{(side effect!)}\]
\[z \mid_{\text{GE}} \Rightarrow 10\]
\[(\text{set! } z \ '\text{apple}) \mid_{\text{GE}} \Rightarrow \text{unspecified} \quad \text{(side effect!)}\]
\[z \mid_{\text{GE}} \Rightarrow \text{apple}\]
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 E1 and E2 with frames A and B, and the enclosing environment pointer.]

This arrow is called the enclosing environment pointer.
Environments & Lexical Scope (Block Structure)

(let ((z 10))
  (let ((x 15)
        (y '(1 2)))
    ...)

local variables

enclosing environment

enclosing scope

local frame ("inside" the enclosing environment)

E1

B

E2

A

z: 10

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.

  \[
  z \mid_{GE} \Rightarrow 10 \quad z \mid_{E1} \Rightarrow 10 \quad x \mid_{GE} \Rightarrow 3
  \]

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

  \[
  x \mid_{E1} \Rightarrow 15
  \]

  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 $E$

$$\text{(define } z \text{ 20)} \mid \text{GE} \quad \text{(define } z \text{ 25)} \mid \text{E}_1$$

Diagram:

- Environment $B$: $z: 10 \quad x: 3 \quad z: 20$
- Environment $A$: $x: 15 \quad y: \quad z: 25$
- Environment $E_1$: $1$ and $2$
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.

\[
(\text{set! } z \ 20) \mid_{GE} \quad (\text{set! } z \ x) \mid_{E_1}
\]

\[
(\text{set! } z \ x) \mid_{E_1} \Rightarrow 15
\]
Your turn: evaluate the following in order

\[
\begin{align*}
(+ \ z \ 1) & \quad |_{E_1} \\
(set! \ z \ (+ \ z \ 1)) & \quad |_{E_1} \\
(define \ z \ (+ \ z \ 1)) & \quad |_{E_1} \\
(set! \ y \ (+ \ z \ 1)) & \quad |_{G_E}
\end{align*}
\]

\[\Rightarrow \quad 11\]

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

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

A compound proc that squares its argument

Environment pointer

Lambda-rule

Parameters: \(x\)

Body: \((\ast \ x \ x)\)

eval
text
Lambda-rule

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

```
(define square (lambda (x) (* x x))) | E_1
```

A lambda special form evaluated in environment $E$ creates a procedure whose environment pointer is $E$. Evaluating a lambda actually returns a pointer to the procedure object. Environment pointer points to frame $A$ because the lambda was evaluated in $E_1$. 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
(square 4) | GE

GE →

x: 10
square:
parameters: x
body: (* x x)

*: #[prim]

square | GE ==> #[proc]
(* x x) | E1 ==> 16
* | E1 ==> #[prim] x | E1 ==> 4
Example: inc-square

```
(define square (lambda (x) (* x x)))  | GE
(define inc-square
  (lambda (y) (+ 1 (square y))))  | GE
```

inc-square:

```
square:
```

p: x
b: (* x x)

p: y
b: (+ 1 (square y))
Example cont'd: \((\text{inc-square } 4) \mid \text{GE}\)

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

inc-square: square:

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

\[
\begin{align*}
\text{inc-square} \mid \text{GE} & \implies \# [\text{compound-proc} \ldots] \\
(+ 1 (\text{square } y)) \mid \text{E}_1 & \implies \# [\text{prim}] (\text{square } y) \mid \text{E}_1
\end{align*}
\]
Example cont'd: \((\text{square } y) \mid E_1\)

\[
\begin{align*}
\text{inc-square:} & \quad \text{square:} \\
\text{GE} & \\
p: x & \quad p: y \\
b:(*x\,x) & \quad b: (+ 1 (\text{square } y)) \\
\rightarrow & \\
\text{square} \mid E_1 & \rightarrow \#[\text{compound}] \\
(*x\,x) \mid E_2 & \rightarrow 16 \\
\rightarrow & \\
\ast \mid E_2 & \rightarrow \#[\text{prim}] \\
x \mid E_2 & \rightarrow 4 \\
y \mid E_1 & \rightarrow 4 \\
(+ 1\,16) \mid E_1 & \rightarrow 17
\end{align*}
\]
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 – $\text{sqrt}$

```scheme
(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

sqrt: 

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

(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)

(sqrt 2) |GE

(sqrt-iter 1) |E1

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

E2 guess: 1
    (if (good...) ...)|E2

E3 guess: 1
    (< ...) |E3

E4 guess: 1
    (average ...) |E4
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

\[
\text{(define make-counter}
   \text{  (lambda (n)}
   \text{      (lambda () (set! n (+ n 1))}
   \text{         n )}))
\]

\[
\text{(define ca (make-counter 0))}
\]
\[
\text{(ca) ==> 1}
\]
\[
\text{(ca) ==> 2}
\]

\[
\text{(define cb (make-counter 0))}
\]
\[
\text{(cb) ==> 1}
\]
\[
\text{(ca) ==> 3}
\]
\[
\text{(cb) ==> 2 ; ca and cb are independent}
\]
(define (make-counter n)
  (set! n (+ n 1))
  n)
(define ca (make-counter 0)) | \text{GE}

make-counter:
\text{ca}:

\text{p: n}
\text{b: (lambda () (set! n (+ n 1)) n)}

\text{p: b: (set! n (+ n 1)) n}

(lambda () (set! n (+ n 1)) n) | \text{E1}
(ca) | \( GE \Rightarrow 1 \)

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

\[
\text{ca:}
\]

\[
p: n
\]

\[
b: (\lambda () \ (\text{set! } n \ (+ \ n \ 1)) \ n)
\]

\[
(\text{set! } n \ (+ \ n \ 1)) \mid E2 \quad n \mid E2 \Rightarrow 1
\]

3/18/2004
E3

(set! n (+ n 1)) | E3 n | E3 ==> 2

(make-counter:
   ca:
   p: n
   b: (lambda ()
      (set! n (+ n 1))
      n)
   E1
   n: 0 1 2
   empty

GE

(ca) | GE ==> 2
(define cb (make-counter 0)) | GE

(make-counter:
  ca:
  cb:

GE

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

E1
n: 2
p:
b: (set! n (+ n 1)) n

E4
n: 0
p:
b: (set! n (+ n 1)) n

(lambda () (set! n (+ n 1)) n) | E4
\[
(c_b) \quad |_{E_G} \quad ==\Rightarrow 1
\]

\textbf{make-counter:}

\textbf{ca:}

\textbf{cb:}

\textbf{p: } n

\textbf{b: } (\textbf{lambda } ()

\textbf{(set! n (+ n 1)) n)}

\textbf{E1}

\textbf{n: } 2

\textbf{E4}

\textbf{n: } 0

\textbf{E5}

\textbf{p: } n

\textbf{b: } (\textbf{set! n (+ n 1)) n}
Capturing state in local frames & procedures

```scheme
(make-counter: 
  ca: 
    n: 1 
    p: n 
    b: (lambda () 
        (set! n (+ n 1)) 
        n)

  cb: 
    n: 2 
    p: n 
    b: (set! n (+ n 1)) 
    n)
```

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