Why do we need a new evaluation model?

- The substitution model broke down when we added side effects!
- In Object Oriented Programming, we need a model to represent hierarchies, shadowing, and inheritance.
- Looking Ahead: We’ll be writing a meta-circular evaluator in Scheme. You’ll see that the code we write closely relates to the rules of the environment model.
- It’s boring, but important: It’s a technical investment now, but you’ll get a lot out of it soon.

Diagram Components

- **Frames:** We draw a frame as a box. In the box go bindings. Every frame (except the frame representing the global environment) needs to have a link to exactly one other frame.
- **Bindings:** A binding is an association between an identifier (or symbol) to a value.
- **Procedure Objects:** These are special objects (symbolized by the double bubble) created by evaluating a lambda expression as described below.

Environment: a sequence of frames

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

Rules

- Looking up a name
- Define / Set
- Lambda
- Combination / Procedure application
Looking up an identifier

1. Look for a value in the current frame.
2. If there is no value for that identifier in the current frame, follow the link from the current frame to the one that it is linked from.
3. Continue until we find a binding for the identifier we're looking up or until we run out of frames in our chain of links. If there's no binding in the GE, the identifier we're looking up is an unbound variable.

set and define

Define (define var expression)
1. Evaluates the expression with respect to the current environment.
2. Adds a binding to the frame in which it is evaluated.

Lambda

- Evaluating a lambda expression will result in a two-part procedure object (two circles next to each other -- the double bubble).
- The pointer of the left circle points down to a list of the parameters and the body of the procedure.
- The pointer of the right circle points up to the environment frame in which the lambda expression was evaluated. Note that nothing else happens until we apply the procedure.

Procedure object: “double bubble”

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

Points to environment in which lambda was evaluated!

Combinations / Procedure application

To evaluate a combination with respect to an environment, first evaluate the subexpressions with respect to the environment and then apply the value of the operator subexpression to the values of the operand subexpressions.

Applying a procedure P
1. Draw a new environment frame.
2. Link the new environment frame to the environment frame pointed to by the right circle of the procedure object.
3. Bind the parameters to their values in this new environment frame.
4. Evaluate the body of the procedure in this new environment frame.
Example 1
Evaluate:

(define a 5)
(define foo
  (lambda (x) (+ a x)))
(foo 4)

Example 1
Here’s what happens:
1. We bind a to 5
2. A new frame is created because of the application. It is shown with a
dashed link to the procedure whose application creates the new
frame.
3. In the new frame we bind the procedure’s formal parameter x to the
actual argument 4.
4. The new frame has as its parent the environment pointer from the
procedure. Together then, the new frame and the enclosing
environment define a new environment (E1).
5. Now we evaluate the body of the procedure with respect to the new
environment (E1).
6. To evaluate the (+ a x) expression, we look up x and find it in the
local frame to get a value 4. We look up a and find that it is NOT in
the local frame. So we follow the frame parent link up to the
surrounding frame (the GE), where we successfully find a binding for
a whose value is 5. The addition finally returns 9.

Example 2
Evaluate:

(define (fact n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))
(define n 10)
(fact 3)

Example 2
Procedure application: “drop a frame”

(define a 5)
(define foo
  (lambda (x) (+ a x)))
(foo 4)

Example 2
1. Draw frame
2. Set enclosing env. ptr of frame to be env. ptr of proc.
3. Assign args to vals in new frame
4. Eval body w.r.t. new frame
let

Let just creates another frame linked from the current frame. The bindings in that frame are let-variables bound to the result of evaluating the let-expressions with respect to the original environment.

\[(\text{let } ((a \ b \ c) \ \text{body}) \rightarrow \ (\text{let } a = \text{evaluated-body}) \ \text{body}) \] 

\[(\text{let } ((x \ y) \ \text{body}) \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}) \] 

\[(\text{let } ((x \ y)) \ \text{body}) \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}) \] 

\[
\begin{align*}
\text{let } (\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
\text{let } (\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
\text{let } (\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

\[
\begin{align*}
(\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

\[
\begin{align*}
(\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

\[
\begin{align*}
(\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

\[
\begin{align*}
(\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

\[
\begin{align*}
(\text{let } ((x \ y)) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body} \\
(\text{let } (\lambda \ x \ y) \ \text{body}) & \rightarrow \ (\text{let } x = \text{evaluated-body}) \ \text{body}
\end{align*}
\]

Example3

\[
(\text{define } \text{adder y})
(\lambda x ((\text{let } \text{inc = x}) \ \text{body})))
(\text{define add3 } (\text{adder 3}))
(\text{add3 4})
(\text{define } \text{adder y})
(\lambda x ((\text{let } \text{inc = x}) \ \text{body})))
(\text{define add3 } (\text{adder 3}))
(\text{add3 4})
\]

Example4

Let’s use this technique to build a one-element buffer...write a function that returns the argument in it’s previous call...

\[
(\text{define last-value } \text{"first-call"})
(\text{define } \text{prev new})
(\text{define temp last-value})
(\text{set! Last-value new})
\]

Example4

• Now, consider the following let statement. The effect of this statement in the environment is then shown.

\[
(\text{let } ((\text{last-value } \text{"first-call"}) \ (\text{define } \text{prev new}) \ ...))
\]

Example4

• Define instead a procedure \text{make-prev} that can be used to create the procedure \text{prev} and add it to the environment (to make completely clear the binding action of procedures)

\[
(\text{define } \text{make-prev last-value})
\]

\[
(\text{define } \text{prev new})
\]

\[
(\text{define temp last-value})
\]

\[
(\text{set! last-value new})
\]

\[
(\text{temp})
\]
Example 4

(define prev (make-prev "first-call"))
(prev 'a):

Block structure and Lexical Scoping

Example 5 - Corresponding Env. Diagram

What created this?

- (define (foo x) ...) w.r.t. GE;
  (foo 1) w.r.t. ANY ENVIRONMENT
- (define (foo)
  (define x 1)
  ...)

or this?

(define bar
  (let ((x 1))
    (lambda (...) ...)))

this?

(define (make-bar x)
  (lambda (...))
  (define bar1 (make-bar 1))

6.001 SICP
Example 6

Evaluate:

(define make-count-proc-1
  (lambda (f)
    (lambda (x)
      (let ((count 0))
        (cond ((eq? x 'count) count)
              (else (set! count (+ count 1))
                    (f x)))))))

(define sqrt-c-1 (make-count-proc-1 sqrt))

(sqrt-c-1 4) ==> ?
(sqrt-c-1 'count) ==> ?