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.

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

Environment: a sequence of frames
- Environment $E_1$ consists of frames A and B
- Environment $E_2$ 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.

set and define

Set! (set! var expression)
1. Evaluates the expression with respect to the current environment.
2. Lookup the identifier in the current environment
3. Rebind the identifier in the frame it was found to the value of the expression.
4. If identifier is not found, unbound variable error

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”

(lambda (x) (* x x))

<table>
<thead>
<tr>
<th>Environment pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code pointer</td>
</tr>
<tr>
<td>parameters: x</td>
</tr>
<tr>
<td>body: (* x x)</td>
</tr>
</tbody>
</table>

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. 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
Evaluate:

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

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.

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

---

**Example 2**

Evaluate:

```
(define (fact n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))
```

```
(define n 10)
(fact 3)
```

---

**Procedure application: “drop a frame”**

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

```
(define (fact n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))
```

```
(define n 10)
(fact 3)
```
Example 2

\[
\text{fact: n: 10}
\]

\[
\text{p: n} \text{ n: 3 n: 2 n: 1}
\]

\[
p: (\text{if} (= 1 \text{n}) 1 (* (\text{fact} (- \text{n} 1))))
\]

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.

\[
(\text{let ((n v) …)} \text{body}) \rightarrow ((\lambda (\text{n v) …)} \text{body}) \text{n v})
\]

\[
(\text{let ((k (+ 2 5)) y 7) (* x y)})
\]

\[
(\lambda (\text{x y}) (* \text{x y}) (+ 2 5)) \text{7}
\]

Example 3

\[
(\text{define (adder y)}
(\text{let ((\text{inc y}) (\lambda (\text{x}) (+ \text{y x})))}
(\lambda (\text{inc x}))
(\lambda (\text{x y}) (* \text{x y}))
(adder 3) 4)
\]

\[
(\text{define (adder y)
(\lambda (\text{inc x})
(\lambda (\text{x y}) (* \text{x y}))
(adder 3) 4)
\]

\[
(\text{define (adder y)
(\lambda (\text{inc})
(\lambda (\text{x y}) (* \text{x y}))
(adder 3) 4)
\]

\[
(\text{define (adder y)
(\lambda (\text{inc})
(\lambda (\text{x y}) (* \text{x y}))
(adder 3) 4)
\]

Example 4 - Corresponding Env. Diagram

Block structure and Lexical Scoping

[Diagram of block structure and lexical scoping]