MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Friday, March 19

1. Environment Diagrams

Why are we doing this?

- 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.

What are environment diagrams made of?

- **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.

What are the rules again?

1. **Combination**

   To evaluate a comibination 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.
2. **Looking up an identifier**

   Look for a value in the current frame.

   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.

   Continue in this manner until we find a binding for the identifier we're looking up or until we run out of frames in our chain of links (i.e. we come to the global environment). If there's no binding in the global environment, the identifier we're looking up is an unbound variable.
3. **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.

4. **Define**

   Define adds a binding to the frame in which it is evaluated.

5. **Applying a procedure**

   Draw a new environment frame. Bind the parameters to their values in this new environment frame.

   Link the new environment frame to the environment frame pointed to **by the right circle of the procedure object.**

   Evaluate the body of the procedure in this new environment frame.

6. **Set!** `(set! var expression)`

   Evaluate the expression with respect to the current environment.

   Lookup the identifier in the current environment (see step #2)

   Rebind the identifier in the frame it was found to the value of the expression.

7. **Let**

   Just de-sugar! (See examples, below)

2. A Simple Example

Evaluate the expressions below, following the rules of the environment model. Draw the cooresponding environment diagram in the box.

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

(define n 6)
(fact 3)
```

3. Let

We haven't defined the rule for `let`. Well, let's just desugar it:

```
(let ((x (+ 2 5))
      (y 7))
  (* x y))
     ⇐⇒
((lambda (x y) (* x y))
  (+ 2 5) 7)
```

`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.

4. Another Example

Consider the example below using higher order procedures. Use the environment model to evaluate the following expressions.

```
(define make-counter
  (lambda ()
    (let ((count 0))
      (lambda ()
        (set! count (+ 1 count))
        count))))

(define days (make-counter))
(define dollars (make-counter))

(days)    ⇒ 1
(days)    ⇒ 2
(dollars) ⇒ 1
(dollars) ⇒ 2
```

4. More Environment Diagrams

```
(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)        ==>  2
(sqrt-c-1 'count)   ==>  0
```

```
(define make-count-proc-2
  (lambda (f)
    (let ((count 0))
      (lambda (x)
        (cond ((eq? x 'count) count)
              (else
               (set! count (+ count 1))
               (f x)))))))

(define sqrt-c-2
  (make-count-proc-2 sqrt))
(define sqr-c-2
  (make-count-proc-2 square))

(sqrt-c-2 4)        ==>   2
(sqrt-c-2 'count)   ==>   1

(sqr-c-2 4)         ==>  16
(sqr-c-2 'count)    ==>  1
```

```
(define make-count-proc-3
  (let ((count 0))
    (lambda (f)
      (lambda (x)
        (cond ((eq? x 'count) count)
              (else
               (set! count (+ count 1))
               (f x)))))))

(define sqrt-c-3
  (make-count-proc-3 sqrt))
(define sqr-c-3
  (make-count-proc-3 square))

(sqrt-c-3 4)        ==>  2
(sqrt-c-3 'count)   ==>  1

(sqr-c-2 4)         ==>  16
(sqr-c-2 'count)    ==>   2
```