Lecture 4. Part 1. More on "First-Class" Procedures

4.1 The Lambda Special Form

An unnamed "doubling" procedure:

```
(lambda (n) (* 2 n))
```

Think of this as saying "The procedure that takes an argument *n*, and, when called on that argument, will return the value of 2 times *n*."

If we evaluate this expression at the Scheme interpreter, we simply see the following:

```
>>> (lambda (n) (* 2 n))
#<PROCEDURE>
```

We can call an unnamed procedure on arguments by using it within a standard procedure call. Suppose, for instance, we evaluate the following expression at the Scheme interpreter:

```
>>> ((lambda (n) (* 2 n)) 3)
6
```

The rule for evaluating this expression is actually the very same that we have been using all along. Each of the two subexpressions in the expression is evaluated: the second is a number, the first is a lambda expression that returns a procedure. Then the procedure is applied to the (just-evaluated) argument values. Up until now, we have employed only procedure names as the first subexpression of a procedure call:

```
(double 3)
```

but there is no inconsistency in using any expression that returns a procedure.

Some other examples: a "cubing procedure", and an "octagon-drawing procedure":

```
(lambda (x) (expt x 3))
```

```
(lambda (side) (repeat 8 (fd side) (rt 45)))
```

4.2 Procedures that return new procedures

Procedures can, when called, return new procedures as their values:

```
(define (make-expt n)
  (lambda (x) (expt x n)))
```

What this procedure says, in prose: "Give me a number n, and I will return to you the 'x-to-the-n procedure'. (So, if you give me as an example the number 3, I will return to you a cubing procedure; if you give me 5, I will return to you the "x-to-the-fifth procedure"; and so forth.)"

```
(define (make-n-poly-procedure n)
  (lambda (side) (repeat n (fd side) (rt (/ 360 n)))))
```

What this procedure says, in prose: "Give me a number n, and I will return to you the 'n-fold-polygon-maker procedure.' So, if you give me the number 3 (as an example), I will return to you a triangle-making procedure; if you give me 6, I will return a hexagon-making procedure; and so on.

These two examples— `make-expt` and `make-n-poly-procedure` — take numbers as arguments, and return procedures as their results. Here's an example that takes a list as argument, and returns a procedure:

```
(define (make-random-selector list-of-possibilities)
  (lambda ()
    (list-ref list-of-possibilities
              (random (length list-of-possibilities)))))
```

What this procedure says, in prose: "Give me a list, and I will return to you a procedure of no arguments. This new procedure, when called on no arguments, will return an object chosen at random from the original list."

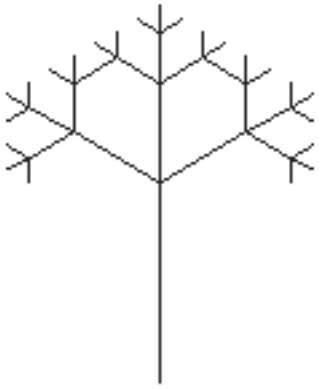
Here's an example of our new procedure in action:

```
>>> (define pick-a-number
      (make-random-selector '(1 2 3 4 5)))
pick-a-number
>>> (pick-a-number)
2
>>> (pick-a-number)
4
```

A more complicated turtle-graphics example:

We would like to write a procedure which, when called on a number n, will return a new n-fold branching tree procedure. Here's an example of how we would like our new procedure to work:

```
(define three-fold-branch (make-n-branch-procedure 3))
```



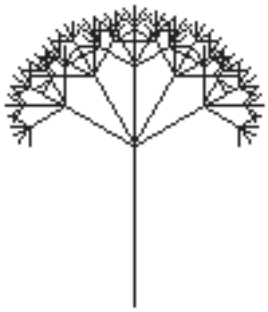
```
(three-fold-branch 50 4)
```

Here's how we could write `make-n-branch-procedure` (note that for this purpose, we write a total of two procedures):

```
(define (make-n-branch-procedure n)
  (lambda (side level)
    (n-fold-branch side level n)))

(define (n-fold-branch side level n)
  (cond ((= level 0) 0)
        (else (fd side)
              (lt 60)
              (n-fold-branch (/ side 2) (- level 1) n)
              (repeat (- n 1)
                      (rt (/ 120 (- n 1)))
                      (n-fold-branch (/ side 2) (- level 1) n))
              (lt 60)
              (bk side))))
```

When called on a number, `make-n-branch-procedure` returns a new procedure of two arguments (side and level). This new procedure can be used to create n-fold trees.



```
((make-n-branch-procedure 5) 40 4)
```

4.3 Procedures that return new procedures (continued)

Just now, we saw examples of procedures that take numbers or lists as arguments, and return new procedures as their results. Imagine that we have a table in which the rows denote the types of arguments a procedure can take, and the columns denote the types of results a procedure can return:

	Type of result		
Type of argument	Number	List	Procedure
Number	**double**	**list**	**make-expt**
List	**length**	**reverse**	**make-random-selector**
Procedure	**apply-to-3**	**map**	**??**

The entries in bold indicate examples of procedures that take (or are capable of taking) arguments in the given row, and returning results in the given column. Note that we are playing a little bit loose with a few of these examples: for instance, the list primitive need not take a number as its argument; the map primitive takes two arguments (one of which is a procedure); and the apply-to-3 procedure might not return a number (we might call apply-to-3 on the list primitive, for example, and thereby return a list). Finally, there are other rows and columns we haven't included here: booleans and symbols are other object types which could easily be used to extend this table.

The point of making this table is to suggest that there is nothing wrong with filling that bottom-right-hand slot: we could now write a procedure that takes some procedure as its argument, and returns a new procedure as its result. Here's a turtle-graphics example: this procedure takes as its argument a turtle-mover (a procedure of one side argument), and returns a new procedure

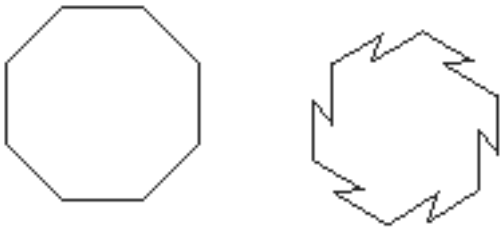
which, when called on arguments n and side, will cause the turtle-mover to be repeated n times (with appropriate turns in between):

```
(define (make-poly-procedure mover)
  (lambda (n side)
    (repeat n (mover side) (rt (/ 360 n)))))
```

Here are two procedures created with our general-purpose make-poly-procedure. The first can be used to make regular polygons (using fd as the mover); the second can be used to make "zig" polygons.

```
(define make-regular-polygon (make-poly-procedure fd))

(define make-zig-polygon (make-poly-procedure zig))
```



```
(make-regular-polygon 8 20)
(make-zig-polygon 6 15)
```

An example from mathematics: here is an approximation to the meaning of the derivative of a function F:

$$F'(x) = \frac{F(x+h) - F(x)}{h}$$

For this (approximate) definition we assume that h is a "sufficiently small value".

Here is a Scheme procedure named `derivative` that takes as its argument a numeric procedure and returns as its result a new numeric procedure approximating the derivative of the original:

```
(define (derivative f)
  (lambda (x)
    (/  (- (f (+ x 0.0001)) (f x))
        0.0001)))
```

Here, we have used 0.0001 as our "small value" h. Now, to test our derivative procedure, we first make a simple numeric procedure that cubes its argument:

```
>>> (define (cube x) (expt x 3))
cube
>>>  (cube 4)
64
```

The derivative of the cubing procedure is (approximately) a 3-x-squared procedure:

```
>>> ((derivative cube) 4)
48.00120000993502
```

We can call derivative on the 3-x-squared procedure to get (approximately) a 6-times-x procedure:

```
>>> ((derivative (derivative cube)) 4)
24.000598841666942
```

The environment model of the Scheme interpreter

(For a full explanation of this picture of the Scheme interpreter, see Chapter 11 in *Programming in MacScheme*.)

4.4 The rules for creating procedures and evaluating procedure calls

When a new procedure is created:

Create a two-part "procedure object" (drawn as two little circles in the text). One part of this object is linked to the text of the new procedure (its arguments and body); the other is linked to the environment in which the procedure was just created.

When a procedure-call expression is evaluated:

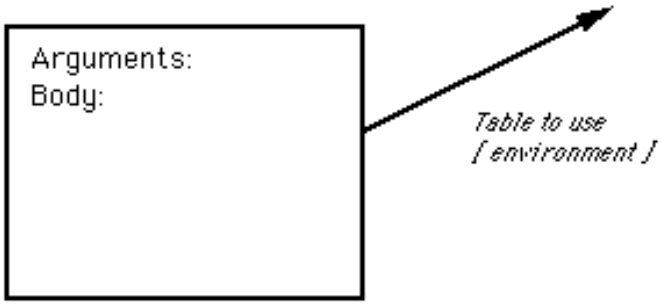
a. Evaluate each of the subexpressions (including, naturally, the argument subexpressions).

b. Create a new "frame" (or local table) in which the names of the procedure arguments are linked to the results of evaluating the argument subexpressions.

c. Link this new "frame" (or local table) to the table pointed to by the procedure object being applied.

d. Using the new "environment" (or composite table) created by step c, evaluate the body of the procedure and return the result of the last complete expression evaluated.

What a procedure object looks like:



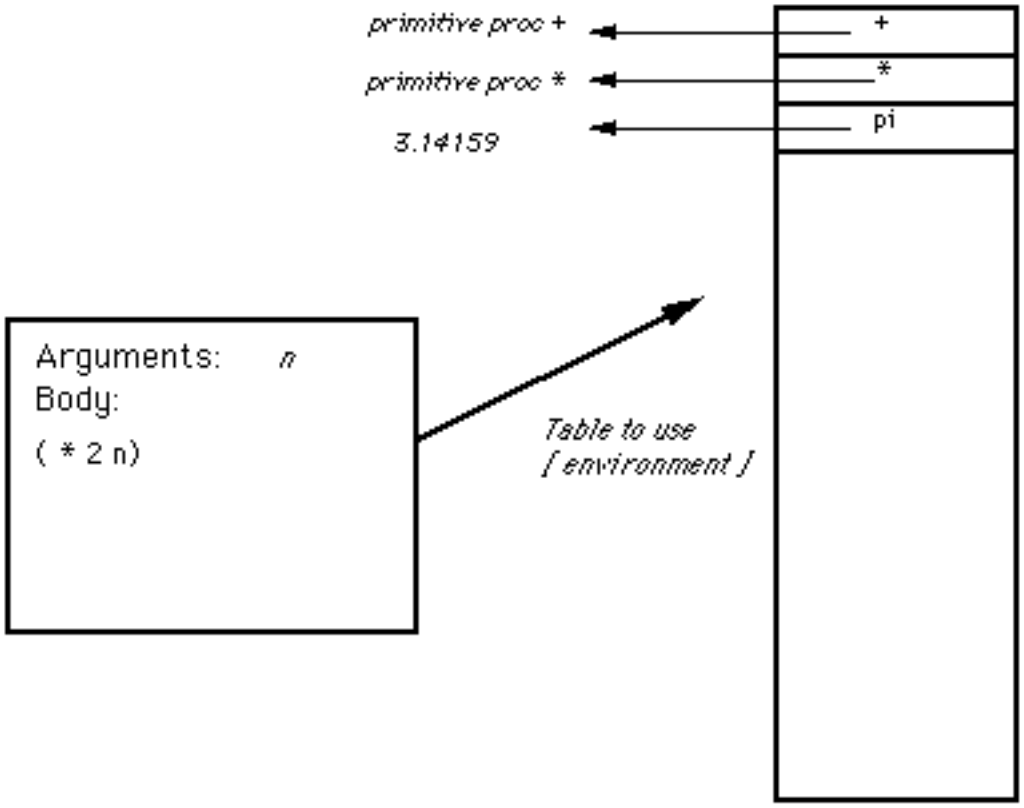
Actually, nobody knows what a procedure object looks like: you can imagine any structure that you're comfortable with. In the text, I used two little circles; here I've used a box. The important thing is that the procedure "knows about" two things: its text (arguments and body), and the table (environment) in which it was created and which it will use as the "parent table" for any future calls.

4.5 A diagram of the "initial environment" after evaluating a lambda expression

Suppose we evaluate the expression:

```
(lambda (n) (* 2 n))
```

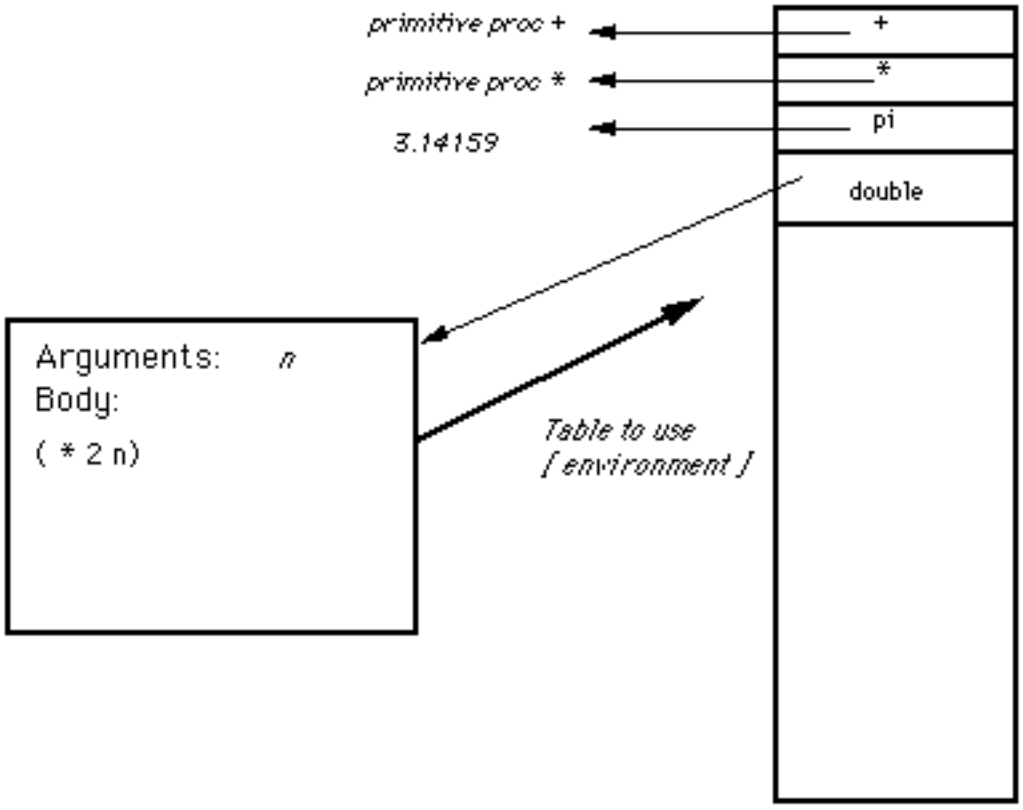
Here is a picture of the procedure object created:



This is the same idea as in the previous diagram, but now we've evaluated the expression:

```
(define (double n)
  (* 2 n))
```

Note that we have thereby added a new name to our environment.



We could have arrived at the same situation by evaluating:

```
(define double (lambda (n) (* 2 n))
```

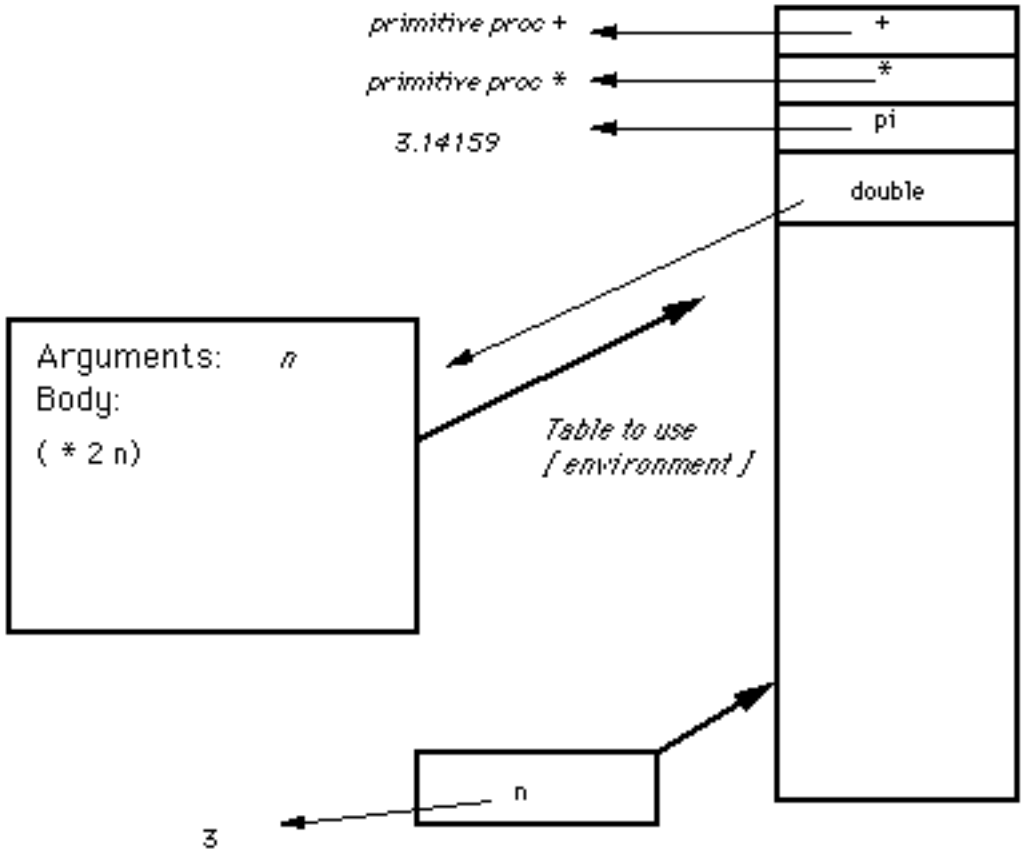
using the other form of define. In fact, the two expressions are equivalent: the Scheme interpreter automatically translates the "procedure" define into the version below, inserting a `lambda` expression to preserve the correct meaning.

4.6 Calling the newly-created `double` procedure

Suppose we now evaluate the expression

```
(double 3)
```

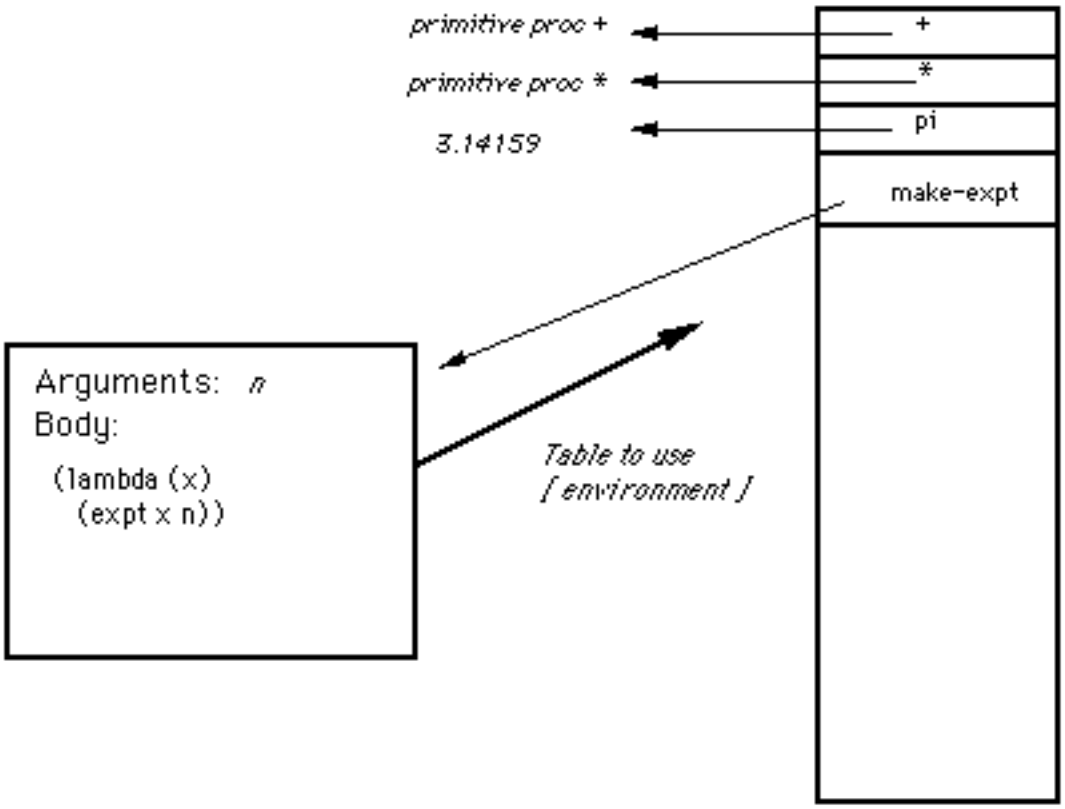
Here is a picture of the environment created for the purpose of performing that evaluation. First, we evaluate subexpressions: the name `double` evaluates to the procedure object at (upper) left; and the number 3 evaluates to a number object. Then we create a new "little table" in which the name *n* (the argument name for the procedure object) is bound to the number 3 (the result of evaluating the argument expression). We link the new "little table" to the table pointed to by the procedure object; and then we evaluate the expression (* 2 n) -- that is, the body of the procedure -- using the new composite table just created.



4.7 The make-expt procedure

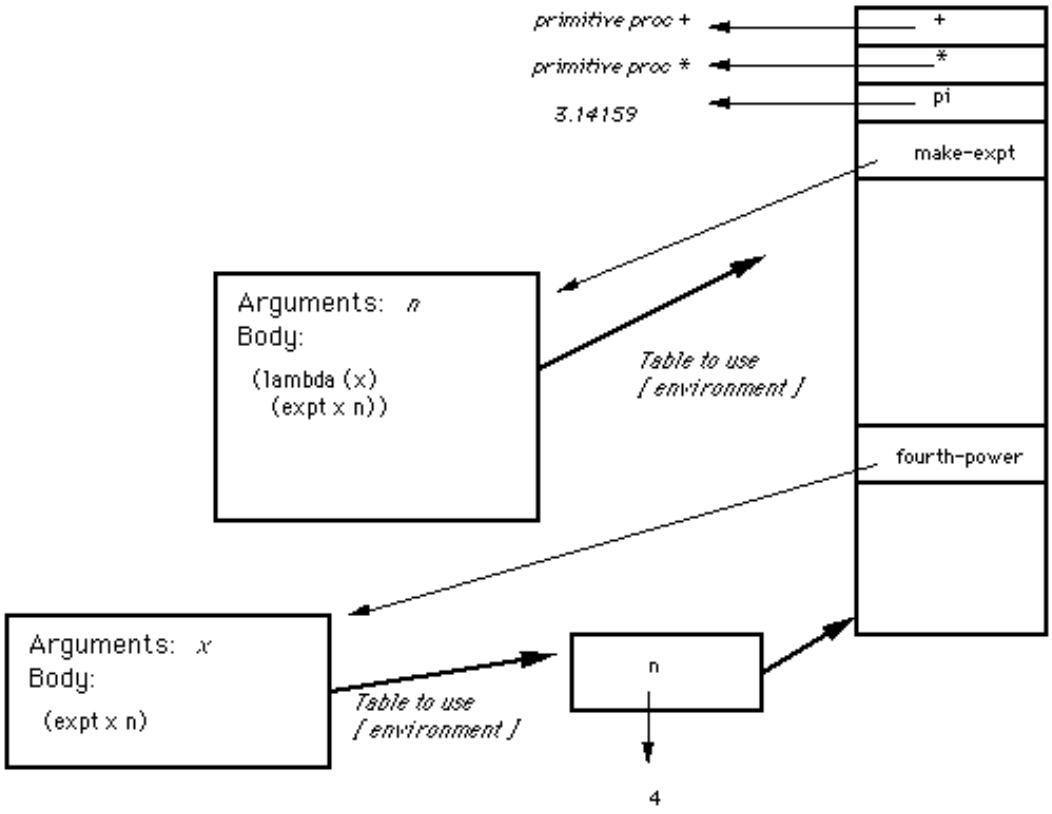
Here is a picture of the initial environment after evaluating the expression

```
(define (make-expt n)
  (lambda (x)
    (expt x n)))
```



Here is a picture of the initial environment after evaluating:

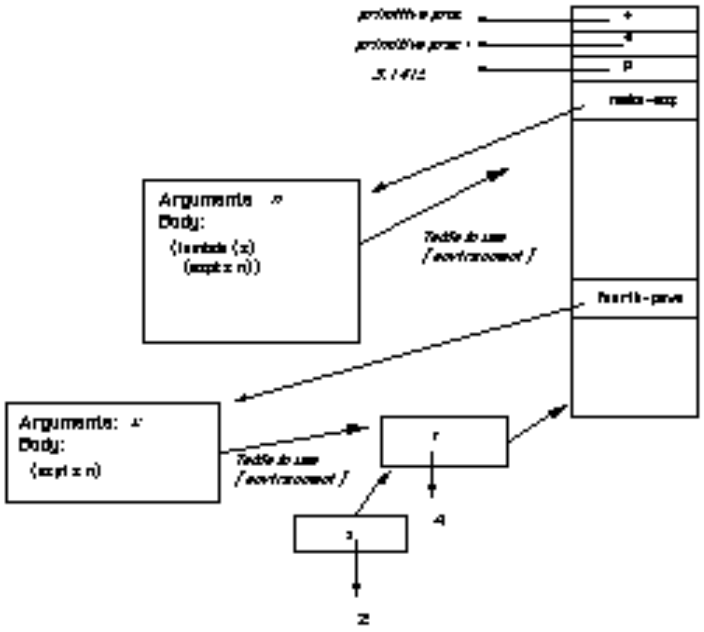
```
(define fourth-power
  (make-expt 4))
```



Now if we evaluate the expression

```
(fourth-power 2)
```

during the course of that evaluation process we create a new frame (or "little table") in which the name `x` is bound to the number object 2. In the new (three-fold) table, we evaluate the call to `expt`, and finally return the overall value of 16.



Here's what's going on, step by step:

(a) First, evaluate the names fourth-power and 2. The first expression returns the procedure object to which fourth-power is linked, and the second returns a number object.

(b) Create a new frame (little table) in which the name x (the argument of the procedure object) is bound to the number object 2.

(c) Link that new frame to the table pointed to by the procedure object being applied. (This is the point depicted by the drawing above.)

(d) Evaluate the body of the procedure (the call to expt) in the three-fold table just created, and return the result.

Part 2. Some Remaining Topics

4.8 The SET! special form

This material is covered in Chapter 11 of *Programming in MacScheme*.

The rule for evaluating an expression of the form:

(set! *name value-expression*)

in a given environment is as follows:

(i) Evaluate the value-expression.
(ii) Find the closest binding for name in the current environment (i.e., look up the chain of frames until you find a binding for name).
(iii) Reset the binding of name to the value of value-expression found in (i).

Informally, what is going on here is that we are changing the value of a given name by using SET!.

Example:

```
>>> (define a 5)  ;; First we create a binding for A

>>> a
5

>>> (set! a 10)   ;; Now we change it
10

>>> a
10
```

Parenthetically, if you evaluate a SET! expression for a name that currently has no binding, a new binding is created in the initial environment. But most Scheme programmers prefer to think of SET! as a binding-changer rather than a binding-creater (which is really DEFINE's) role.

While we're on the subject of SET! versus DEFINE, it is worth comparing a subtle difference between the rules for the two special forms. The rule for a DEFINE expression (given in terms of the environment model) is as follows. To evaluate an expression of the form

```
(define name value-expression)
```

you do the following:

(i) Evaluate the value-expression
(ii) Create a new binding for name in the current frame (i.e., the lowest frame of this environment). Name should now be bound to the value of value-expression found in (i). If there already is a binding for name, change it so that name is bound to the value of value-expression.

Again, note the difference here. SET! will look through an environment, trying to find a binding to change; DEFINE, on the other hand, affects only the lowest frame of the environment in which the DEFINE expression is evaluated.

To see a more interesting use of a SET! expression, consider the following technique for creating a "counter object":

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

>>> (define counter-1 (make-counter 0))
counter-1

>>> (counter-1)
1

>>> (counter-1)
2

>>> (define counter-2 (make-counter 0))
counter-2

>>> (counter-2)
1

>>> (counter-1)
3
```

You are strongly encouraged to work this example through using the environment model of evaluation. You will find that each counter "object" (in this case, `counter-1` and `counter-2`) is actually a procedure object associated with the same text but with distinct, independent environments. Note that this example would not work if we used a DEFINE expression instead of a SET! expression in the definition of `make-counter`. (Why?)

4.9 The set-car! and set-cdr! procedures

In the case of `set!`, we saw a special form whose role is to alter existing bindings between names and their values. But we haven't seen any procedures whose role is to actually change *objects* themselves. In most cases of object types, this is exactly as it should be—we wouldn't want any procedure to change the value of (say) the number object 4 into a number object representing 5. In other words, objects such as numbers and booleans ought to be immutable.

On the other hand, occasionally we do want to write a procedure that will change the value of a pair object—that is, to actually change that object so that the car and/or cdr arrows point to different objects than they did previously. The `set-car!` and `set-cdr!` primitive procedures play this role in Scheme.

Each of these procedures takes two arguments—a pair object and a new value—and returns the original pair object, but with the new value substituted for the car (or cdr) of the that pair. Here are a couple of examples:

```
>>> (define list-of-numbers '(1 2 3))
list-of-numbers

>>> (set-car! list-of-numbers 5)
(5 2 3)

;; note below that list-of-numbers is bound
;; to the same "pair object", but the contents
;; pointed to by that pair object have changed.
>>> list-of-numbers
(5 2 3)

>>> (define other-list '(a b c))
other-list

>>> (set-cdr! other-list (cdr list-of-numbers))
(a 2 3)

>>> other-list
(a 2 3)
```

4.10 The Let special form

One last special form is worth introducing here, if only because it comes up so often in practice. This is the `let` special form, which is used as an easy shorthand for providing "local bindings" to names while an expression is evaluated. For example, if we see an expression like the following:

```
(let ((val 3))
  (+ val 5))
```

we could read this (in "prose") as "with `val` temporarily bound to the number 3, evaluate the expression `(+ val 5)`."

The general form of a let expression is as follows:

```
(let ((name1 binding-exp1)
      (name2 binding-exp2)
           .
           .
           .)
     <body-expressions>
     )
```

Each binding-expression is evaluated in the environment **E** in which the `let` expression appears; a new, local frame is created (attached to the environment **E**) in which each name is bound to the result of its matching binding-expression. Within this newly-created environment, the body-expressions are

evaluated in turn (typically there is only one expression), and the result of the final body-expression is the result of the overall `let` expression.

This is a precise way—perhaps it appears to you as a pedantic way—of specifying the evaluation rule for a `let` expression. Intuitively, we wish to evaluate the body-expressions with the particular bindings for name1, name2, and so forth in effect. Notice, however, that according to our rule for evaluating a `let` expression, the following example would not work:

```
(let ((a 2)
      (b (+ a 3))
   (* b a))
```

We might expect this expression to return the value 10, but note the crucial error here: the binding expression for `b` is evaluated in the environment in which the overall `let` expression appears; and in this environment (presumably) there is no binding for `a`. In other words, the successive binding-expressions cannot make use of the results of bindings that appear earlier in the let expression.

Another way of describing the rule for evaluating a let expression is to note that let is just shorthand—or "syntactic sugar"—for an equivalent form that makes use of the lambda special form. We could translate the general let expression written earlier into this version:

```
((lambda (name1 name2 ... namen) <body-expressions>)
 binding-exp1 binding-exp2 ... binding-expn)
```

In other words, you can think of a `let` expression as simply making a procedure—an unnamed procedure—whose arguments are the temporary names of the `let` expression, and whose body is the body of the `let` expression. This procedure is then called on the values returned by the binding-expressions. It is perfectly accurate, if you like, to think of the interpreter as first *rearranging* a `let` expression into this equivalent form, and then evaluating that newly-produced form.