Scoping

In Scheme scoping is static (lexical). This means that non-local identifiers are bound to containing lambda parameters, or let values, or globally defined values. For example,

```
(define (f x)
        (lambda (y) (+ x y)))
```

Function **`f`** takes one parameter, **`x`**. It returns a function (of **`y`**), with **`x`** in the returned function bound to the value of **`x`** used when **`f`** was called.

Thus

`(f 10)` $\equiv$ **`(lambda (y) (+ 10 y))`**

`((f 10) 12)` $\Rightarrow$ **`22`**

Unbound symbols are assumed to be globals; there is a run-time error if an unbound global is referenced. For example,

```
(define (p y) (+ x y))
(p 20) ; error -- x is unbound
(define x 10)
(p 20) ⇒ 30
```

We can use let bindings to create private local variables for functions:

```
(define F
          (let  ( (X  1) )
               (lambda ()  X)
          )
)
```

`F` is a function (of no arguments).

`(F)` calls `F`.

```
(define X 22)
(F) ⇒ 1;X used in F is private
```

We can *encapsulate* internal state with a function by using private, let-bound variables:

```
(define  cnt
    (let  (  (I  0) )
       (lambda ()
            (set! I (+ I 1))   I)
    )
)
```

Now,

```
(cnt) ⇒ 1
(cnt) ⇒ 2
(cnt) ⇒ 3
```

etc.

Let Bindings can be Subtle

You must check to see if the let-bound value is created when the function is *created* or when it is *called*.

Compare

```
(define  cnt
   (let  (  (I  0) )
      (lambda ()
                (set! I (+ I 1))  I)
   )
)
```

vs.

```
(define  reset
   (lambda ()
      (let  (  (I  0) )
         (set! I (+ I 1))  I)
   )
)
```

(reset) ⇒ 1, (reset) ⇒ 1, etc.

Simulating Class Objects

Using association lists and private bound values, we can *encapsulate* data and functions. This gives us the effect of class objects.

```
(define (point x y)
  (list
   (list 'rect
         (lambda () (list x y)))
   (list 'polar
         (lambda ()
          (list
           (sqrt (+ (* x x) (* y y)))
           (atan (/ x y))
          )
         )
   )
  )
)
```

A call `(point 1 1)` creates an association list of the form

`( (rect  funct)  (polar  funct) )`

We can use **`structure`** to access components:

```
(define  p  (point 1 1) )
( (structure 'rect p) ) ⇒ (1 1)
( (structure 'polar p) ) ⇒
```

$$(\sqrt{2}\ \ \frac{\pi}{4})$$

We can add new functionality by just adding new `(id function)` pairs to the association list.

```
(define (point x y)
  (list
   (list 'rect
         (lambda () (list x y)))
   (list 'polar
         (lambda ()
          (list
           (sqrt (+ (* x x) (* y y)))
           (atan (/ x y))
   )))
   (list 'set-rect!
         (lambda (newx newy)
                 (set! x newx)
                 (set! y newy)
                 (list x y)
   ))
   (list 'set-polar!
         (lambda (r theta)
           (set! x (* r (sin theta)))
           (set! y (* r (cos theta)))
           (list r theta)
   ))
))
```

Now we have

```
(define  p  (point 1 1) )
( (structure 'rect p) ) ⇒ (1 1)
( (structure 'polar p) ) ⇒
```

$(\sqrt{2} \ \ \frac{\pi}{4})$

```
((structure 'set-polar! p) 1 π/4)
 ⇒ (1 π/4)
 ( (structure 'rect p) ) ⇒
```

$(\frac{1}{\sqrt{2}} \ \ \frac{1}{\sqrt{2}})$

Limiting Access to Internal Structure

We can improve upon our association list approach by returning a single function (similar to a C++ or Java object) rather than an explicit list of (id function) pairs.

The function will take the name of the desired operation as one of its arguments.

First, let’s differentiate between

```
(define  def1
  (let  (  (I 0)  )
    (lambda () (set! I (+ I 1)) I)
  )
)
```

and

```
(define  (def2)
  (let  (  (I 0)  )
    (lambda () (set! I (+ I 1)) I)
  )
)
```

`def1` is a function of zero arguments that increments a local variable and returns its updated value.

`def2` is a function (of zero arguments) that *generates* a function of zero arguments (that increments a local variable and returns its updated value). Each call to `def2` creates a *different* function.

Stack Implemented as a Function

```
(define  ( stack )
   (let  (  (s   () ) )
      (lambda (op . args) ; var # args
       (cond
        ((equal? op 'push!)
          (set!  s (cons (car args) s))
           (car s))
        ((equal?  op  'pop!)
           (if (null? s)
                #f
                (let  ( (top  (car s)) )
                     (set!  s  (cdr s))
                     top )))
        ((equal? op 'empty?)
         (null? s))
        (else   #f)
        )
       )
   )
)
```

```
(define stk (stack));new empty stack
(stk 'push! 1) ⇒ 1 ;s = (1)
(stk 'push! 3) ⇒ 3 ;s = (3 1)
(stk 'push! 'x) ⇒ x ;s = (x 3 1)
(stk 'pop!) ⇒ x ;s = (3 1)
(stk 'empty?) ⇒ #f ;s = (3 1)
(stk 'dump) ⇒ #f ;s = (3 1)
```

Higher-Order Functions

A higher-order function is a function that takes a function as a parameter or one that returns a function as its result.

A very important (and useful) higher-order function is **map**, which applies a function to a list of values and produces a list or results:

```
(define (map f L)
  (if (null? L)
      ()
      (cons (f (car L))
            (map f (cdr L)))
  )
)
```

Note: In Scheme's built-in implementation of **map**, the order of function application is unspecified.

```
(map sqrt '(1 2 3 4 5))  ⇒
  (1  1.414  1.732  2  2.236)
(map  (lambda(x) (* x x))
           '(1 2 3 4 5))  ⇒
  (1 4 9 16 25)
```

Map may also be used with multiple argument functions by supplying more than one list of arguments:

```
(map + '(1 2 3) '(4 5 6))  ⇒
  (5 7 9)
```

The Reduce Function

Another useful higher-order function is `reduce`, which reduces a list of values to a single value by repeatedly applying a binary function to the list values.

This function takes a binary function, a list of data values, and an identity value for the binary function:

```
(define
  (reduce f L id)
  (if (null? L)
      id
      (f (car L)
         (reduce f (cdr L) id))
  )
)
```

`(reduce + '(1 2 3 4 5) 0)` ⇒ `15`

`(reduce * '(1 2 4 6 8 10) 1)` ⇒ `3840`

```
(reduce append
 '((1 2 3) (4 5 6) (7 8)) ())
```
$\Rightarrow$ `(1 2 3 4 5 6 7 8)`

`(reduce expt '(2 2 2 2) 1)` $\Rightarrow$

$2^{2^{2^{2}}}$ `= 65536`

`(reduce expt '(2 2 2 2 2) 1)`
$\Rightarrow 2^{65536}$

```
(string-length
 (number->string
  (reduce expt '(2 2 2 2 2) 1)))
```
$\Rightarrow$ `19729 ; digits in` 2^{65536}