Functions are First-class Objects

Functions may be passed as parameters, returned as the value of a function call, stored in data objects, etc. This is a consequence of the fact that

\((\text{lambda} \ (\text{args}) \ (\text{body}))\)

evaluates to a function just as

\((+ \ 1 \ 1)\)

evaluates to an integer.
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,

\[
\text{(define (f x)} \\
\quad \text{(lambda (y) (+ x y)))}
\]

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

\[
(f 10) \equiv (\text{lambda (y) (+ 10 y)})
\]

\[
((f 10) 12) \Rightarrow 22
\]
Unbound symbols are assumed to be globals; there is a run-time error if an unbound global is referenced. For example,

\[
\text{(define (p y) (+ x y))}
\]

\[
\text{(p 20) ; error -- x is unbound}
\]

\[
\text{(define x 10)}
\]

\[
\text{(p 20) ⇒ 30}
\]

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

\[
\text{(define F}
\]

\[
\quad \text{(let ( (X 1) )}
\]

\[
\quad \quad \text{(lambda () X)}
\]

\[
\quad \}
\]

\[
\text{F is a function (of no arguments).}
\]

\[
\text{(F) calls F.}
\]

\[
\text{(define X 22)}
\]

\[
\text{(F) ⇒ 1;X used in F is private}
\]
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
          (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:

```scheme
(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

\[
\begin{align*}
&(\text{define } p \ (\text{point} \ 1 \ 1) ) \\
&(\ (\text{structure} \ '\text{rect} \ p) ) \Rightarrow (1 \ 1) \\
&(\ (\text{structure} \ '\text{polar} \ p) ) \Rightarrow \\
&\quad (\sqrt{2} \ \frac{\pi}{4})
\end{align*}
\]

\[
((\text{structure} \ '\text{set-polar!} \ p) \ 1 \ \pi/4) \\
\Rightarrow (1 \ \pi/4)
\]

\[
(\ (\text{structure} \ '\text{rect} \ p) ) \Rightarrow \\
\quad (\frac{1}{\sqrt{2}} \ \frac{1}{\sqrt{2}})
\]
**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

\[
\begin{align*}
\text{(define def1} & \quad \text{(let ( (I 0) )} \\
& \quad \quad \text{(lambda () (set! I (+ I 1)) I) )} \\
\text{)} \\
\text{and} \\
\text{(define (def2) } \quad \text{(let ( (I 0) )} \\
& \quad \quad \text{(lambda () (set! I (+ I 1)) I) )} \\
\text{)}
\end{align*}
\]

\text{def1} is a zero argument function that increments a local variable and returns its updated value. 

\text{def2} is a a zero argument function that \textit{generates} a function of zero arguments (that increments a local variable and returns its updated value). Each call to \text{def2} creates a \textit{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)
        )
      )
    ))
  )
)
\[
\begin{align*}
\text{define stk (stack)} &; \text{new empty stack} \\
\text{stk 'push! 1)} & \Rightarrow 1 \; ; s = (1) \\
\text{stk 'push! 3)} & \Rightarrow 3 \; ; s = (3 \ 1) \\
\text{stk 'push! 'x)} & \Rightarrow x \; ; s = (x \ 3 \ 1) \\
\text{stk 'pop!)} & \Rightarrow x \; ; s = (3 \ 1) \\
\text{stk 'empty?)} & \Rightarrow \#f \; ; s = (3 \ 1) \\
\text{stk 'dump)} & \Rightarrow \#f \; ; s = (3 \ 1)
\end{align*}
\]
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 \texttt{map}, which applies a function to a list of values and produces a list of results:

\begin{verbatim}
(define (map f L)
  (if (null? L)
      ()
      (cons (f (car L))
           (map f (cdr L))))
)
\end{verbatim}

Note: In Scheme's built-in implementation of \texttt{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 \texttt{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:

\begin{verbatim}
(define (reduce f L id)
  (if (null? L)
    id
    (f (car L)
      (reduce f (cdr L) id)))
)

(reduce + '(1 2 3 4 5) 0) \Rightarrow 15
(reduce * '(1 2 4 6 8 10) 1) \Rightarrow 3840
\end{verbatim}
\[(\text{reduce} \ \text{append} \ 
\quad \text{'(}(1 \ 2 \ 3) \ (4 \ 5 \ 6) \ (7 \ 8)) \ (())) \Rightarrow (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8)\]

\[(\text{reduce} \ \text{expt} \ \text{'(}(2 \ 2 \ 2 \ 2) \ 1) \Rightarrow \]

\[2^{2^{2^{2^2}}} = 65536\]

\[(\text{reduce} \ \text{expt} \ \text{'(}(2 \ 2 \ 2 \ 2 \ 2) \ 1) \Rightarrow 2^{65536}\]

\[(\text{string-length} \ 
\quad \text{(number->string} \ 
\quad \text{(reduce} \ \text{expt} \ \text{'(}(2 \ 2 \ 2 \ 2) \ 1)))) \Rightarrow \]

\[19729; \text{digits in } 2^{65536}\]
**Sharing vs. Copying**

In languages without side-effects an object can be copied by copying a pointer (reference) to the object; a complete new copy of the object isn’t needed.

Hence in Scheme \(\text{(define A B)}\) normally means

\[A \rightarrow B\]
But, if side-effects are possible we may need to force a physical copy of an object or structure:

\[
\text{(define (copy obj)}
\begin{align*}
&\quad \text{(if (pair? obj)} \\
&\qquad \quad \text{ (cons (copy (car obj))} \\
&\qquad \qquad \quad \text{(copy (cdr obj))}) \\
&\qquad \text{obj}
\end{align*}
\text{)}
\]
For example,

\[
\text{(define A '}(1\ 2))
\]

\[
\text{(define B (cons A A))}
\]

\[
\text{B} = (\ (1\ 2)\ 1\ 2)
\]\n
\[
A
\]

\[
B
\]

\[
1
\]

\[
2
\]

\[
()
\]
(set-car! (car B) 10)
B = ( (10 2) 10 2)

(define C (cons (copy A) (copy A)))

C

A

B

10

10

2

()
(set-car! (car C) 20)
C = ((20 2) 10 2)

Similar concerns apply to strings and vectors, because their internal structure can be changed.
Shallow & Deep Copying

A copy operation that copies a pointer (or reference) rather than the object itself is a shallow copy. For example, in Java,

Object O1 = new Object();
Object O2 = new Object();
O1 = O2; // shallow copy

If the structure within an object is physically copied, the operation is a deep copy.

In Java, for objects that support the clone operation,

O1 = O2.clone(); // deep copy

Even in Java’s deep copy (via the clone() operation), objects referenced from within an object are shallow copied. Thus given
class List {
    int value;
    List next;
}
List L, M;
M = L.clone();
L.value and M.value are independent, but L.next and M.next refer to the same List object.
A complete deep copy, that copies all objects linked directly or indirectly, is expensive and tricky to implement.
(Consider a complete copy of a circular linked list).