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
\[(\text{lambda (args) (body)})\]
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,
\[
\text{(define (f x)}
\text{\quad (lambda (y) (+ x y)))}
\]
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,
\[
\text{(define (p y) (+ x y))}
\]
\[
(p \ 20) \ ; \text{error -- x is unbound}
\]
\[
\text{(define x 10)}
\]
\[
(p \ 20) \Rightarrow 30
\]
We can use let bindings to create private local variables for functions:
\[
\text{(define F}
\text{\quad (let ( (X 1) )}
\text{\quad \quad (lambda () X))}
\text{)}
\]
\(F\) is a function (of no arguments). \(F\) calls \(F\).
\[
(F) \Rightarrow 1; \text{X used in F is private}
\]
We can encapsulate internal state with a function by using private, let-bound variables:
\[
\text{(define cnt}
\text{\quad (let ( (I 0) )}
\text{\quad \quad (lambda ()}
\text{\quad \quad \quad (set! I (+ I 1)) I)))}
\text{)}
\]
Now,
\[
(cnt) \Rightarrow 1
\]
\[
(cnt) \Rightarrow 2
\]
\[
(cnt) \Rightarrow 3
\]
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

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

VS.

```scheme
(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.

```scheme
(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:

```scheme
(define p (point 1 1))
((structure 'rect p)) ⇒ (1 1)
((structure 'polar p)) ⇒ (√2 π/4)
```

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

```scheme
(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 pi/4)
```

```
((structure 'set-polar! p) 1 pi/4) ⇒ (1 pi/4)
((structure 'rect p) ) ⇒ ( sqrt 2 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 zero argument function that increments a local variable and returns its updated value.

def2 is a a zero argument function 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
      (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)) ())) ⇒ (1 2 3 4 5 6 7 8)
(reduce expt '(2 2 2 2) 1) ⇒ 2^2^2 = 65536
(reduce expt '(2 2 2 2) 1) ⇒ 2^{65536}
(string-length (number->string (reduce expt '(2 2 2 2) 1))) ⇒ 19729 ; 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 (define A B) normally means

But, if side-effects are possible we may need to force a physical copy of an object or structure:
(define (copy obj)
  (if (pair? obj)
      (cons (copy (car obj))
            (copy (cdr obj)))
      obj))

For example,
(define A '(1 2))
(define B (cons A A))
B = ( (1 2) 1 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).