Using `assoc`, we can easily define a structure function:

\[
\text{(structure key alist)} \quad \text{will return the value associated with key in \ \text{alist}; in C or Java notation, it returns \text{alist.key}.}
\]

\[
\text{(define} \\
\text{(structure key alist)} \quad \text{(if (assoc key alist)} \\
\text{\quad (car (cdr (assoc key alist))) \ #f)} \\
\text{\quad )})
\]

We can improve this function in two ways:
- The same call to `assoc` is made twice; we can save the value computed by using a `let` expression.
- Often combinations of `car` and `cdr` are needed to extract a value. Scheme has a number of predefined functions that combine several calls to `car` and `cdr` into one function. For example,

\[
\text{(caar x) ≡ (car (car x))} \\
\text{(cadr x) ≡ (car (cdr x))} \\
\text{(cdar x) ≡ (cdr (car x))} \\
\text{(cddr x) ≡ (cdr (cdr x))}
\]

Using these two insights we can now define a better version of `structure`:

\[
\text{(define} \\
\text{(structure key alist)} \quad \text{(let ((p (assoc key alist)))} \\
\text{\quad (if p \\
\text{\quad \quad (cadr p) \ #f)}} \\
\text{\quad \quad )})
\]

What does `assoc` do if more than one sublist with the same key exists? It returns the first sublist with a matching key. In fact, this property can be used to make a simple and fast function that updates association lists:

\[
\text{(define} \\
\text{(set-structure key alist val)} \quad \text{(cons (list key val) alist)})
\]

If we want to be more space-efficient, we can create a version that updates the internal structure of an association list, using `set-cdr!` which changes the `cdr` value of a list:

\[
\text{(define} \\
\text{(set-structure! key alist val)} \quad \text{(let ((p (assoc key alist)))} \\
\text{\quad (if p \\
\text{\quad \quad (begin \\
\text{\quad \quad \quad (set-cdr! p (list val)) \ \text{alist)}} \\
\text{\quad \quad \quad (cons (list key val) alist)}) \\
\text{\quad \quad )})}
\]
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,
\[
(\text{define} \ (f \ x) \\
(\text{lambda} \ (y) \ (+ \ x \ y)))
\]
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,
\[
(\text{define} \ (p \ y) \ (+ \ x \ y)) \\
(p \ 20) ; \text{error -- } x \text{ is unbound} \\
(\text{define} \ x \ 10) \\
(p \ 20) \Rightarrow 30
\]

We can use let bindings to create private local variables for functions:
\[
(\text{define} \ F \\
(\text{let} \ ( (X \ 1) ) \\
(\text{lambda} () \ X) \\
) \\
F \text{ is a function (of no arguments).}
(F) \text{ calls } F. \\
(\text{define} \ X \ 22) \\
(F) \Rightarrow 1; X \text{ used in } F \text{ is private}
\]

We can encapsulate internal state with a function by using private, let-bound variables:
\[
(\text{define} \ cnt \\
(\text{let} \ ((I \ 0)) \\
(\text{lambda} () \\
(\text{set!} \ I \ (+ \ I \ 1)) \ I) \\
) \\
)
\]

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

(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) ) ⇒ (√2 π 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)))
      )
    )
  )
)

(define (point x y)
  (list
    (list 'set-rect!
      (lambda (newx newy)
        (set! x newx)
        (set! y newy)
        (list x y)
      )
    )
  )
)

(define (point x y)
  (list
    (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} \quad \frac{\pi}{4})

((structure 'set-polar! p) 1 \pi/4)
⇒ (1 \pi/4)
( (structure 'rect p) ) ⇒
\left(\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}\right)