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

`(structure key alist)` will return the value associated with `key` in `alist`; in C or Java notation, it returns `alist.key`.

```scheme
(define
  (structure key alist)
  (if (assoc key alist)
      (car (cdr (assoc key alist)))
      #f)
)
```

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,

\begin{align*}
\text{caar } x & \equiv (\text{car } (\text{car } x)) \\
\text{cadr } x & \equiv (\text{car } (\text{cdr } x)) \\
\text{cdar } x & \equiv (\text{cdr } (\text{car } x)) \\
\text{cddr } x & \equiv (\text{cdr } (\text{cdr } x))
\end{align*}

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

\begin{verbatim}
(define (structure key alist)
  (let ((p (assoc key alist)))
    (if p
      (cadr p)
      #f))
)
\end{verbatim}
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:

```
(define
  (set-structure key alist val)
  (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:

```scheme
(define 
  (set-structure! key alist val)
  (let ( (p (assoc key alist)))
    (if p
      (begin
        (set-cdr! p (list val))
        alist
      )
    (cons (list key val) alist)
    )
  )
)
```
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

(lambda (args) (body))

evaluates to a function just as

(+ 1 1)

evaluates to an integer.
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,

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

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,

\begin{verbatim}
(define (p y) (+ x y))
(p 20) ; error -- x is unbound
(define x 10)
(p 20) ⇒ 30
\end{verbatim}

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

\begin{verbatim}
(define F
  (let ((X 1))
    (lambda () X))
)
\end{verbatim}

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

\((F)\) calls \(F\).

\begin{verbatim}
(define X 22)
(F) ⇒ 1; X used in F is private
\end{verbatim}
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.

```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:

(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

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

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

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