Records and Structs

In Scheme we can represent a record, struct, or class object as an association list of the form

\[(\text{obj1 val1}) (\text{obj2 val2}) \ldots\]  

In the association list, which is a list of \((\text{object value})\) sublists, object serves as a “key” to locate the desired sublist.

For example, the association list

\[
( (A 10) (B 20) (C 30) )
\]

serves the same role as

```
struct
{
    int a = 10;
    int b = 20;
    int c = 30;
}
```
The predefined Scheme function
(assoc obj alist)
checks alist (an association list) to see if it contains a sublist with obj as its head. If it does, the list starting with obj is returned; otherwise #f (indicating failure) is returned.

For example,
(define L
  '( (a 10) (b 20) (c 30) ) )
(assoc 'a L) ⇒ (a 10)
(assoc 'b L) ⇒ (b 20)
(assoc 'x L) ⇒ #f
We can use non-atomic objects as keys too!

\[
\text{(define price-list}
\]
\[
'( ( (bmw m5) 71095) \\
( (bmw z4) 40495) \\
( (jag xj8) 56975) \\
( (mb sl500) 86655) \\
) 
\)

\[
\text{(assoc ' (bmw z4) price-list)} \\
\Rightarrow ( (bmw z4) 40495)
\]
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.
```

```
(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,

\( (\text{caar } x) \equiv (\text{car} (\text{car } x)) \)
\( (\text{cadr } x) \equiv (\text{car} (\text{cdr } x)) \)
\( (\text{cdar } x) \equiv (\text{cdr} (\text{car } x)) \)
\( (\text{cddr } x) \equiv (\text{cdr} (\text{cdr } x)) \)

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

\[
\begin{align*}
(\text{define} & \quad \text{(structure key alist)}) \\
& \quad \text{(let ((p (assoc key alist)))} \\
& \quad \quad \text{(if p} \\
& \quad \quad \quad \text{(cadr p)} \\
& \quad \quad \quad \quad \text{#f} \\
& \quad \quad \text{)} \\
& \quad \text{)} \\
& \text{)}
\end{align*}
\]
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.
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 \)
\[\quad (\text{let } (\ (X \ 1) ) \]
\[\quad \ (\lambda () \ X)\]
\[\quad )\]
\[F \text{ is a function (of no arguments).} \]
\[(F) \ \text{calls } F. \]
\[(\text{define } X \ 22)\]
\[(F) \Rightarrow 1; X \ \text{used in } F \text{ is private} \]
We can encapsulate internal state with a function by using private, let-bound variables:

```
(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

```scheme
(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.

\[
\text{(define (point x y)} \\
\text{(list)} \\
\text{\hspace{1em} (list 'rect)} \\
\text{\hspace{2em} (lambda () (list x y)))} \\
\text{\hspace{1em} (list 'polar)} \\
\text{\hspace{2em} (lambda ()}} \\
\text{\hspace{3em} (list)} \\
\text{\hspace{4em} (sqrt (+ (* x x) (* y y)))} \\
\text{\hspace{4em} (atan (/ x y)))}} \\
\text{\hspace{1em})}} \\
\text{\hspace{1em})}} \\
\text{\hspace{1em})}} \\
\text{\hspace{1em})}}
\]

A call \text{(point 1 1)} creates an association list of the form

\[
\text{( (rect funct) (polar funct))}
\]
We can use structure to access components:

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

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