Records and Structs

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

```scheme
((obj1 val1) (obj2 val2) ...)
```

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

For example, the association list

```scheme
( (A 10) (B 20) (C 30) )
```

serves the same role as

```scheme
struct
{
    int a = 10;
    int b = 20;
    int c = 30;
}
```

We can use non-atomic objects as keys too!

```scheme
(define price-list
  '( ((bmw m5) 71095)
      ((bmw z4) 40495)
      ((jag xj8) 56975)
      ((mb sl500) 86655)
  )
)
```

```scheme
(assoc '(bmw z4) price-list) ⇒ ((bmw z4) 40495)
```

The predefined Scheme function

```scheme
(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,

```scheme
(define L
  '( (a 10) (b 20) (c 30) ) )
(assoc 'a L) ⇒ (a 10)
(assoc 'b L) ⇒ (b 20)
(assoc 'x L) ⇒ #f
```

Using assoc, we can easily define a structure function:

```scheme
(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,

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

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

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

What does \text{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:

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

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

\[\text{(define (set-structure! } \text{key} \text{alist } \text{val)}\]
\[\text{(let ((p (assoc } \text{key} \text{alist))}\]
\[\text{(if } p\]
\[\text{(begin}\]
\[\text{(set-cdr! } p \text{ (list } \text{val})\]
\[\text{alist}\]
\[)\]
\[\text{alist}\]
\[\text{cons } (\text{list } \text{key } \text{val}) \text{alist}\]
\[)\]
\[)\]

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
\[\text{(lambda } (\text{args}) \text{ (body))}\]
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,

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

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,

\[
\text{(define (p y) (+ x y))}
\]
\[
(p 20) \text{ ; error -- x is unbound}
\]
\[
\text{(define x 10)}
\]
\[
(p 20) \Rightarrow 30
\]

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

\[
\text{(define cnt}
\quad \text{(let ( (I 0) )}
\quad \lambda ()
\quad \text{(set! I (+ I 1)) I)}
\quad )
\]
\[
\text{(cnt)} \Rightarrow 1
\]
\[
\text{(cnt)} \Rightarrow 2
\]
\[
\text{(cnt)} \Rightarrow 3
\]
\[
\text{etc.}
\]

\[
\text{VS.}
\]
\[
\text{(define reset}
\quad \lambda ()
\quad \text{(let ( (I 0) )}
\quad \text{(set! I (+ I 1)) I)}
\quad )
\]
\[
\text{(reset)} \Rightarrow 1, \text{(reset)} \Rightarrow 1, \text{etc.}
\]

We can encapsulate internal state with a function by using private, let-bound variables:

\[
\text{(define cnt}
\quad \text{(let ( (I 0) )}
\quad \lambda ()
\quad \text{(set! I (+ I 1)) I)}
\quad )
\]
\[
\text{Now,}
\]
\[
\text{(cnt)} \Rightarrow 1
\]
\[
\text{(cnt)} \Rightarrow 2
\]
\[
\text{(cnt)} \Rightarrow 3
\]
\[
\text{etc.}
\]

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

\[
\text{(define cnt}
\quad \text{(let ( (I 0) )}
\quad \lambda ()
\quad \text{(set! I (+ I 1)) I)}
\quad )
\]
\[
\text{(cnt)} \Rightarrow 1
\]
\[
\text{(cnt)} \Rightarrow 2
\]
\[
\text{(cnt)} \Rightarrow 3
\]
\[
\text{etc.}
\]
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 'set-rect!
  (lambda (newx newy)
    (set! x newx)
    (set! y newy)
    (list x y)
  ))

(define '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) ) ⇒
( (√2 π/4)

((structure 'set-polar! p) 1 π/4)
⇒ (1 π/4)
((structure 'rect p) ) ⇒
( (√2 √2)
```