Data Structures in Scheme

In Scheme, lists and S-expressions are basic. Arrays can be simulated using lists, but access to elements “deep” in the list can be slow (since a list is a linked structure). To access an element deep within a list we can use:

- (list-tail \( L \) \( k \))
  This returns list \( L \) after removing the first \( k \) elements. For example,
  \[
  (\text{list-tail } '(1 2 3 4 5) 2) \Rightarrow (3 4 5)
  \]

- (list-ref \( L \) \( k \))
  This returns the \( k \)-th element in \( L \) (counting from 0). For example,
  \[
  (\text{list-ref } '(1 2 3 4 5) 2) \Rightarrow 3
  \]
Vectors in Scheme

Scheme provides a vector type that directly implements one dimensional arrays.

Literals are of the form #( ... )

For example, #(1 2 3) or #(1 2.0 "three")

The function (vector? val) tests whether val is a vector or not.

(vector? 'abc) ⇒ #f
(vector? '(a b c)) ⇒ #f
(vector? #(a b c)) ⇒ #t

The function (vector v1 v2 ... ) evaluates v1, v2, ... and puts them into a vector.

(vector 1 2 3) ⇒ #(1 2 3)
The function \( \text{(make-vector k val)} \) creates a vector composed of \( k \) copies of \( \text{val} \). Thus
\[
\text{(make-vector 4 (/ 1 2))} \Rightarrow \\
\text{#(1/2 1/2 1/2 1/2)}
\]
The function \( \text{(vector-ref vect k)} \) returns the \( k \)-th element of \( \text{vect} \), starting at position 0. It is essentially the same as \( \text{vect}[k] \) in C or Java. For example,
\[
\text{(vector-ref #(2 4 6 8 10) 3)} \Rightarrow 8
\]
The function \( \text{(vector-set! vect k val)} \) sets the \( k \)-th element of \( \text{vect} \), starting at position 0, to be \( \text{val} \). It is essentially the same as \( \text{vect}[k]=\text{val} \) in C or Java. The value returned by the function is unspecified. The suffix “!” in \( \text{set!} \) indicates that the function has a side-effect.
For example,

\[
\begin{align*}
&\text{(define } v \ (\#(1 \ 2 \ 3 \ 4 \ 5)) \\
&\text{(vector-set! } v \ 2 \ 0) \\
&v \ \Rightarrow \ \#(1 \ 2 \ 0 \ 4 \ 5)
\end{align*}
\]

Vectors aren’t lists (and lists aren’t vectors).

Thus \text{(car } \#(1 \ 2 \ 3)) \text{ doesn’t work.}

There are conversion routines:

- \text{(vector->list } V) \text{ converts vector } V \text{ to a list containing the same values as } V. \text{ For example,}
  \[
  \text{(vector->list } \#(1 \ 2 \ 3)) \Rightarrow (1 \ 2 \ 3)
  \]

- \text{(list->vector } L) \text{ converts list } L \text{ to a vector containing the same values as } L. \text{ For example,}
  \[
  \text{(list->vector } \'(1 \ 2 \ 3)) \Rightarrow \#(1 \ 2 \ 3)
  \]
In general Scheme names a conversion function from type $T$ to type $Q$ as $T \rightarrow Q$. For example, **string->list** converts a **string** into a **list** containing the characters in the string.


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 (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

\{
  \text{int a = 10;}
  \text{int b = 20;}
  \text{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!

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

(assoc '(bmw z4) price-list)
⇒ ((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`.

```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

\[
\begin{align*}
\text{(define} & \\
\text{ (structure key alist))} \\
\text{(let} & \ ((\text{p (assoc key alist)})) \\
\text{(if} & \ p \\
\text{ (cadr p)} & \ #f \\
\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:

```scheme
(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:

```
(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

$$\text{(lambda (args) (body))}$$
evaluates to a function just as

$$\text{(+ 1 1)}$$
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)
\text{ )}
\]

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

\((F)\) 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

(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 \texttt{structure} to access components:

\begin{verbatim}
(define p (point 1 1))
((structure 'rect p) ) ⇒ (1 1)
((structure 'polar p) ) ⇒
(\sqrt{2} \ \frac{\pi}{4})
\end{verbatim}
We can add new functionality by just adding new \texttt{(id function)} pairs to the association list.

\begin{verbatim}
(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))
  ))
\end{verbatim}
Now we have

\[(\text{define } p \text{ (point 1 1)) \implies (1 1)}\]

\[(\text{structure 'rect p)} \) \implies (1 1)\]

\[(\text{structure 'polar p)} \) \implies (\sqrt{2} \ \pi \ \frac{\pi}{4})\]

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

\[(\text{structure 'rect p)} \) \implies (\frac{1}{\sqrt{2}} \ \frac{1}{\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

```scheme
(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 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)