**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,
  `(list-tail '(1 2 3 4 5) 2) ⇒ (3 4 5)`

- `(list-ref L k)`
  This returns the `k`-th element in `L` (counting from 0). For example,
  `(list-ref '(1 2 3 4 5) 2) ⇒ 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)) ⇒ #t`
- `(vector? #((a b c))) ⇒ #f`

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

- `(vector 1 2 3) ⇒ #(1 2 3)`

The function `(make-vector k val)` creates a vector composed of `k` copies of `val`. Thus

- `(make-vector 4 (/ 1 2)) ⇒ #((1/2 1/2 1/2 1/2))`

The function `(vector-ref vect k)` returns the `k`-th element of `vect`, starting at position 0. It is essentially the same as `vect[k]` in C or Java. For example,

- `(vector-ref #(2 4 6 8 10) 3) ⇒ 8`

The function `(vector-set! vect k val)` sets the `k`-th element of `vect`, starting at position 0, to be `val`. It is essentially the same as `vect[k]=val` in C or Java. The value returned by the function is unspecified. The suffix “!” indicates that the function has a side-effect. For example,

- `(define v #(1 2 3 4 5))`
- `(vector-set! v 2 0)`
- `v ⇒ #(1 2 0 4 5)`

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

Thus `(car #(1 2 3))` doesn’t work.

There are conversion routines:

- `(vector->list v)` converts vector `v` to a list containing the same values as `v`. For example,
  `(vector->list #(1 2 3)) ⇒ (1 2 3)`

- `(list->vector L)` converts list `L` to a vector containing the same values as `L`. For example,
  `(list->vector '(1 2 3)) ⇒ #(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

\[
((\text{A 10}) (\text{B 20}) (\text{C 30})
\]

serves the same role as

```scheme
define 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,

```scheme
(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!

```scheme
(define price-list '( ( (bmw m5) 71095) ((bmw z4) 40495) ((jag xj8) 56975) ((mb sl500) 86655) )
)
(assoc '(bmw z4) price-list) ⇒ ((bmw z4) 40495)
```
Using \texttt{assoc}, we can easily define a structure function:

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

\begin{verbatim}
(define (structure key alist)
  (if (assoc key alist)
      (car (cdr (assoc key alist)))
      #f)
)
\end{verbatim}

We can improve this function in two ways:

\begin{itemize}
  \item The same call to \texttt{assoc} is made twice; we can save the value computed by using a \texttt{let} expression.
  \item Often combinations of \texttt{car} and \texttt{cdr} are needed to extract a value. Scheme has a number of predefined functions that combine several calls to \texttt{car} and \texttt{cdr} into one function. For example,

\begin{verbatim}
(car x) \equiv (car (car x))
(cadr x) \equiv (car (cdr x))
(cdar x) \equiv (cdr (car x))
(cddr x) \equiv (cdr (cdr x))
\end{verbatim}

Using these two insights we can now define a better version of \texttt{structure}.

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

What does \texttt{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:

\begin{verbatim}
(define (set-structure key alist val)
  (cons (list key val) alist))
\end{verbatim}

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

\begin{verbatim}
(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))
  )
)
\end{verbatim}
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

\[ (+ 1 1)\]

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,

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

\[ (p 20) ; \text{error -- x is unbound} \]

\[ (\text{define x 10}) \]

\[ (p 20) \Rightarrow 30 \]

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 \(F\).

\[ (\text{define X 22}) \]

\[ (F) \Rightarrow 1; X \text{ used in F is private} \]

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

\[
\text{(define cnt}
\text{ (let ( (I 0) )}
\text{ (lambda ()}
\text{ (set! I (+ I 1)) I))}
\text{)}
\]

Now,

\[ (\text{cnt}) \Rightarrow 1 \]

\[ (\text{cnt}) \Rightarrow 2 \]

\[ (\text{cnt}) \Rightarrow 3 \]

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.