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))` ⇒ `#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 `(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 "!" in `set!` 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->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

((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

((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!

```
(define price-list
  '( ((bmw m3)      40270)
     ((bmw 740)     62070)
     ((jag  xj8)    55330)
     ((mb slk230)   40295)
   )
)
(assoc '(bmw 740) price-list)
    ⇒ ((bmw 740)  62070)
```

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,

(caar x) ≡ **(car (car x))**

(cadr x) ≡ **(car (cdr x))**

(cdar x) ≡ **(cdr (car x))**

(cddr x) ≡ **(cdr (cdr x))**

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

```
(define
   (structure key alist)
   (let ((p (assoc key alist)))
     (if p
        (cadr p)
        #f
     )
   )
)
```


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:

```
(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.