**Subsets**

Another good example of Scheme’s recursive style of programming is subset computation.

Given a list of distinct atoms, we want to compute a list of all subsets of the list values.

For example,

```scheme
(subsets '(1 2 3)) ⇒
( () (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))
```

The order of atoms and sublists is unimportant, but all possible subsets of the list values must be included.

Given Scheme’s recursive style of programming, we need a recursive definition of subsets.

Thus (subsets '(1 2 3)) ⇒
( () (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3)) =
( () (1) (2) (1 2) ) plus
( (3) (1 3) (2 3) (1 2 3) )

This insight leads to a concise program for subsets.

We will let (distrib L E) be a function that “distributes” E into each list in L.

For example,

```scheme
(distrib '(() (1) (2) (1 2)) 3) =
( (3) (3 1) (3 2) (3 1 2) )
```

```scheme
(define (distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L))
            (distrib (cdr L) E))
  )
)
```

We will let (extend L E) extend a list L by distributing element E through L and then appending this result to L.

For example,

```scheme
(extend '( () (a) ) 'b) ⇒
( () (a) (b) (b a))
```

```scheme
(define (extend L E)
  (append L (distrib L E))
)
```

Now subsets is easy:

```scheme
(define (subsets L)
  (if (null? L)
      (list ())
      (extend (subsets (cdr L))
              (car L))
  )
)
```
**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)
⇒ #(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)
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

struct

\[
\begin{align*}
\{ & \text{ int a = 10; } \\
& \text{ int b = 20; } \\
& \text{ int c = 30; } \\
\}
\end{align*}
\]

The predefined Scheme function

\text{(assoc obj alist)}

checks \text{alist} (an association list) to see if it contains a sublist with \text{obj} as its head. If it does, the list starting with \text{obj} is returned; otherwise \text{#f} (indicating failure) is returned.

For example,

\[
\text{(define L ' ( (a 10) (b 20) (c 30) ))}
\]

\[
\text{(assoc 'a L) \Rightarrow (a 10)}
\]

\[
\text{(assoc 'b L) \Rightarrow (b 20)}
\]

\[
\text{(assoc 'x L) \Rightarrow #f}
\]

We can use non-atomic objects as keys too!

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

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

In general Scheme names a conversion function from type \text{T} to type \text{Q} as \text{T->Q}. For example, \text{string->list} converts a \text{string} into a list containing the characters in the string.
Using **assoc**, we can easily define a **structure** function:

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

\[
\begin{align*}
\text{(caar x)} & \equiv (\text{car (car x)}) \\
\text{(cadr x)} & \equiv (\text{car (cdr x)}) \\
\text{(cdar x)} & \equiv (\text{cdr (car x)}) \\
\text{(cddr x)} & \equiv (\text{cdr (cdr x)})
\end{align*}
\]

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

```scheme
(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:

```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:

```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))
  )
)