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,

(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.
That is, if we have a list of all subsets of \( n \) atoms, how do we extend this list to one containing all subsets of \( n+1 \) values?

First, we note that the number of subsets of \( n+1 \) values is exactly twice the number of subsets of \( n \) values.

For example,

\[
(\text{subsets } '(1 2) ) \Rightarrow \n( () (1) (2) (1 2)), \text{ which contains 4 subsets.}
\]

\[
(\text{subsets } '(1 2 3)) \text{ contains 8 subsets (as we saw earlier).}
\]

Moreover, the extended list (of subsets for \( n+1 \) values) is simply the list of subsets for \( n \) values plus the result of “distributing” the new value into each of the original subsets.
Thus \( \text{subsets } ' (1 \ 2 \ 3) \Rightarrow \)
\[
( () \ (1) \ (2) \ (3) \ (1 \ 2) \ (1 \ 3) \\
(2 \ 3) \ (1 \ 2 \ 3)) =
\]
\[
( () \ (1) \ (2) \ (1 \ 2) ) \quad \text{plus}
\]
\[
( (3) \ (1 \ 3) \ (2 \ 3) \ (1 \ 2 \ 3) )
\]
This insight leads to a concise program for subsets.

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

For example,

\[
(\text{distrib } '(()) \ (1) \ (2) \ (1 \ 2)) \ 3) =
\]
\[
( (3) \ (3 \ 1) \ (3 \ 2) \ (3 \ 1 \ 2) )
\]

\[
(\text{define } \text{distrib } L \ E)
\]
\[
(\text{if } (\text{null? } L)
\]
\[
()
\]
\[
(\text{cons } (\text{cons } E \ (\text{car } L))
\]
\[
(\text{distrib } (\text{cdr } L) \ E))
\]
\[
)
)
We will let \((\text{extend } L E)\) extend a list \(L\) by distributing element \(E\) through \(L\) and then appending this result to \(L\).

For example,

\[
(\text{extend } '( () (a) ) 'b) \Rightarrow
( () (a) (b) (b a))
\]

\[
(\text{define } (\text{extend } L E)
(\text{append } L (\text{distrib } L E))
)
\]

Now subsets is easy:

\[
(\text{define } (\text{subsets } L)
(\text{if } (\text{null? } L)
(\text{list } ())
(\text{extend } (\text{subsets } (\text{cdr } L))
(\text{car } L))
)
)
\]
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 \( \text{val} \). Thus

\[
\text{(make-vector 4 (/ 1 2))} \Rightarrow \#(1/2\ 1/2\ 1/2\ 1/2)
\]

The function `(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 `(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 set! indicates that the function
has a side-effect. For example,

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

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

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

There are conversion routines:

- \text{\textbf{(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{\textbf{(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, \texttt{string\textasciitilde{}list} converts a string into a list containing the characters in the string.