The Let Construct

Scheme allows us to create local names, bound to values, for use in an expression.

The structure is

\[(\text{let} \ (\ (\text{id}_1 \ \text{val}_1) \ (\text{id}_2 \ \text{val}_2) \ldots \ ) \ \text{expr} \ )\]

In this construct, \(\text{val}_1\) is evaluated and bound to \(\text{id}_1\), which will exist only within this \text{let} expression. If \(\text{id}_1\) is already defined (as a global or parameter name) the existing definition is hidden and the local definition, bound to \(\text{val}_1\), is used. Then \(\text{val}_2\) is evaluated and bound to \(\text{id}_2\), .... Finally, \(\text{expr}\) is evaluated in a scope that includes \(\text{id}_1\), \(\text{id}_2\), ...
For example,

\[
(\text{let } (\ (a \ 10) \ (b \ 20) ) \ (\ + \ a \ b) ) \Rightarrow 30
\]

Using a \texttt{let}, the definition of \texttt{revall}, a version of \texttt{rev} that reverses all levels of a list, is easy:

\[
\begin{align*}
(\text{define } (\texttt{revall } \texttt{L}) & ) \\
(\texttt{if } (\texttt{null? } \texttt{L}) & ) \\
& \texttt{L} \\
& (\texttt{(let } ((E (\texttt{if } (\texttt{list? } (\texttt{car } \texttt{L}))) \\
& \phantom{(} (\texttt{revall } \texttt{(car } \texttt{L}) \ \\
& \phantom{(} (\texttt{car } \texttt{L} )))) \\
& \phantom{(} (\texttt{append } (\texttt{revall } \texttt{(cdr } \texttt{L})) \\
& \phantom{(} (\texttt{list } E)) \\
& \phantom{(} ) \\
& ) \\
& ) \\
(\texttt{(revall ' ( (1 2) (3 4) )) } \Rightarrow \\
& ((4 3) (2 1))
\end{align*}
\]
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,

$$\text{(subsets '}(1 \ 2 \ 3)) \Rightarrow$$

$$\text{( ( ) (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))}$$

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,

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

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

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 \( (\text{subsets } '(1 2 3)) \Rightarrow \)
\[
( () (1) (2) (3) (1 2) (1 3)
 (2 3) (1 2 3)) =
( () (1) (2) (1 2) ) \text{ plus }
( (3) (1 3) (2 3) (1 2 3) )
\]

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 (null? } L)\)
\[
 ( ()
 (\text{cons } (\text{cons } E \text{ (car } L))
\text{ (distrib } (\text{cdr } L) E))
\text{ ) }
\text{ )}
\]
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,

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

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

Now subsets is easy:

(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) \Rightarrow (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) \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 `(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,

\begin{verbatim}
(define v #(1 2 3 4 5))
(vector-set! v 2 0)
\end{verbatim}
\begin{verbatim}
v ⇒ #(1 2 0 4 5)
\end{verbatim}

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

Thus \begin{verbatim}(car #(1 2 3))\end{verbatim} doesn't work.

There are conversion routines:

- \begin{verbatim}(vector->list V)\end{verbatim} converts vector \begin{verbatim}V\end{verbatim} to a list containing the same values as \begin{verbatim}V\end{verbatim}. For example,
  \begin{verbatim}(vector->list #(1 2 3))⇒(1 2 3)\end{verbatim}

- \begin{verbatim}(list->vector L)\end{verbatim} converts list \begin{verbatim}L\end{verbatim} to a vector containing the same values as \begin{verbatim}L\end{verbatim}. For example,
  \begin{verbatim}(list->vector '(1 2 3))⇒#(1 2 3)\end{verbatim}
In general Scheme names a conversion function from type $T$ to type $Q$ as $T \rightarrow Q$. For example, \texttt{string} $\rightarrow$ \texttt{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

\[(\text{assoc obj alist})\]

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

For example,

\[
\begin{align*}
\text{(define } L \text{ '( (a 10) (b 20) (c 30) ) )} \\
\text{(assoc 'a L) } \Rightarrow (a 10) \\
\text{(assoc 'b L) } \Rightarrow (b 20) \\
\text{(assoc 'x L) } \Rightarrow \text{#f}
\end{align*}
\]
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)