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

\[
\text{(define (revall L) )}
\]
\[
\text{(if (null? L) L)
}\]
\[
\text{(let ((E (if (list? (car L)) (revall (car L)) (car L)))))
}\]
\[
\text{(append (revall (cdr L)) (list E))}
\]
\[
\text{(revall } '((1 2) (3 4))) \Rightarrow 
\]
\[
((4 3) (2 1))
\]

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,

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

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 \text{twice} the number of subsets of \( n \) values.

For example,

\[
\text{(subsets } '((1 2)) \Rightarrow 
\]
\[
( () (1) (2) (1 2)), \text{ which contains 4 subsets.}
\]
\[
\text{(subsets } '((1 2 3)) \text{ contains 8 subsets (as we saw earlier).}
\]

Moreover, the extended list (of subsets for \( n+1 \) values) is simply the list of subsets for \( n \) values \text{plus} the result of “distributing” the new value into each of the original subsets.
Thus \((\text{subsets } ' (1 2 3)) \Rightarrow \)
\begin{align*}
( () & (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) \\
\end{align*}
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 \textit{subsets} is easy:

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

\section*{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:

\begin{itemize}
\item \((\text{list-tail } L k)\)
This returns list \(L\) after removing the first \(k\) elements. For example,
\[
(\text{list-tail } ' (1 2 3 4 5) 2) \Rightarrow \\
(3 4 5)
\]
\item \((\text{list-ref } L k)\)
This returns the \(k\)-th element in \(L\) (counting from 0). For example,
\[
(\text{list-ref } ' (1 2 3 4 5) 2) \Rightarrow 3
\]
\end{itemize}

\section*{Vectors in Scheme}

Scheme provides a vector type that directly implements one dimensional arrays.

Literals are of the form \#\(\cdot\)\(\cdot\)\(\cdot\)\(\cdot\)\(\cdot\)

For example, \#( 1 2 3 ) or \#( 1 2.0 "three")

The function \((\text{vector? } \text{val})\) tests whether \textit{val} is a vector or not.

\[
(\text{vector? } ' \text{abc}) \Rightarrow \#f
\]
\[
(\text{vector? } '(a b c)) \Rightarrow \#f
\]
\[
(\text{vector? } #(a b c)) \Rightarrow \#t
\]

The function \((\text{vector } v_1 v_2 \ldots)\) evaluates \(v_1, v_2, \ldots\) and puts them into a vector.

\[
(\text{vector } 1 2 3) \Rightarrow \#(1 2 3)
\]
The function `(make-vector k val)` creates a vector composed of \( k \) copies of `val`. Thus

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

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,

\[
\text{(vector-ref #(2 4 6 8 10) 3) ⇒ 8}
\]

The function

\[
\text{(vector-set! vect k val)}
\]

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,

\[
\text{(define v #(1 2 3 4 5))}
\]

\[
\text{(vector-set! v 2 0)}
\]

\[
v ⇒ #(1 2 0 4 5)
\]

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,

\[
\text{(vector->list #(1 2 3)) ⇒ (1 2 3)}
\]

- `(list->vector L)` converts list \( L \) to a vector containing the same values as \( L \). For example,

\[
\text{(list->vector '(1 2 3)) ⇒ #(1 2 3)}
\]

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

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

serves the same role as

```scheme
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 m5)  71095)
     ((bmw z4)  40495)
     ((jag xj8) 56975)
     ((mb sl500) 86655)
   )
)

(assoc '(bmw z4) price-list) ⇒ ((bmw z4) 40495)

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)
      (cadr (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,

(car 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)
    )
  )
)
```