The Let Construct

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

The structure is
(let ( (id1 val1) (id2 val2) ... )
expr)

In this construct, val1 is evaluated and bound to id1, which will exist only within this let expression. If id1 is already defined (as a global or parameter name) the existing definition is hidden and the local definition, bound to val1, is used. Then val2 is evaluated and bound to id2, ... Finally, expr is evaluated in a scope that includes id1, id2, ...
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:

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

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

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 \textit{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 \textit{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,

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

\[
( (3) (3 1) (3 2) (3 1 2) )
\]

(define (distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L))
        (distrib (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 \text{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 \((\text{make-vector } k \ \text{val})\)
creates a vector composed of \(k\) copies of \(\text{val}\). Thus
\[
(\text{make-vector } 4 \ (\text{/} \ 1 \ 2)) \Rightarrow \text{ #(1/2 1/2 1/2 1/2)}
\]
The function \((\text{vector-ref } \text{vect} \ k)\) 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,
\[
(\text{vector-ref } \text{ #(2 4 6 8 10)} \ 3) \Rightarrow 8
\]
The function
\[
(\text{vector-set! } \text{vect} \ k \ \text{val})
\]
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 \text{set!}\ indicates that the function has a side-effect.
For example,

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

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

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

There are conversion routines:

- \text{(vector->list } V)\text{ converts vector } V\text{ to a list containing the same values as } V.\text{ For example,}
  \[
  \text{(vector->list } \text{ #(1 2 3))} \Rightarrow \text{ (1 2 3)}
  \]

- \text{(list->vector } L)\text{ converts list } L\text{ to a vector containing the same values as } L.\text{ For example,}
  \[
  \text{(list->vector } \text{ '(1 2 3))} \Rightarrow \text{ #(1 2 3)}
  \]
• In general Scheme names a conversion function from type \( T \) to type \( Q \) as \( T \rightarrow Q \). For example, \( \text{string} \rightarrow \text{list} \) converts a \text{string} into a \text{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

```scheme
((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

```scheme
((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
\((\text{assoc } \text{obj } \text{alist})\)
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 \texttt{#f} (indicating failure) is returned.

For example,
\[(\text{define } \text{L} \\quad ' ( (a 10) (b 20) (c 30) ) ) \]
\[(\text{assoc } 'a \text{ L}) \Rightarrow (a 10) \]
\[(\text{assoc } 'b \text{ L}) \Rightarrow (b 20) \]
\[(\text{assoc } 'x \text{ L}) \Rightarrow \texttt{#f} \]
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)
        (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 \texttt{car} and \texttt{cdr} into one function. For example,

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

Using these two insights we can now define a better version of \texttt{structure}

\begin{verbatim}
(define
  (structure key alist)
  (let ((p (assoc key alist)))
    (if p
      (cadr p)
      #f))
)
\end{verbatim}
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)
  )
)
)