**User-defined Functions**

The list

```
(lambda (args) (body))
```

evaluates to a function with

`(args)` as its argument list and

`(body)` as the function body.

No quotes are needed for

`(args)` or `(body)`.

Thus

```
(lambda (x) (+ x 1))
```
evaluates to the increment function.

Similarly,

```
((lambda (x) (+ x 1)) 10) ⇒ 11
```

We can bind values and functions to global symbols using the `define` function.

The general form is

```
(define id object)
```

`id` is not evaluated but `object` is. `id` is bound to the value `object` evaluates to.

For example,

```
(define pi 3.1415926535)
(define plus1
  (lambda (x) (+ x 1))
)(define pi*2 (* pi 2))
```

Once a symbol is defined, it evaluates to the value it is bound to:

```
(plus1 12) ⇒ 13
```

Since functions are frequently defined, we may abbreviate

```
(define id
  (lambda (args) (body)))
```
as

```
(define (id args) (body))
```

Thus

```
(define (plus1 x) (+ x 1))
```

**Conditional Expressions in Scheme**

A *predicate* is a function that returns a boolean value. By convention, in Scheme, predicate names end with “?”

For example,

```
number?  symbol?  equal?  null?  list?
```

In conditionals, `#f` is false, and everything else, including `#t`, is true.

The `if` expression is

```
(if pred E1 E2)
```

First `pred` is evaluated. Depending on its value (`#f` or not), either `E1` or `E2` is evaluated (but not both) and returned as the value of the `if` expression.
For example,

\[
\text{(if } (= \ 1 \ (+ \ 0 \ 1)) \ \\
\ \\
'Yes \ \\
'No \ \\
)\]

\[
\text{(define } \ (fact \ n) \ \\
\ \\
\text{(if } (= \ n \ 0) \ \\
\ \\
1 \ \\
\ \\
(* \ n \ (fact \ (- \ n \ 1))) \ \\
\)\]

\[
\text{Generalized Conditional} \]

This is similar to a switch or case:

\[
\text{(cond } \ \\
\text{(p1 } e1) \ \\
\text{(p2 } e2) \ \\
\ldots \ \\
\text{(else } en) \ \\
)\]

Each of the predicates \((p1, p2, \ldots)\) is evaluated until one is true \((\neq \ #f)\). Then the corresponding expression \((e1, e2, \ldots)\) is evaluated and returned as the value of the \text{cond}. \text{else} acts like a predicate that is always true.

Example:

\[
\text{(cond } \ \\
\text{((= a 1) } 2) \ \\
\text{((= a 2) } 3) \ \\
\text{(else } 4) \ \\
)\]

\[
\text{Recursion in Scheme} \]

Recursion is widely used in Scheme and most other functional programming languages. Rather than using a loop to step through the elements of a list or array, recursion breaks a problem on a large data structure into a simpler problem on a smaller data structure.

A good example of this approach is the \text{append} function, which joins (or appends) two lists into one larger list containing all the elements of the two input lists (in the correct order).

Note that \text{cons} \text{ is not append}. \text{cons} adds one element to the head of an existing list.

Thus

\[
\text{(cons } '(a \ b) \ '(c \ d)) \Rightarrow \ ((a \ b) \ c \ d) \]  

\[
\text{(append } '(a \ b) \ '(c \ d)) \Rightarrow \ (a \ b \ c \ d) \]

The \text{append} function is predefined in Scheme, as are many other useful list-manipulating functions (consult the Scheme definition for what’s available).

It is instructive to define \text{append} directly to see its recursive approach:

\[
\text{(define } \ (append \ L1 \ L2) \ \\
\ \\
\text{(if } (\text{null?} \ L1) \ \\
\ \\
L2 \ \\
\ \\
\text{(cons} \ (\text{car} \ L1) \ \\
\ \\
\text{(append} \ (\text{cdr} \ L1) \ L2) \ \\
\)) \ \\
)\]

Thus

\[
\text{(cons} \ '(a \ b) \ '(c \ d)) \Rightarrow \ ((a \ b) \ c \ d) \]  

\[
\text{(append} \ '(a \ b) \ '(c \ d)) \Rightarrow \ (a \ b \ c \ d) \]

The \text{append} function is predefined in Scheme, as are many other useful list-manipulating functions (consult the Scheme definition for what’s available).

It is instructive to define \text{append} directly to see its recursive approach:

\[
\text{(define} \ (append \ L1 \ L2) \ \\
\ \\
\text{(if} \ (\text{null?} \ L1) \ \\
\ \\
L2 \ \\
\ \\
\text{(cons} \ (\text{car} \ L1) \ \\
\ \\
\text{(append} \ (\text{cdr} \ L1) \ L2) \ \\
\)) \ \\
)\]
Let's trace \((\text{append } '(a \ b) \ '(c \ d))\) 

Our definition is 
\[
(\text{define} \\
(\text{append} \ L1 \ L2) \\
(\text{if} \ (\text{null?} \ L1) \ \\
L2 \\
(\text{cons} \ (\text{car} \ L1) \\
(\text{append} \ (\text{cdr} \ L1) \ L2)) \\
))
\]

Now \(L1 = (a \ b)\) and \(L2 = (c \ d)\). 

\((\text{null?} \ L1)\) is false, so we evaluate 
\[
(\text{cons} \ (\text{car} \ L1) \ (\text{append} \ (\text{cdr} \ L1) \ L2)) \\
= (\text{cons} \ (\text{car} \ '(a \ b)) \\
(\text{append} \ (\text{cdr} \ '(a \ b)) \ '(c \ d))) \\
= (\text{cons} \ 'a \ (\text{append} \ '(b) \ '(c \ d)))
\]

We need to evaluate 
\[
(\text{append} \ '(b) \ '(c \ d))
\]
In this call, \(L1 = (b)\) and \(L2 = (c \ d)\). 

\(L1\) is not null, so we evaluate 
\[
(\text{cons} \ (\text{car} \ L1) \ (\text{append} \ (\text{cdr} \ L1) \ L2)) \\
= (\text{cons} \ (\text{car} \ '(b)) \\
(\text{append} \ (\text{cdr} \ '(b)) \ '(c \ d))) \\
= (\text{cons} \ 'b \ (\text{append} \ () \ '(c \ d))
\]

We need to evaluate 
\[
(\text{append} \ () \ '(c \ d))
\]
In this call, \(L1 = ()\) and \(L2 = (c \ d)\). 

\(L1\) is null, so we return \((c \ d)\). 

Therefore 
\[
(\text{cons} \ 'b \ (\text{append} \ () \ '(c \ d)) = \\
(\text{cons} \ 'b \ '(c \ d)) = (b \ c \ d) = \\
(\text{append} \ '(b) \ '(c \ d))
\]

Finally, 
\[
(\text{append} \ '(a \ b) \ '(c \ d)) = \\
(\text{cons} \ 'a \ (\text{append} \ '(b) \ '(c \ d)) = \\
(\text{cons} \ 'a \ (b \ c \ d) = (a \ b \ c \ d)
\]

Note:
Source files for \textit{append}, and other Scheme examples, are in 
\~\texttt{cs538-1/public/scheme/example1.scm}, \~\texttt{cs538-1/public/scheme/example2.scm}, etc.

---

**Reversing a List**

Another useful list-manipulation function is \textit{rev}, which reverses the members of a list. That is, the last element becomes the first element, the next-to-last element becomes the second element, etc.

For example, 
\[
(\text{rev} \ '(1 \ 2 \ 3)) \Rightarrow (3 \ 2 \ 1)
\]

The definition of \textit{rev} is straightforward:
\[
(\text{define} \ (\text{rev} \ L) \\
(\text{if} \ (\text{null?} \ L) \\
L \\
(\text{append} \ (\text{rev} \ (\text{cdr} \ L)) \ \\
(\text{list} \ (\text{car} \ L))) \\
))
\]

---

As an example, consider
\[
(\text{rev} \ '(1 \ 2))
\]
Here \(L = (1 \ 2)\). \(L\) is not null so we evaluate 
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ L)) \\
(\text{list} \ (\text{car} \ L))) = \\
(\text{append} \ (\text{rev} \ '(1 \ 2)) \\
(\text{list} \ (\text{car} \ '(1 \ 2)))) = \\
(\text{append} \ '(2) \ (\text{list} \ 1)) = \\
(\text{append} \ '(2) \ '(1))
\]

We must evaluate \((\text{rev} \ '(2))\)
Here \(L = (2)\). \(L\) is not null so we evaluate 
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ L)) \\
(\text{list} \ (\text{car} \ L))) = \\
(\text{append} \ (\text{rev} \ '(2)) \\
(\text{list} \ (\text{car} \ '(2)))) = \\
(\text{append} \ '(2) \ (\text{list} \ 2)) = \\
(\text{append} \ '(2) \ '(2))
\]
We must evaluate \((\text{rev} \ '())\)
Here \(L = ()\). \(L\) is null so 
\[
(\text{rev} \ '()) = ()
\]
Thus \((\text{append } (\text{rev } ()) '(2)) = \text{append } ()' (2)) = (2) = (\text{rev } '(2))\)

Finally, recall \((\text{rev } '(1 2)) = \text{append } (\text{rev } '(2))' (1)) = \text{append } '(2)' (1) = (2 1)\)

As constructed, \text{rev} only reverses the “top level” elements of a list. That is, members of a list that themselves are lists aren’t reversed.

For example,
\[
(\text{rev } '((1 2) (3 4))) = ((3 4) (1 2))
\]

We can generalize \text{rev} to also reverse list members that happen to be lists.

To do this, it will be convenient to use Scheme’s \text{let} construct.

\[
\text{let } ( (\text{id1 val1}) (\text{id2 val2}) \ldots ) \text{ expr } \]

In this construct, \text{val1} is evaluated and bound to \text{id1}, which will exist only within this \text{let} expression. If \text{id1} is already defined (as a global or parameter name) the existing definition is hidden and the local definition, bound to \text{val1}, is used. Then \text{val2} is evaluated and bound to \text{id2}, .... Finally, \text{expr} is evaluated in a scope that includes \text{id1}, \text{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 } (\text{revall } L)
\text{if } (\text{null? } L)
L
\text{let } ((E (\text{if } (\text{list? } (\text{car } L))
(\text{revall } (\text{car } L))
(\text{car } L)))
(\text{append } (\text{revall } (\text{cdr } L))
(\text{list } E)))
)

(\text{revall } '((1 2) (3 4))) \Rightarrow ((4 3) (2 1))
\]

\[\text{Subsets}\]

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 twice the number of subsets of $n$ values.

For example,

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

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,

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

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

\[
\text{We will let (extend L E) extend a list L by distributing element E through L and then appending this result to L.}
\]

For example,

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

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

Now \text{subsets} is easy:

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