User-defined Functions

The list

\[(\text{lambda} \ (\text{args}) \ (\text{body}))\]

evaluates to a function with \(\text{(args)}\) as its argument list and \(\text{(body)}\) as the function body.

No quotes are needed for \(\text{(args)}\) or \(\text{(body)}\).

Thus

\[(\text{lambda} \ (x) \ (+ \ x \ 1))\]

evaluates to the increment function.

Similarly,

\[((\text{lambda} \ (x) \ (+ \ x \ 1)) \ 10) \Rightarrow 11\]
We can bind values and functions to global symbols using the \texttt{define} function. The general form is

\texttt{(define id object)}

\texttt{id} is not evaluated but \texttt{object} is. \texttt{id} is bound to the value \texttt{object} evaluates to.

For example,

\texttt{(define pi 3.1415926535)}

\texttt{(define plus1 (lambda (x) (+ x 1)))}

\texttt{(define pi*2 (* pi 2))}

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

\texttt{(plus1 12) \Rightarrow 13}
Since functions are frequently defined, we may abbreviate

\[
(\text{define id} \\
(\lambda (\text{args}) (\text{body})))
\]

as

\[
(\text{define (id args) (body)})
\]

Thus

\[
(\text{define (plus1 x) (+ x 1)})
\]
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,

```scheme
(if (= 1 (+ 0 1))
   'Yes
   'No
)

(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))
  )
)
**Generalized Conditional**

This is similar to a switch or case:

```
(cond
  (p1  e1)
  (p2  e2)
  ...
  (else  en)
)
```

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

Example:

```
(cond
  ((= a 1)  2)
  ((= a 2)  3)
  (else     4)
)
```
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 *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 *cons* is *not* *append*. *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 **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 **append** directly to see its recursive approach:

```
(define
  (append L1 L2)
  (if  (null? L1)
        L2
        (cons (car L1)
              (append (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
\[(\text{cons (car } L1) \ (\text{append } (\text{cdr } L1) \ L2))\]  
\[= \ (\text{cons (car } '(b)) \]  
\[\quad (\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 \texttt{append}, and other Scheme examples, are in  
\~{cs538-1/public/scheme/example1.scm,}  
\~{cs538-1/public/scheme/example2.scm,}  
etc.
Reversing a List

Another useful list-manipulation function is `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,

`(rev '(1 2 3)) ⇒ (3 2 1)`

The definition of `rev` is straightforward:

```scheme
(define (rev L)
    (if (null? L)
        L
        (append (rev (cdr L))
                (list (car L))
        )
    )
)
```
As an example, consider
\[(\text{rev}'(1\ 2))\]
Here \(L = (1\ 2)\). \(L\) is not null so we evaluate
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ L)) \\\n\phantom{+} \ (\text{list} \ (\text{car} \ L))) =
\]
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ '(1\ 2))) \\\n\phantom{+} \ (\text{list} \ (\text{car} \ '(1\ 2)))) =
\]
\[
(\text{append} \ (\text{rev} \ '(2)) \ (\text{list} \ 1)) =
\]
\[
(\text{append} \ (\text{rev} \ '(2)) \ '(1))
\]
We must evaluate \((\text{rev} \ '(2))\)
Here \(L = (2)\). \(L\) is not null so we evaluate
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ L)) \\\n\phantom{+} \ (\text{list} \ (\text{car} \ L))) =
\]
\[
(\text{append} \ (\text{rev} \ (\text{cdr} \ '(2))) \\\n\phantom{+} \ (\text{list} \ (\text{car} \ '(2)))) =
\]
\[
(\text{append} \ (\text{rev} \ ()) \ (\text{list} \ 2)) =
\]
\[
(\text{append} \ (\text{rev} \ ()) \ '(2))
\]
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 \textit{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 \texttt{let} construct.
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,
\[
(\text{let} \ (\ (a\ 10)\ (b\ 20)\ ) \n(+\ a\ b)) \Rightarrow 30
\]

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

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

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

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

For example,

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

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

Moreover, the extended list (of subsets for \( n+1 \) values) is simply the list of subsets for \( n \) values \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 \( (\text{distrib } \text{L } \text{E}) \) be a function that “distributes” \( \text{E} \) into each list in \( \text{L} \).

For example,

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

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

(\text{if} \ (\text{null? } \text{L})

()

(cons (cons \text{E} (\text{car } \text{L}))

(\text{distrib } (\text{cdr } \text{L}) \text{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 (extend } L \ E) \\
\quad (\text{append } L \ (\text{distrib } L \ E)) \\
\) \\
\]

Now \texttt{subsets} is easy:

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