Generalized Conditional

This is similar to a switch or case:

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

Each of the predicates \((p_1, p_2, \ldots)\) is evaluated until one is true \((\neq \#f)\). Then the corresponding expression \((e_1, e_2, \ldots)\) 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

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

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

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:

```scheme
(define
  (append L1 L2)
  (if (null? L1)
      L2
      (cons (car L1)
         (append (cdr L1) L2)))
)
```
Let’s trace `(append ' (a b) ' (c d))

Our definition is

```scheme
(define
  (append L1 L2)
  (if  (null? L1)
      L2
      (cons (car L1)
          (append (cdr L1) L2))
  )
)
```

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

`(null? L1)` is false, so we evaluate

```scheme
(cons (car L1) (append (cdr L1) L2))
= (cons (car ' (a b))
          (append (cdr ' (a b)) ' (c d)))
= (cons 'a (append ' (b) ' (c d))
```

We need to evaluate

```scheme
(append ' (b) ' (c d))
```

In this call, `L1 = (b)` and `L2 = (c d)`.

`L1` is not null, so we evaluate

```scheme
(cons (car L1) (append (cdr L1) L2))
= (cons (car ' (b))
          (append (cdr ' (b)) ' (c d)))
```
= (cons 'b (append '() '(c d))

We need to evaluate

(append '() '(c d))

In this call, \( L_1 = () \) and \( L_2 = (c \ d) \).

\( L_1 \) is null, so we return \( (c \ d) \).

Therefore

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

Finally,

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

Note:

Source files for \text{append}, and other Scheme examples, may be found in

\(~\text{cs538-1/public/scheme/example1.scm},\)
\(~\text{cs538-1/public/scheme/example2.scm},\)
etc.
Reversing a List

Another useful list-manipulation function is \texttt{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,

\[
\texttt{(rev '(1 2 3))} \quad \Rightarrow \quad \texttt{(3 2 1)}
\]

The definition of \texttt{rev} is straightforward:

\[
\begin{align*}
(\text{define} & \ (\texttt{rev} \ \texttt{L}) \\
& \quad (\text{if} \ (\text{null?} \ \texttt{L}) \\
& \quad \quad \texttt{L} \\
& \quad \quad (\text{append} \ (\text{rev} \ (\text{cdr} \ \texttt{L})) \\
& \quad \quad \quad (\text{list} \ (\text{car} \ \texttt{L}))) \\
& \quad ) \\
& )
\end{align*}
\]
As an example, consider

\((\text{rev } '(1 2))\)

Here \(L = (1 2)\). \(L\) is not null so we evaluate

\((\text{append } (\text{rev } (\text{cdr } L))\)

\((\text{list } (\text{car } L))) = \)

\((\text{append } (\text{rev } '(1 2)))\)

\((\text{list } (\text{car } '(1 2)))) = \)

\((\text{append } (\text{rev } '(2)) (\text{list } 1)) = \)

\((\text{append } (\text{rev } '(2)) '(1))\)

We must evaluate \((\text{rev } '(2))\)

Here \(L = (2)\). \(L\) is not null so we evaluate

\((\text{append } (\text{rev } (\text{cdr } L))\)

\((\text{list } (\text{car } L))) = \)

\((\text{append } (\text{rev } '(2)))\)

\((\text{list } (\text{car } '(2)))) = \)

\((\text{append } (\text{rev } ())(\text{list } 2)) = \)

\((\text{append } (\text{rev } ()')'(2))\)

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 \text{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.
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,

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

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

\[
\begin{align*}
(\text{define } (\text{revall} \ L)) \\
(\text{if } (\text{null?} \ L) \\
\quad L \\
(\text{let } ((E (\text{if } (\text{list?} \ (\text{car} \ L)) \\
\quad (\text{revall} \ (\text{car} \ L)) \\
\quad (\text{car} \ L)))) \\
(\text{append} \ (\text{revall} \ (\text{cdr} \ L)) \\
\quad (\text{list} \ E)))
\end{align*}
\]

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