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,
(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 (≠ #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:

\[
\begin{align*}
\text{(define } & (\text{append } L1 \ L2) \\
& \text{(if } (\text{null? } L1) \\
& \quad \text{L2} \\
& \quad \text{(cons } (\text{car } L1) \\
& \quad \quad \text{(append } (\text{cdr } L1) \ L2)\text{))} \\
& ))
\end{align*}
\]

Let’s trace \((\text{append } '(a \ b) \ '(c \ d))\)

Our definition is

\[
\begin{align*}
\text{(define } & (\text{append } L1 \ L2) \\
& \text{(if } (\text{null? } L1) \\
& \quad \text{L2} \\
& \quad \text{(cons } (\text{car } L1) \\
& \quad \quad \text{(append } (\text{cdr } L1) \ L2)\text{))} \\
& ))
\end{align*}
\]

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

\((\text{null? } L1)\) is false, so we evaluate

\[
\begin{align*}
\text{(cons } \text{(car } & L1) \\
& \text{(append } \text{(cdr } L1) \ L2)\text{)} \\
& \text{(cons } \text{(car } & \text{(append } (\text{cdr } (a \ b)) \ '(c \ d))\text{)} \\
& \text{(cons } 'a \text{ (append } & \text{(b) } '(c \ d)\text{)} \\
\] \text{We need to evaluate}

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

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

\(L1\) is not null, so we evaluate

\[
\begin{align*}
\text{(cons } \text{(car } & L1) \\
& \text{(append } \text{(cdr } L1) \ L2)\text{)} \\
& \text{(cons } \text{(car } & \text{(append } (\text{cdr } (b)) \ '(c \ d))\text{)} \\
\]

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,

\[
\begin{align*}
\text{(rev } & '(1 \ 2 \ 3) \Rightarrow (3 \ 2 \ 1) \\
\text{The definition of } & \text{rev is straightforward:}
\end{align*}
\]

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

Note:
Source files for `append`, and other Scheme examples, may be found in

~cs538-1/public/scheme/example1.scm,
~cs538-1/public/scheme/example2.scm,
etc.
As an example, consider

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

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

\[
\text{(append \ (rev \ (cdr \ L)) \ (list \ \text{car \ L}))} =
\]

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

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

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

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

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

\[
\text{(append \ (rev \ (cdr \ L)) \ (list \ \text{car} \ L))} =
\]

\[
\text{(append \ (rev \ '(2)) \ (list \ \text{car} \ '(2)))} =
\]

\[
\text{(append \ (): (list \ 2))} =
\]

\[
\text{(append \ (): (2))} =
\]

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

Here \( L = () \). \( L \) is null so \( \text{(rev '())} = () \)

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

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

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

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

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

As constructed, 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 rev to also reverse list members that happen to be lists.

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