Conditional Expressions in Scheme

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

For example,

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

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

The `if` expression is

```scheme
(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 \((p_1, p_2, \ldots)\) 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

(cons '(a b) '(c d)) ⇒
((a b) c d)

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

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

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

We need to evaluate
\[
\begin{align*}
\text{(append } '(b)\ ' (c\ d))
\end{align*}
\]

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

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

**We need to evaluate**

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

**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,**

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

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

\[ (c \ a (b \ c \ d)) = (a \ b \ c \ d) \]

**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.
Reversing a List

Another useful list-manipulation function is \( \text{rev} \), 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 \( \text{rev} \) is straightforward:

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