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
(cons '(a b) '(c d)) \(\Rightarrow\)
((a b) c d)
(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 `(append '(a b) '(c d))`
Our definition is
(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
(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
(append '(b) '(c d))
In this call, L1 = (b) and L2 = (c d).
L1 is not null, so we evaluate

(c 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, L1 = () and L2 = (c d).
L1 is null, so we return (c d).
Therefore
(cons 'b (append '() '(c d)) =
(cons 'b '(c d)) = (b c d) =
(append 'b '(c d))
Finally,
(append '(a b) '(c d)) =
(cons 'a (append '(b) '(c d)) =
(cons 'a '(b c d)) = (a b c d)

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,
```scheme
(rev '(1 2 3)) \(\Rightarrow\) (3 2 1)
```
The definition of `rev` is straightforward:
(define (rev L)
  (if (null? L)
      L
      (append (rev (cdr L))
              (list (car L))
  )))
Note:
Source files for `append`, and other Scheme examples, are in
~cs538-1/public/scheme/example1.scm,
~cs538-1/public/scheme/example2.scm,
etc.
As an example, consider

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

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

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

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

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

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

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 \textit{let} construct.

\section*{The Let Construct}

Scheme allows us to create local names, bound to values, for use in an expression.

The structure is

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

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

For example,

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

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

\[
(\text{define } (\text{revall } L)
(\text{if } (\text{null? } L)
L
(\text{let } ((\text{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 } \text{E}))))
)
)
(\text{revall } '(1 2) (3 4))) \Rightarrow
((4 3) (2 1)) \)
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\)
\[
  ( () \ (1) \ (2) \ (1 2)), \text{ which contains 4 subsets.}
\]

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

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) =\)
\[
  (3) \ (3 1) \ (3 2) \ (3 1 2)
\]

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

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\)
\[
  ( () \ (a) \ (b) \ (b a))
\]

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

Now \text{subsets} is easy:

\[\text{(define (subsets } L)\]
\[
  (if \ (null? \ L) \ ()
  \ (list \ (extend \ (subsets \ (cdr \ L)) \ (car \ L)))
  )
\]
**Data Structures in Scheme**

In Scheme, lists and S-expressions are basic. Arrays can be simulated using lists, but access to elements “deep” in the list can be slow (since a list is a linked structure). To access an element deep within a list we can use:

- **(list-tail L k)**
  This returns list L after removing the first k elements. For example,
  \[ (\text{list-tail '}(1 \ 2 \ 3 \ 4 \ 5) \ 2) \Rightarrow (3 \ 4 \ 5) \]

- **(list-ref L k)**
  This returns the k-th element in L (counting from 0). For example,
  \[ (\text{list-ref '}(1 \ 2 \ 3 \ 4 \ 5) \ 2) \Rightarrow 3 \]