Recursion in Scheme

Recursion is widely used in Scheme and 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 **`(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

```
(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)
```

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 **`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))` $\Rightarrow$ **`(3 2 1)`**

The definition of **`rev`** is straightforward:

```
(define (rev L)
  (if (null? L)
      L
      (append (rev (cdr L))
              (list (car L)))
      )
  )
)
```

As an example, consider

`(rev '(1 2))`

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

```
(append (rev (cdr L))
                 (list (car L))) =
(append (rev (cdr '(1 2)))
                 (list (car '(1 2)))) =
(append (rev '(2)) (list 1)) =
(append (rev '(2)) '(1))
```

We must evaluate **`(rev '(2))`**

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

```
(append (rev (cdr L))
                 (list (car L))) =
(append (rev (cdr '(2)))
                 (list (car '(2)))) =
(append (rev ())(list 2)) =
(append (rev ())'(2))
```

We must evaluate **`(rev '())`**

Here **`L = ()`**. **`L`** is null so **`(rev '())= ()`**

Thus `(append (rev ())'(2)) =`
`(append  () '(2)) = (2) = (rev '(2))`

Finally, recall `(rev '(1 2)) =`
`(append (rev '(2)) '(1)) =`
`(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,

```
(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.

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,

`(let ( (a 10) (b 20) )`
`(+ a b))` $\Rightarrow$ `30`

Using a `let`, the definition of `revall`, a version of `rev` that reverses all levels of a list, is easy:

```
(define (revall L)
  (if (null? L)
      L
      (let ((E (if (list? (car L))
                   (revall (car L))
                   (car L) )))
        (append (revall (cdr L))
                (list E))
      )
  )
)
```

`(revall '( (1 2) (3 4)))` $\Rightarrow$
`((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,

```
(subsets  '(1 2 3)) ⇒
  ( () (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,

`(subsets '(1 2) )` ⇒
`( () (1) (2) (1 2))`, which contains 4 subsets.

`(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 *plus* the result of “distributing” the new value into each of the original subsets.

Thus `(subsets '(1 2 3))` ⇒

```
( () (1) (2) (3) (1 2) (1 3)
  (2 3) (1 2 3)) =
( ()   (1)   (2)   (1 2) )  plus
( (3) (1 3) (2 3) (1 2 3) )
```

This insight leads to a concise program for subsets.

We will let `(distrib L E)` be a function that "distributes" `E` into each list in `L`.

For example,

```
(distrib '(() (1) (2) (1 2)) 3) =
( (3) (3 1) (3 2) (3 1 2) )
```

```
(define (distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L))
            (distrib (cdr L) E))
   )
)
```

We will let **(extend L E)** extend a list **L** by distributing element **E** through **L** and then appending this result to **L**.

For example,

```
(extend '( () (a) ) 'b) ⇒
( () (a) (b) (b a))
```

```
(define (extend L E)
  (append L (distrib L E))
)
```

Now **subsets** is easy:

```
(define (subsets L)
  (if (null? L)
      (list ())
      (extend (subsets (cdr L))
              (car L))
  )
)
```