

In LISP, unlike most conventional programming languages, the primary method used in program development is recursion. Recursion can and is efficiently implemented in most LISP environments. Recursion occurs naturally in many mathematical and digital filtering applications and is "human-efficient." Consider the requirement to program a function to return t (non-Nil) if a Sex is a member of a given Set as an example.

- a. Trivial Case 1: A Sex cannot be in any Set if the Set is Nil (empty)
- b. Trivial Case 2: A Sex is a member of the Set if it is EQ to the 1st element
- c. Otherwise we must search the rest of the Set

This leads us to the following direct implementation below:

```
(defun memset (Sex Set) (cond
  ( (null Set)          nil)
  ( (eq Sex             (car Set)) t)
  ( t                  (findit Sex (cdr Set))) ))
(defun findit (Sex Rest) (memset Sex Rest))
```

We immediately notice two possible improvements to the code above, mainly, that function findit wants to check to see if the Sex is a member of a different set called Rest. There are no restrictions on how findit needs to do this. In particular, findit can call on memset to do it. But if this is so, then we might as well replace the call to findit in memset to a call to memset itself. This is called recursion and it poses no particular restrictions in LISP. Also, since t is any non-nil expression, we might as well replace the t in the second clause of memset to be something that we can use that is not nil. Well, we know that if clause 2 is considered Set is not nil (clause 1 takes care of this), so we can use Set rather than t. These improvements produce the code below.

```
(defun memset (Sex Set) (cond
  ( (null Set)          nil)
  ( (eq Sex (car Set))  Set)
  ( t                  (memset Sex (cdr Set))) ))
```

Let us try some other examples. Try a function hasand (on our way to a function has) that checks to see if the atom AND occurs in a list. We could implement it as shown below:

- a. Check to see if the list, say Lis, is null, if so there is no AND in it
- b. Check to see if the list is exactly a 1 element list (AND)
- c. Check to see if the list has > 1 element, if so is it's 1st element an AND?
- d. Check the rest of the list to see if the AND is there

```
(defun poorhasand (Lis) (cond
  ( (null Lis)          nil )
  ( (null (cdr Lis))    (eq (car Lis) 'and))
  ( (eq (car Lis) 'and) t )
  ( t                  (poorhasand (cdr Lis))))
```

Poorhasand is poor because clause 2 (Case b.) is not really necessary. Notice how clause 3 (Case c.) handles clause 2 anyway. We say clause 3 subsumes clause 2. We reprogram poorhasand into hasand.

```
(defun hasand (Lis) (cond
  ( (null Lis)                nil )
  ( (eq (car Lis) 'and)       t )
  ( t                          (hasand (cdr Lis)) ) ) )
```

But we might as well replace the literal occurrences of the atom AND in hasand to be any atom we might be interested in, say Atm. This leads us to *has*.

```
(defun has (Atm Lis) (cond
  ( (null Lis)                nil )
  ( (eq (car Lis) Atm)        t )
  ( t                          (has Atm (cdr Lis)) ) ) )
```

Do you see that has and memset are identical! Recursive programming in LISP has many advantages to the human and for AI purposes. However, let me share some hints that have proven invaluable over the years. Look at the recursive function pattern below:

```
(defun recursive_function (Args) (cond
  ( <handle the required trivial case> )
  ( <handle any other trivial case(s) if any> )
  ( t <the recursive call with a subproblem, the n-1 case> ) ) )
```

First, there is always a trivial ( $n=0$  or 0th case) in recursive definitions of functions. For example, factorial can be defined as  $n! = n * (n-1)!$  if and only if we have a limiting case. For factorial, we define  $0! = 1$ . In LISP, we want to make sure that we are doing legal operations on lists (for example, if you don't know something is a list, you don't know if car is a legal operation), thus, we almost always check for the null list as the limiting case. Other limiting cases might be atoms, etc. Please note that nil or () is both an atom and a list! Almost always, we see the first clause of a recursive program is a check for the null list or an atomic occurrence of something.

Second, we always pass a smaller problem (a subproblem, a smaller set, a smaller list, etc.) to the recursive call. This is important! In the list domain, we see functions such as cdr which eventually break down a list all the way to nil. Note that nil is always (implicitly) the last element of any list (we may call this reducing the list to an empty list). Care should be taken to consider lists that have nil as an explicit element besides the implicit nil terminator.

Third, we need to be aware of the fact that recursion can be made VERY efficient by the use of additional arguments and so-called helper functions. We will consider this when we discuss set operations below.

Before leaving this discussion, we might notice the fact that many built-in LISP functions can be described using a few primitive operations such as null, atom, cond, defun, car, cdr, cons, eq, etc. Let us consider a few examples.

(list Sex) <==> (cons Sex ())

(list Sex1 Sex2) <==> (cons Sex1 (cons Sex2 nil))

(append L1 L2) <==> (defun append (L1 L2) (cond  
                           ( (null L1)                          L2)  
                           (       t          (cons (car L1) (append (cdr L1) L2)))) ))

(equal Sex1 Sex2) <==> (defun equal (Sex1 Sex2) (cond  
                           ( (atom Sex1)                  (eq Sex1 Sex2) )  
                           ( (atom Sex2)                  nil     )  
                           ( (equal (car Sex1) (car Sex2))  
   (equal (cdr Sex1) (cdr Sex2)))) ))

(length Lis) <==> (defun length (Lis) (cond  
                           ( (null Lis)                      0 )  
                           (       t          (+ 1 (length (cdr Lis)))) ))

Let us try some simple set operations. The union of two sets is defined as the set in which every element of the union is a member of either the 1st or the 2nd set. This leads us to the following recursive definition.

(defun union0 (S1 S2) (cond  
   ( (null S1)  S2)  
   ( (memset (car S1) S2)          (union0 (cdr S1) S2) )  
   (       t          (union0 (cdr S1) (cons (car S1) S2)))) ))

But union0 is VERY inefficient. Why? We know that (car S1) is not a member of S2 (which is checked in clause 2) and we know (car S1) is not a member of the (cdr S1) or it would not be a set (by definition). Thus, in clause 3 we should not be setting up a check (in the subsequent recursive call) to see if the 1st element of the (cdr S1) matches the new S2 which now has the original 1st element of S1 CONSed to it. We can improve it as follows:

```
(defun union1 (S1 S2) (cond
  ( (null S1) S2)
  ( (memset (car S1) S2) (union1 (cdr S1) S2) )
  ( t (cons (car S1) (union1 (cdr S1) S2))))))
```

Another way is to use helper functions with additional arguments. Look at the implementation of union below.

```
(defun union (S1 S2) (helpunion S1 S2 S2) )

(defun helpunion (S1 Partial S2) (cond
  ( (null S1) Partial)
  ( (memset (car S1) S2) (helpunion (cdr S1) Partial S2) )
  ( t (helpunion (cdr S1) (cons (car S1) Partial) S2)) ))
```

There is both an increase in human efficiency in the above example and an increase in computer efficiency. To close, consider intersection of two sets. The intersection of two sets S1 and S2 is defined to be the set whose elements are both in S1 and S2. this leads to the following implementation.

```
(defun inter (S1 S2) (cond
  ( (null S1) nil)
  ( (memset (car S1) S2) (cons (car S1) (inter (cdr S1) S2)))
  ( t (inter (cdr S1) S2)) ))
```