Using APPLY to Apply a Function to a List of Arguments

APPLY is also a Lisp primitive function. APPLY takes a function and a list of objects as input. It invokes the specified function with those objects as its inputs.

From sec. 3.21 of Touretzky

```
(apply #'+ '(2 3)) ⇒ 5

(apply #'equal '(12 17)) ⇒ nil
```

The objects APPLY passes to the function are *not* evaluated first. In the following example, the objects are a symbol and a list. Evaluating either the symbol AS or the list (YOU LIKE IT) would cause an error.

```
(apply #'cons '(as (you like it)))
  ⇒ (as you like it)
```

Example: Use APPLY to write the SUM function

A. SUM is a function that is already defined on venus and euclid; if L *is any list of numbers* then (SUM L) returns the sum of the elements of L. [Thus (SUM ( )) returns 0.] Complete the following definition of a

of Lisp Assignment 4 without using recursion.

Solution: `(defun sum (L) (apply #'+ L))`

If f => a function and L => a list, then

$$(\text{APPLY } f\ e_1 \dots\ e_n\ L)$$

is evaluated by calling the function with the values of $e_1, \dots, e_n$ as the first n arguments and the elements of the list as the remaining arguments.

The result returned by the function is returned by APPLY as its own result.

```
   (apply #'+ 2 20 200 2000 nil)
 = (apply #'+ 2 20 200 '(2000))
 = (apply #'+ 2 20 '(200 2000))
 = (apply #'+ 2 '(20 200 2000))
 = (apply #'+ '(2 20 200 2000)) = (+ 2 20 200 2000) => 2222

   (apply #'mapcar #'+ '((1 2 3) (10 20 30) (100 200 300)))
 = (mapcar #'+  '(1   2   3)
                '(10  20  30)
               '(100 200 300))

             => (111 222 333)
```

This example is relevant to problem 16(c) of Lisp Assignment 5!

Tail Recursive Functions and Tail Recursion Optimization

A call of a function F is said to be a ***tail call*** if, when the call of F returns control to the calling function, the calling function ***immediately*** returns control to its own caller and returns as ***its own*** result any value that was returned by the call of F.

- A recursive call that is also a tail call is said to be a ***tail recursive*** call.
- A recursive function is said to be ***tail recursive*** if ***every*** recursive call it makes is a tail call.

The Concept of Tail Recursion is ***Not*** Specific to Lisp or Functional Programming

Java Examples:

```
static long f(int n, long r) // Returns n!*r when n >= 0
{                            //              and n!*r < 2^63
  if (n > 1) return f(n-1, n*r);   // A tail recursive call.
  else return r;
} Comment: This function works because  n! * r = (n-1)! * n*r
```

An example of Tail Recursion in *Imperative* Programming:

```
static void reverseArray(int[] A, int i, int j)
// Reverses the subarray A[i .. j] of the array A[].
{
  if (i < j) {
    swap(A, i, j);                // Swap values in A[i] and A[j].
    reverseArray(A, i+1, j-1); // A tail recursive call
  }
}
```

Examples of Recursive Calls That are NOT Tail Recursive:

```
static void reverseArray(int[] A, int i, int j)
// Reverses the subarray A[i .. j] of the array A[].
{
  if (i < j) {
    reverseArray(A, i+1, j-1); // NOT a tail recursive call:
    swap(A, i, j);             // Swap is performed after call.
  }
}

static long f(int n) // returns n! when n >= 0
{                    //           and n! < 2^63
  if (n > 1) return n * f(n-1); // NOT a tail recursive call:
                                // * is performed after call
  else return 1;
}
```

A Compiler May Do *Tail Recursion Elimination**

*also called ***tail recursion optimization***

This replaces a ***tail recursive*** call with code that:

1. Sets ***each formal parameter*** of the caller to the value of the corresponding actual argument of the tail recursive call.
2. Executes a jump that transfers control to the start of the body of the caller.

C++ Example to Illustrate this Transformation:

```
long f(int n, long r) {
  if (n > 1) {
    return f(n-1, n*r);
  }
  else return r;
}
```

⇒

```
long f(int n, long r) {
  START: if (n > 1) {
           r = n*r;
           n = n-1;
           goto START;
         }
         else return r;
}
```

Note: The code on the right updates r ***before*** updating n, as the new value of r (i.e. n*r) depends on n.

RECALL: A call of a function F is said to be a ***tail call*** if, when the call of F returns control to the calling function, the calling function ***immediately*** returns control to its own caller and returns as ***its own*** result any value that was returned by the call of F.

The following function definition contains 2 tail calls:

From p. 259 of Touretzky

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

The call of `cons` ***is a tail call***, and the ***2nd*** recursive call of `extract-symbols` ***is a tail call***.

But the calls of `null`, `symbolp`, `first`, and `rest`, and the ***1st*** recursive call of `extract-symbols` ***are not tail calls***!

RECALL: A recursive call that is also a tail call is said to be a ***tail recursive*** call.

RECALL: A recursive function is said to be ***tail recursive*** if ***every*** recursive call it makes is a tail call.

Touretzky says this about such functions on p. G-14:

tail recursive

TYK

A function is tail recursive if it does all its work before making each ~~the~~ recursive call. Tail recursive functions return the result of each ~~the~~ recursive call without augmenting (modifying) it, or doing any other additional work. Clever Lisp compilers turn tail recursive calls into jump instructions, eliminating the need for a call stack.

The above function

From p. 259 of Touretzky

```
(defun extract-symbols (x)
  (cond ((null x) nil)
        ((symbolp (first x))
         (cons (first x)
               (extract-symbols (rest x))))
        (t (extract-symbols (rest x)))))
```

NOT a tail call!

is ***not*** a tail recursive function, because it sometimes makes a recursive call that is ***not*** a tail call.

Tail Recursion Elimination May Be Necessary to Limit Memory Use When Depth of Recursion is High

The usual way to execute a function call written in a language that supports recursion involves ***allocating memory on a stack*** (to store values of parameters and other local variables of the called function, and to store the contents of the program counter and other registers that are in use at the time of the call when executing compiled code). **The allocated memory *will only be deallocated when the called function returns control to its caller.***

- This results in ***stack overflow*** if the depth of recursion is too great!

The Clisp *compiler* does tail recursion elimination for *all* tail recursive calls, which ***eliminates the need to allocate memory for such calls***: For ***compiled*** tail recursive Clisp functions, there's no limit on recursion depth.

Example of Stack Overflow When Recursion is Too Deep

Write a function COUNTDOWN-FROM such that:

If n => a non-negative integer, then

(COUNTDOWN-FROM n) => a list of the integers from n down to 0.

Thus (countdown-from 10) => (10 9 8 7 6 5 4 3 2 1 0)

```
[1]> (defun countdown-from (n)
       (if (zerop n)
           '(0)
           (cons n (countdown-from (- n 1)))))
COUNTDOWN-FROM
[2]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
[3]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
[4]> (compile 'countdown-from)
COUNTDOWN-FROM ;
NIL ;
NIL
[5]> (length (countdown-from 20000))
20001
[6]> (length (countdown-from 50000))
50001
[7]> (length (countdown-from 100000))

*** - Program stack overflow. RESET
[8]>
```

This depth of recursion is too great for the clisp interpreter!

In addition to running faster, *compiled* code uses less stack stack space for function calls than interpreted code: So the recursion depth can be greater.

But recursion depth is still limited, because COUNTDOWN-FROM is ***not*** tail recursive!

```
[1]> (defun countdown-from-aux (hi lo accumulator)
       (if (< hi lo)
           accumulator
           (countdown-from-aux hi (+ lo 1) (cons lo accumulator))))
```

This is a *tail recursive* helping function.

```
COUNTDOWN-FROM-AUX
[2]> (countdown-from-aux 50 43 '(the cat sat))
(50 49 48 47 46 45 44 43 THE CAT SAT)
```

It returns the result of appending (hi ... lo) to the accumulator list.

```
[3]> (defun countdown-from (n) (countdown-from-aux n 0 nil))
COUNTDOWN-FROM
[4]> (countdown-from 10)
(10 9 8 7 6 5 4 3 2 1 0)
```

This definition of COUNTDOWN-FROM can take advantage of tail recursion elimination!

```
[5]> (length (countdown-from 20000))

*** - Program stack overflow. RESET
```

__Before__ COUNTDOWN-FROM-AUX is compiled, there's no tail recursion elimination and so recursion depth is limited!

```
[6]> (compile 'countdown-from-aux)
COUNTDOWN-FROM-AUX ;
NIL ;
NIL
[7]> (length (countdown-from 20000))
20001
[8]> (length (countdown-from 200000))
200001
[9]> (length (countdown-from 2000000))
2000001
[10]> (length (countdown-from 20000000))
20000001
[11]>
```

__After__ COUNTDOWN-FROM-AUX is compiled, **its recursion depth is no longer limited by the size of the stack**, as the compiler has performed ***tail recursion elimination***.

Differences Between Scheme and Common Lisp That are Relevant to Functions That Take Functions as Arguments

In Common Lisp, if F is the name of a certain function then the value of #’F is that same function, but F itself may have any value or no value. Moreover:

- (DEFUN F ...) makes F the name of a function but does ***NOT*** affect the ***value*** (if any) of F.
- If F is the name of a certain function, then (F ...) calls that function.
- If the ***value*** of F is a certain function, then (FUNCALL F ...) calls that function.

In Scheme, the ***value*** of F is a function ***just if*** F is the name of that same function. Accordingly:

- (FUNCALL F ...) is NOT used: A Common Lisp call (FUNCALL F ...) is written as (F ...) in Scheme.
- #’F is NOT used: A Scheme programmer would write F where a Common Lisp programmer writes #’F.