LISP Examples

1) Write a function *filter* which takes a list and a predicate, and returns the list of the elements from the original list for which the predicate returns true. (There are actually LISP built-ins to do this called *remove-if* and *remove-if-not*t. Of course you may not use them for this problem!)

```
? (defun even(num) (= (mod num 2) 0))
? (filter '(6 4 3 5 2) #'even)
(6 4 2)
```

2) Write a function *non-nil* which takes a list and replaces each non-null element with 1 and each null element with 0. Write it twice – once recursively and once using *mapcar*. Bonus problem: write an expression that counts the number of non-null elements in a list.

```
? (non-nil '(a nil (b) (nil) 2))
(1 0 1 1 1)
```

3) Assume you have a function `(nums start stop)` which returns a list of the numbers in the range start..stop inclusive. Write a boolean function *forall* which takes a list and a predicate and returns true if and only if the predicate returns true for every element in the list. So the predicate must be true for all the elements in the list. A predicate is a one-argument function that returns true or false.

```
? (forall '(2 4 6 12) #'even)                ; Is every element a number?
T
? (forall '(1 2 (3) 4) #even)
NIL
? (forall '(nil nil nil nil) #'null)         ; Is every element null?
T
? (forall '(nil definitely-not nil)  #'null)
NIL
```

4) (One of Nick's all time favorite old CS107 final exam problems) Write a function `(prime n)` which takes a positive integer and returns true if the number is prime. A number is prime if it is not divisible by any number in the range `2..(floor (sqrt n))`. `Sqrt` takes the square root, and `Floor` rounds down. Use the answer to `forall` function above which you may assume is correct.

5) Write an expression using `nums`, `filter`, and `prime` which is the list of prime numbers in the range 1..100.

6) Write a function *increasing-sum* which takes a non-empty list of non-empty lists of numbers, such as `'((2 4 3) (6 2) (5 6 7 3))`. The function should return a list where the sublists are arranged in increasing order by their sum.

```
?(increasing-sum '((2 4 3) (6 2) (5 6 7 3)))
 ((6 2) (2 4 3) (5 6 7 3))
```

Most

The Most utility function is a general purpose utility. It takes a non-empty list and two argument comparator function (as sort does) and returns the element in the list which was "most" according to the comparator.

```
; MOST
; Takes a non-empty list and two argument comparator function which
; returns T if its first argument was "more than" its second argument.
; Returns the element in the list which was "most" according to the
; comparator.
(defun most (list comparator) ...

? (most '(3 46 7 -72 6 -8) #'>)
46
? (most '(3 46 7 -72 6 -8) #'(lambda (x y) (> (* x x) (* y y))))
-72
```

7) Write a function *increasing-max* which takes the same sort of list as above, but orders by the maximum number in each list. You may use the `most` function.

```
? (increasing-max '((2 4 3) (6 2) (5 6 7 3)))
((2 4 3) (6 2) (5 6 7 3))
```

SOLUTIONS TO FUNCTION QUESTIONS

```
1.  (defun filter(list predicate)
      (if (null list) '()
          (let (
                 (the-rest (filter (cdr list) predicate))
                )
            (if (funcall predicate (car list))
              (cons (car list) the-rest)
              the-rest
            )
          )
      )
    )

2.  (defun non-nil (list)
      (if (null list)
        '()
        (cons
          (if (null (car list))
            0
            1)
          (non-nil (cdr list)))))

    (defun non-nil (list)
      (mapcar #'(lambda (elem)
                  (if (null elem)
                    0
                    1))
              list))
3.
(defun forall (list func)
  (if (null list)
    T
    (and (funcall func (car list))
         (forall (cdr list) func))))

4.
(defun prime(n)
  (and
   (> n 1)       ;;picky case so we don't say numbers <=1 are prime
   (forall (nums 2 (floor (sqrt n)))
           #'(lambda (divisor) (not (= (mod n divisor) 0)))
           ))))

5. this expression yields all the prime numbers in the range 1..100  YEAH!!
?(filter (nums 1 100) #'prime)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)

6.
(defun increasing-sum(list)
  (sort list #'(lambda (sublist1 sublist2)
                 (< (apply #'+ sublist1 ) (apply #'+ sublist2)))))

7.
(defun increasing-max(list)
  (sort list #'(lambda (sublist1 sublist2)
                 (< (most sublist1 #'>) (most sublist2 #'>))))
)
```

Power Set Example

Pro
Very expressive and terse language leads to quick expression of a complex algorithm.

Con
The code is so dense, it can't be read quickly.

```
;; Computes the power-set of any set.
;; The power set of a set X, is the set of all possible
;; subsets of X. Sets are represented by lists, so this
;; takes a list and returns a list of lists.
;;  ? (power-set '(1 2))
;;  ((1 2) (1) (2) NIL)
;;  ? (power-set '(a b c))
;;  ((A B C) (A B) (A C) (A) (B C) (B) (C) NIL)
;; Implementation: recursively compute the power set of the cdr.
;; For each of those sets, include two versions in the final power set-
;; one which includes the car and one which does not.
(defun power-set(set)
  (if (null set) '(())
      (let ((psetOfRest (power-set (cdr set))))
        (append

         ;;add the car into the subsets
         (mapcar #'(lambda (subset) (cons (car set) subset)) psetOfRest)

         ;;take the subsets without adding the car
         psetOfRest
         )
        )
      )
  )
```