Variable binding. Recursion

In LISP the arguments of a procedure are transmitted through value (like in C). At the entrance of the procedure, the parameters are bounded to arguments forming bounded variables. At the exist of the procedure, the bounded variables get the former values. Variables which are used in a procedure but aren't parameters are named regarding to the procedure free variables and their values is lexical determined.

LET Primitive

LET binds variables and gives them values. (SETQ only gives the calues)
Sintac

```
(let ((<param 1> <initial value 1>)
      (<param 2> < initial value 2>)
      (<param n> < initial value n>))
      ...
      <body of let>)
```

Binding is only valid in the body of let.
LET computes al former values before the binding (the attribution is made in parallel, SETQ attributes during the evaluation, step by step).
LET* binds the variables but attributes the values sequentially (like the procedure SET).
SETQ has the variable PSETQ which attributes the values in parallel.

SETQ andLET – sequential and parallel

```
(SETQ a 'a b 'b c 'c d 'd)
D
(LET ((a 'alfa) (b 'beta) (ab (LIST a b))) ab) ;LET parallel
(A B)
(LET* ((c 'gamma) (d 'delta) (cd (LIST c d))) cd) ;LET
sequential
(GAMMA DELTA)
(SETQ a 'alfa b 'beta ab (LIST a b)) ;SETQ sequential
(ALFA BETA)
ab
(ALFA BETA)
(PSETQ c 'gamma d 'delta cd (LIST c d)) ;SETQ parallel
nil ; PSETQ always returns nil
cd
(C D)
```

FUNCALL Primitive

FUNCALL allows the indirect call of the functions.

FUNCALL

```
(setq operation '+)
+
(funcall operation 2 3 4)
9
(setq operation '*)
*
(funcall operation 2 3 4)
24
```

Recursion

When a procedure calls itself, directly or indirectly, for solving a part of the problem, we encounter recursion.

Computing of `m^n` (`n` integer) can be recursively made using the following formula:

```
        / m * m^(n-1), for n > 0
m^n = {
        \ 1, for n = 0
```

The project implemented in LISP is:

Exponential

```
(defun our-expt (m n)
      (cond ((zerop n) 1)                      ; n=0?
            (t (* m (our-expt m (- n 1))))))   ; recursive call
```

Problem 1

Implement recursive the factorial function.

Problem 2

Implement recursive the Fibonacci sequence.

Problem 3

Implement recursive the function `member`, which test if an element is a member of a list

Problem 4

Implement the function `(trim-head l n)` which removes from list l the first n elements.

Problem 5

Implement the function `(trim-tail l n)` which removes from list l the last n elements

Problem 6

Implement `(count-atoms l)` which counts all atoms of list l, including the ones from nested lists.

Problem 7

Saying that + and – can only be used for incrementing or decrementing a number with one (using the procedures inc and dec), write a recursive procedure for adding two positive integers.

Problem 8

Implement the procedure `reverse`.

Problem 9

Implement the predicate `presentp` which indicates if an atom appears inside a list

Problem 10

Implement the procedure `squash` which receives a generated list and returns a list of all found atoms.