RECURSION

In pure Lisp there is no looping; recursion is used instead.

A recursive function is defined in terms of:
1. One or more base cases
2. Invocation of one or more simpler instances of itself.

Note that recursion is directly related to mathematical induction.

An inductive proof has:
1. A basis clause
2. A hypothesis that the theorem is true for some number K
3. An inductive clause that shows it is then true for K+1.

\[ N! = N \times (N-1) \times \ldots \times 1 = N(N-1)! \]

```lisp
(defun factorial (N)
  (cond ((eql N 0) 1)
        ((eql N 1) 1)
        (T (* N (factorial (- N 1))))))
```

(factorial 3)

\[ N! = N \times (N-1) \times \ldots \times 1 = N(N-1)! \]

Can you prove by induction that \((\text{factorial } N)\) produces \(N!\) for \(N \geq 0\)?
**PROOF**

**Basis:**
- (factorial 0) produces 1
- (factorial 1) produces 1
by definition of the first two cases in the cond.

**Hypothesis:** Suppose (factorial K) correctly returns K!, K > 1

**Inductive Part:**
Then (factorial < K + 1>) evaluates to
(* < K + 1 > (factorial (- < K + 1 > 1)))

which will be
(* < K + 1 > (factorial K))
or (K + 1) * K! = (K + 1)!

---

**Doubling the Values of All Elements of a List**

\[
\text{(defun double (x))}
\]

(if (null x) NIL
  (cons (* (first x) 2)
    (double (rest x)))
)

(double '(2 0 6))
(cons 4 (double '(0 6)))
  (cons 0 (double '(6)))
    (cons 12 (double NIL))

(4 0 12)

---

**Summing a List**

\[
\text{(defun sum (L))}
\]

(if (null L) 0
  (+ (first L) (sum (rest L))))


**FIBONACCI FUNCTION**

\[
\begin{align*}
f(n) &= \begin{cases} 
  f(n - 1) + f(n - 2) & n > 1 \\
  1 & n = 1 \\
  1 & n = 0
\end{cases}
\end{align*}
\]

(defun fibonacci (N)
  (if (or (= n 0) (= n 1))
    1
    (+ (fibonacci (- n 1))
        (fibonacci (- n 2)))))

(fibonacci 4)
(+ (fibonacci 3) (fibonacci 2))
(+ (fibonacci 2) (fibonacci 1)) (+ (fibonacci 1) (fibonacci 0))

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**FINDING THE INTEGERS FROM 1 DOWN TO ZERO**

(defun tozero (i)
  (if (zerop i) '0)
  (cons i (tozero (- i))))

(tozero 3)
(cons 3 (tozero 2))
(cons 2 (tozero 1))
(cons 1 (tozero 0))

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>2 1 0</td>
</tr>
<tr>
<td>3 2 1 0</td>
</tr>
</tbody>
</table>
1.40

SEARCHING FOR A VALUE
IN A LIST

(defun isin (val L)
  (cond
    ((null L) NIL)
    ((eql val (car L)) T)
    (T (isin val (cdr L))))
)

(isin 8 '(6 2 3 4 1 31))
(isin 8 '(2 3 4 8 1 31))
(isin 8 '(3 4 8 1 31))
(isin 8 '(4 8 1 31))
(isin 8 '(8 1 31))
T

1.41

Two-Sided Recursion

Searching for a Value in a Tree. Assume this kind of structure

(defun search (val TR)
  (cond
    ((null TR) NIL)
    ((atom TR) (eql TR val))
    (T
      (or
        (search val (car TR))
        (search val (cdr TR))))))
Another One with Similar Form

Determine if there are any odd numbers in an arbitrary list structure.

(defun any-odd (x)
  (cond
    ((null x) nil)
    (numberp x) (oddp x))
  (T (or
      (any-odd (first x))
      (any-odd (rest x))))
))

(any-odd S)

Is this efficient?

ONE MORE

(defun flatten (x)
  (cond
    ((null x) NIL)
    ((symbolp x) (list x))
    (T (append
        (flatten (first x))
        (flatten (rest x))))
  ))

(flatten '((1 (2 3)) (4 5 6)((7))))

(1 2 3 4 5 6 7)
**LOOP**

( loop < sequence of forms > )
where one form is ( return < val > )

( loop
  ( when ( null L ) ( return NIL ) )
  ( print ( car L ) )
  ( setf L ( cdr L ) )
)

--- loops can be more efficient
-- people like loops
-- but they involve assignment and thus side effects

--- loops can be more efficient

--- loops can be more efficient
FILE I/O WITH ITERATION

(with-open-file
  ( < stream name >
<br file specs >
  :direction < :input or :output > )
  < sequence of forms > )

(setf L NIL)
(with-open-file ( fi "myinput.lsp"
  :direction :input)
  ( dotimes ( i 10 )
    ( setf val ( read fi ) )
    ( setf L ( cons val L )))))
(with-open-file ( fo "myoutput.lsp"
  :direction :output)
  ( dolist ( q L ) ( print q fo )))