LECCION 1. ATOMOS Y LISTAS.

En la lección 1 se introducen las construcciones básicas del lenguaje LISP. Al finalizar esta lección, el alumno debe ser capaz de

-escribir cualquier expresión numérica en notación LISP (ejercicios 1 y 2).

-escudriñar listas y construirlas a partir de elementos cualesquiera de otras listas (ejercicios 4, 5 y 6).

-evaluar expresiones (ejercicios 1, 8 y 9) teniendo en cuenta las posibles apariciones de `QUOTE` *(ejercicio 3) y* `EVAL` *(ejercicio 7).*

-dar significado funcional a un símbolo (ejercicios 8, 9 y 10).

"Brindemos por el programador LISP, que encierra sus pensamientos en nidos de paréntesis"
(Alan Perlis).

R.1.1. *Concepto de expresión. Evaluación de números y listas*.
Evaluar las siguientes expresiones:
a) `(+ (* 3 (- 5 3)) (/ 8 4))`
b) `(* 3 2 1 (- 5 3))`
c) `(/ (+ 3 2 1) (- 5 3))`
d) `((+) (* 2 (- 5 3)) (/ 8 4))`
e) `(2 + 2)`
****

SOLUCION:
Una **expresión** es una lista o un átomo.

Un **átomo** es un símbolo o un número. Un **número** se puede escribir en cualquiera de las notaciones habituales (punto decimal, exponencial, ...).

Un **símbolo** se representa por un nombre, que está formado por caracteres alfanuméricos que no puedan interpretarse como un número. Algunos caracteres, como ", #, *espacio* no deben emplearse en el nombre de un símbolo. Sin embargo, otros como -, +, *, / pueden emplearse libremente. Por ejemplo, son símbolos `PEPE, X12, *PEPE*, PE//P, JUAN, PEPE-JUAN, PE-PE_JUAN1.`

Una **lista** es una sucesión de cero o más expresiones entre paréntesis. El blanco -no la coma- es el separador de expresiones dentro de una lista. Cada una de estas expresiones se denomina **elemento** de la lista. Por ejemplo, son listas

```
()
(A B C)
(A (B C))
```

cuyos elementos son respectivamente

-ninguno
-los tres símbolos `A B C`.
-el símbolo `A` y la lista `(B C)`.

Una expresión proporcionada a LISP se puede **evaluar**, es decir, LISP devuelve un valor -que será otra expresión- o bien señala un "error". En estas lecciones, si *E* se evalúa a *V* escribiremos E=>V. Las expresiones que se evalúan se llaman **formas**.

Un número se evalúa a sí mismo.

Una lista se evalúa del siguiente modo:

1. El primer elemento de la lista debe ser un símbolo *s* cuyo **significado funcional** esté definido. Algunos símbolos tienen significados funcionales predefinidos; en cualquier caso, el programador puede definirlos en LISP (vd. R.1.8). El significado funcional es un **objeto-procedimiento** o función propiamente dicha, *F*. *F* es la función en sí, y no debe confundirse con el símbolo *s*.

2. Se evalúan los restantes elementos de la lista, obteniendo los valores *V1, ..., Vn*. Si el número o tipo de los valores obtenidos no es coherente con los argumentos requeridos por *F*, se produce error.

3. La lista se evalúa a *F(V1, ..., Vn*).

Los signos aritméticos tienen asociados sus significados funcionales habituales. Exigen uno o más argumentos numéricos.

Por tanto

a)

```
(+ (* 3 (- 5 3)) (/ 8 4))
 ⇓  ⇓    ⇓       ⇓
 ⇓  ⇓    2       ⇓
 ⇓  6            2
 8
```

b)

```
(* 3 2 1 (- 5 3))
 ⇓       ⇓
 ⇓       2
 12
```

c)

```
(/ (+ 3 2 1) (- 5 3))
 ⇓  ⇓          ⇓
 ⇓  6          2
 3
```

d) Error: `(+)` es una lista.

e) Error: `2` es un número.

NOTA. Los tipos de expresiones presentados en este ejercicio constituyen lo que podemos denominar "LISP puro". Pueden resumirse en el siguiente cuadro:

expresión | *lista*
expresión | *átomo* | *símbolo*
expresión | *átomo* | *número*

En Common LISP existen además muchos otros tipos.

NOTA. Common Lisp permite emplear las notaciones decimal, exponencial, binaria, hexadecimal, etc. Tanto `1.` como `.1` son notaciones válidas . Vd. (CLtL2, pg. 15-23, ANSI 2.3.2).

NOTA. El nombre de un símbolo puede contener casi cualquier carácter. Para ello se emplean los caracteres de escape \ y |. Vd. (CLtL2, pg. 27-29, ANSI 2.2.4).

NOTA. El primer elemento de una lista que se ha de evaluar puede ser también una expresión lambda. Vd. lección 3.

NOTA. El significado funcional de un símbolo *s* puede ser, además de un objeto-procedimiento, una macro o una forma especial. En este caso, la evaluación de la lista (*s* ...) sigue reglas diferentes. Vd. R.1.3., etc.

R.1.2. *Funciones numéricas predefinidas.*
Escribir una expresión LISP que devuelva el mismo valor que las siguientes expresiones aritméticas escritas en notación habitual:
a) (531 *mod* 3) + (1 + 4×3 + 6/(6 - 3))$^{\sqrt{4}}$
b) *ln* |42×21 - 5670|
c) *max*(0, *min*(2,1), *max*(-1, 4´5))
d) 1 + 1 +1 + (7 -1)
****

SOLUCION:

`SQRT` tiene como significado funcional predefinido la función raíz cuadrada.

`EXP` tiene como significado funcional predefinido la función exponencial de base e.

`EXPT` tiene como significado funcional predefinido la función exponencial de base cualquiera, y por ello necesita dos argumentos, el primero la base y el segundo el exponente.

`LOG` tiene como significado funcional predefinido la función logaritmo. Si se da un solo argumento, se supone logaritmo natural. Si se da un segundo argumento, este es la base.

`ABS` tiene como significado funcional predefinido la función valor absoluto.

`MOD` tiene como significado funcional predefinido la función módulo.

`MAX` tiene como significado funcional predefinido la función máximo de cualquier número de argumentos.

`MIN` tiene como significado funcional predefinido la función mínimo de cualquier número de argumentos.

`1+` tiene como significado funcional predefinido la función sucesor.

`1-` tiene como significado funcional predefinido la función predecesor.

Para resolver el problema, hay que escribir las expresiones en notación prefija, empleando estos símbolos para representar las funciones. Por tanto las soluciones son

a) `(+ (MOD 531 3) (EXPT (+ 1 (* 4 3) (/ 6 (- 6 3)) (SQRT 4))))`

b) `(LOG (ABS (- (* 42 21) 5670)))`

c) `(MAX 0 (MIN 2 1) (MAX -1 4.5))`

d) `(1+ (1+ (1+ (1- 7))))`

NOTA:
Common Lisp tiene definidos los tipos numéricos entero, racional, real, doble precisión y complejo, entre otros. También un juego completo de funciones trigonométricas, hiperbólicas, trigonométricas inversas e hiperbólicas inversas. Vd. (CLtL2, cap. 12, ANSI 12).

NOTA:
En los símbolos, LISP no distingue mayúsculas de minúsculas (salvo que se indique explícitamente mediante el uso de caracteres de escape). De esta forma, las siguientes expresiones son idénticas a las anteriores:
a) `(+ (Mod 531 3) (expt (+ 1 (* 4 3) (/ 6 (- 6 3)) (sqrt 4)))`
b) `(log (ABS (- (* 42 21) 5670)))`
c) `(mAX 0 (Min 2 1) (MAX -1 4.5))`
Escribiremos siempre las expresiones LISP en mayúsculas, tipo de letra `COURIER`.

R.1.3. *Evaluación de símbolos. Forma especial* `QUOTE`.
Evaluar las siguientes expresiones:
a) `T`
b) `NIL`
c) `()`
d) `PEPE`
e) `(QUOTE PE-PE)`
f) `(QUOTE (PEPE1 PEPA1 MARI2))`
g) `'PEPE`
h) `'(PEPE '(PEPA MARI))`
i) `'T`
j) `(QUOTE ())`
k) `'7`
l) `'(+ 3 4)`
m) `('+ 3 4)`
****

SOLUCION:
`NIL` es un símbolo que representa la lista vacía. Esto implica que la lista vacía es tanto una lista como un símbolo.

Los símbolos `T` y `NIL` se evalúan a sí mismos.

Un símbolo *s* puede estar ligado, es decir, contener una referencia o indicación a otra expresión *V*. Llamamos **ligadura**. al par (*s*, *V*). Un símbolo *s* (salvo `T` o `NIL`) se evalúa al valor *V* al que está ligado. Posteriormente estudiaremos las formas en las que un símbolo se puede ligar; por ahora supondremos que todos los símbolos están sin ligar. Cuando queremos hacer énfasis en el uso de un símbolo en una ligadura, decimos que es una **variable**.
Según esto, las respuestas son
a) `T => T`
b) `NIL` **=>** `NIL`
c) `() => NIL`
d) `PEPE => ... Error: PEPE sin ligar.`

El significado funcional de un símbolo puede ser un objeto-procedimiento o función en sentido estricto. También puede ser una **macro** o una **forma especial**. Cuando el significado funcional del primer elemento de una lista es una macro o una forma especial, la evaluación de la lista no sigue las reglas dadas en R.1.1., sino reglas propias.

`QUOTE` es una forma especial que tiene un sólo argumento:

(`QUOTE` *expresión*)

El valor de (`QUOTE` *expresión*) es precisamente *expresión*, sin evaluar.

Cuando dos expresiones *expr1* y *expr2* se evalúan siempre al mismo valor, emplearemos la notación

expr1 == *expr2*

Ya que `QUOTE` se emplea muy frecuentemente, existe una abreviatura:

(`QUOTE` *expresión*) == '*expresión*

El intérprete LISP deshace las abreviaturas antes de la evaluación.
Según esto, las respuestas a los restantes apartados son
e) `(QUOTE PE-PE) => PE-PE`
f) `(QUOTE (PEPE1 PEPA1 MARI2))` **=>** `(PEPE1 PEPA1 MARI2)`
g) `'PEPE => PEPE`
h) `'(PEPE '(PEPA MARI)) => (PEPE (QUOTE (PEPA MARI)))`
i) `'T` **=>** `T`
j) `(QUOTE ())` **=>** `NIL`
k) `'7 => 7`
l) `'(+ 3 4)` **=>** `(+ 3 4)`
m) `('+ 3 4) => ...Error: (QUOTE +) no tiene significado funcional.`

NOTA. Las formas especiales forman parte de la definición del lenguaje (CLtL2, pg. 72, ANSI 3.1.2.1.2). El usuario no puede definir nuevas formas especiales. Por el contrario, sí es posible definir nuevas macros. En cualquier caso, el lector puede olvidarse de la diferencia entre macros y formas especiales.

R.1.4. *Manejo elemental de listas.*
Evaluar las siguientes expresiones:
a) `(CONS (CAR '(CUQUI LOLI PEPI)) (CDR '(FALI LELI RODRI)))`
b) `(CDR (CDR (CONS T (CDR '(PEPI)))))`
c) `(CAR (CAR (CONS T (CDR '(PEPI)))))`

****
SOLUCION:

`CAR` tiene como significado funcional predefinido la siguiente función:

```
(CAR lista)
```

-un único argumento que debe ser una lista.
-si el argumento no es `NIL`, el valor devuelto es el primer elemento de la lista argumento.
-si el argumento es `NIL`, el valor devuelto por `CAR` es `NIL`.

`CDR` tiene como significado funcional predefinido la siguiente función:

```
(CDR lista)
```

-un único argumento que debe ser una lista.
-si el argumento no es `NIL`, el valor devuelto es la lista obtenida quitando el primer elemento de la lista argumento.
-si el argumento es `NIL`, el valor devuelto por `CDR` es `NIL`.

`CONS` tiene como significado funcional predefinido la siguiente función:

```
(CONS expresión lista)
```

-exactamente dos argumentos. El segundo debe ser una lista.
-el valor devuelto es una lista cuyo primer elemento es el primer argumento, y cuyos restantes elementos son los del segundo argumento.

Según esto,
a)

```
(CONS (CAR '(CUQUI LOLI PEPI)) (CDR '(FALI LELI RODRI)))
 ⇓       ⇓ ⇓                      ⇓   ⇓
 ⇓       ⇓ (CUQUI LOLI PEPI)      ⇓  (FALI LELI RODRI)
 ⇓      CUQUI                 (LELI RODRI)
(CUQUI LELI RODRI)
```

b)

```
(CDR (CDR (CONS T (CDR '(PEPI)))))
  ⇓    ⇓     ⇓      ⇓   ⇓
  ⇓    ⇓     ⇓      ⇓  (PEPI)
  ⇓    ⇓     ⇓     NIL
  ⇓    ⇓    (T)
  ⇓   NIL
 NIL
```

c)

```
(CAR (CAR (CONS T (CDR '(PEPI)))))
  ⇓    ⇓     ⇓      ⇓   ⇓
  ⇓    ⇓     ⇓      ⇓  (PEPI)
  ⇓    ⇓     ⇓     NIL
  ⇓    ⇓    (T)
  ⇓    T
Error: T no es una lista.
```

NOTA:
Existen funciones equivalentes a CAR y CDR:

```
(CAR lista) == (FIRST lista)
(CDR lista) == (REST lista)
```

NOTA:
En realidad, el segundo argumento de `CONS` no tiene que ser una lista. No obstante, si no lo es, el valor devuelto tampoco es una lista, sino una **lista punteada**; vd. (CLtL, pg. 30-31, ANSI 14) y la lección 8.

R.1.5. *Más funciones para manejo de listas.*
Evaluar las siguientes expresiones:
a)
```
(CONS (CDAR '((CONS BUENOS DIAS)))
            (CADDR '(BUENAS B (TARDES NOCHES))))
```
b)
```
(LIST (CONS '(CUQUI) '(LOLI))
      (APPEND '(CUQUI) '(LOLI))
      (LIST '(CUQUI) '(LOLI)))
```
c)
```
(LENGTH (APPEND '(CUQUI LOLI)
                '(LOLI CUQUI)
                '((CUQUI LOLI))))
```
****
SOLUCION:
a)
Si $a_1, ...a_n$ son caracteres A ó D y n<5, entonces está predefinido que

(Ca$_1$a$_2$...a$_n$R lista) == (Ca$_1$R (Ca$_2$R ... (Ca$_n$R lista) ...))

Por tanto

```
(CONS (CDAR '((CONS BUENOS DIAS))) (CADDR '(BUENAS B (TARDES NOCHES))))
  ⇓      ⇓   ⇓                          ⇓   ⇓
  ⇓      ⇓   ((CONS BUENOS DIAS))       ⇓   (BUENAS B (TARDES NOCHES))
  ⇓   (CONS BUENOS DIAS)              (B (TARDES NOCHES))
  ⇓   (BUENOS DIAS)                   ((TARDES NOCHES))
  ⇓                                   (TARDES NOCHES)
  ⇓
((BUENOS DIAS) TARDES NOCHES)
```

b)

`LIST` tiene como significado funcional predefinido la siguiente función:

`(LIST [expresión]*)`

-un número indeterminado de argumentos.
-si no hay ningún argumento, el valor devuelto es `NIL`.
-si hay algún argumento, el valor devuelto es una lista cuyos elementos son los argumentos.

`APPEND` tiene como significado funcional predefinido la siguiente función:

`(APPEND [lista]*)`

-un número indeterminado de argumentos, que deben ser listas.
-si no hay ningún argumento, el valor devuelto es `NIL`.
-si hay algún argumento, el valor devuelto es una lista cuyos elementos son los elementos de los argumentos. Por tanto

```
LIST
    arg1: CONS
                arg1: (CUQUI)
                arg2: (LOLI)
    ((CUQUI) LOLI)
    arg2: APPEND
                arg1: (CUQUI)
                arg2: (LOLI)
    (CUQUI LOLI)
    arg3: LIST
                arg1: (CUQUI)
                arg2: (LOLI)
   ((CUQUI) (LOLI))
=> (((CUQUI) LOLI) (CUQUI LOLI) ((CUQUI) (LOLI)))
```

c)

`LENGTH` tiene como significado funcional predefinido la siguiente función:

-un único argumento, que debe ser una lista.

-el valor devuelto es la longitud del argumento, es decir, el número de elementos de la lista.

Por tanto

```
LENGTH
   arg: APPEND
           arg1: (CUQUI LOLI)
           arg2: (LOLI CUQUI)
           arg3: ((CUQUI LOLI))
         (CUQUI LOLI LOLI CUQUI (CUQUI LOLI))
=> 5
```

NOTA:

Existen funciones equivalentes a algunas de las anteriores:

```
(CADR lista) == (SECOND lista)
(CADDR lista) == (THIRD lista)
```

NOTA:

En realidad, el último argumento de `APPEND` no tiene que ser una lista. No obstante, si no lo es, el valor devuelto tampoco es una lista, sino una **lista punteada**; vd. (CLtL2 90, pg. 30-31, ANSI 14) y la lección 8.

R.1.6. *Manejo de listas. Relaciones notables.*
Discutir en qué condiciones son válidas las siguientes afirmaciones:
a) `(CONS (CAR 'lista) (CDR 'lista)) => lista`
b) `(CAR (CONS 'expresión 'lista)) => expresión`
c) `(CDR (CONS 'expresión 'lista)) => lista`
d) `(CONS 'expresión1 (LIST 'expresión2)) == (LIST 'expresión1 'expresión2)`
e) `(CONS 'expresión1 (LIST NIL)) == (APPEND (LIST 'expresión1) NIL)`
****

SOLUCION:
Supongamos que *lista* es realmente una lista.
a) Si *lista* no es `NIL`, `CAR` devolverá su primer elemento, `CDR` su resto, y `CONS` aplicado a estos argumentos devolverá la lista original.
Sin embargo, si la lista es `NIL`, `CAR` devolverá `NIL`, `CDR` devolverá `NIL`, y `CONS` aplicado a `NIL NIL` devolverá (`NIL`).
Por tanto, la afirmación es válida únicamente si *lista* es distinta de `NIL`.

b y c) Si `CONS` se ha evaluado, el valor devuelto es una lista, cuyo primer elemento es *expresión* y cuyo resto es *lista*. Por tanto, `CAR` devolverá *expresión* y `CDR` *lista*. Las afirmaciones son válidas.

d) Ambas expresiones son siempre equivalentes, ya que ambas se evalúan a listas de dos elementos cuyo primer elemento es `expresión1` y cuyo segundo elemento es `expresión2`.

e) Ambas expresiones no son nunca equivalentes, ya que la primera se evalúa siempre a una lista de dos elementos y la segunda a una lista de sólo un elemento. Por ejemplo,

```
(CONS 'A (LIST NIL)) => (A NIL)
(APPEND (LIST 'A) NIL) => (A)
```

R.1.7. *La función* EVAL.
Evaluar las siguientes expresiones:
a)
```
(EVAL (APPEND (CDDDR '(- + * /))
              (LIST (EVAL '(+ 3 5)))
              (CONS 4 NIL)))
```
b)
```
(EVAL (QUOTE (QUOTE (CAR (A B)))))
```
c)
```
(EVAL (QUOTE (CAR (QUOTE (A B)))))
```
d)
```
(EVAL (CAR (QUOTE (QUOTE (A B)))))
```
e)
```
(CAR (EVAL (QUOTE (QUOTE (A B)))))
```

****
SOLUCION:
EVAL tiene como significado funcional predefinido la siguiente función:
(EVAL expresión)
-un único argumento.
-el valor devuelto es el valor resultante de evaluar el argumento.
De esta forma, (EVAL *expresión*) realiza dos evaluaciones de *expresión*: la primera, debida al ciclo normal de evaluación; la segunda, debida al uso explícito de EVAL.
Por tanto
a)
```
EVAL
    arg: APPEND
              arg1: CDDDR   arg: (- + * /)
                (/)
              arg2: LIST    arg: EVAL  arg: (+ 3 5)
                                 8
               (8)
              arg3: CONS    arg1: 4
                            arg2: NIL
                (4)
    (/ 8 4)
=> 2
```
b)
```
(EVAL (QUOTE (QUOTE (CAR (A B)))))
=> (CAR (A B))
```
c)
```
(EVAL (QUOTE (CAR (QUOTE (A B)))))
=> A
```
d)
```
(EVAL (CAR (QUOTE (QUOTE (A B)))))
=> error: QUOTE sin ligar
```
e)
```
(CAR (EVAL (QUOTE (QUOTE (A B)))))
=> A
```

NOTA:
No es buen estilo de programación emplear la forma EVAL. Además, EVAL no recupera los valores de las variables ligadas léxicamente. Por todo ello, "*en la práctica, la principal utilidad de* EVAL *es explicar la semántica del lenguaje.* EVAL *y* SET *no se suelen usar realmente en los programas, así que si descubres que los estás empleando -y no estás implementando un intérprete LISP- es probable que estés haciendo algo mal*" (Charniak 87, pg. 110).

R.1.8. *Definición de significados funcionales de símbolos. Concepto de entorno.*
Evaluar las siguientes expresiones en el orden dado, indicando los **efectos laterales**:

a)
```
(DEFUN CUADRADO (N) (* N N))
(CUADRADO 7)
(CUADRADO (CUADRADO 7))
```

b)
```
(DEFUN RAICES (A B C)
      (LIST (/ (+ (- B) (SQRT (- (* B B) (* 4 A C)))) (* 2 A))
            (/ (- (- B) (SQRT (- (* B B) (* 4 A C)))) (* 2 A))))
(RAICES 1 0 -1)
RAICES
```

****

SOLUCION:

DEFUN tiene el siguiente significado funcional predefinido:

```
(DEFUN símbolo lista-lambda cuerpo)
```

-DEFUN es una macro. Ello quiere decir que no evalúa sus argumentos en la forma explicada en R.1.1. DEFUN toma sus argumentos literalmente.
-el primer argumento es un símbolo.
-el segundo argumento es una **lista-lambda**. Por ahora, consideraremos que una lista-lambda es una lista de cero o más símbolos que se denominan **parámetros**.
-*cuerpo* está formado por un número cualquiera de expresiones.
-el valor devuelto es símbolo.
-Se entiende por efecto lateral cualquier consecuencia de la evaluación de una expresión que no sea la simple determinación del valor resultante. DEFUN tiene el siguiente efecto lateral:
-se crea un objeto-procedimiento *F* a partir del *cuerpo*
-se establece *F* como el significado funcional de símbolo.

Concretamente, una vez ejecutado (DEFUN símbolo lista-lambda cuerpo), cuando se evalúa

```
(símbolo [argumento]*)
```

ocurre lo siguiente:

-Se crea un **entorno**. Un entorno es un conjunto de ligaduras. Los símbolos ligados son precisamente los parámetros de la función, y sus valores los argumentos que se pasan. Si el número de argumentos no coincide con el de parámetros, se produce un error.
-En este entorno se evalúan las expresiones del cuerpo.
-El valor devuelto por la función es el de la última expresión evaluada.

Más adelante se explicarán aspectos adicionales de los entornos.
Por tanto

a)

```
(DEFUN CUADRADO (N) (* N N)) => CUADRADO
```

y como efecto lateral, CUADRADO tiene como significado funcional "la función de *n* que devuelve el cuadrado de *n*". Representemos los entornos por rectángulos en cuyo interior figuran las ligaduras, y adosemos las expresiones a los entornos donde se evalúan. Entonces

```
(CUADRADO 7) ==
      N<-7
      (* N N)
                                    => 49
```

```
(CUADRADO (CUADRADO 7)) ==
      N<-?
      (* N N)
```

Para determinar el valor al que se liga N se realiza otra llamada a CUADRADO

```
      N<-7
      (* N N)
                                 => 49
```

Por tanto

```
      N<-49
      (* N N)
                                        => 2401
```

b)

```
(DEFUN RAICES (A B C)
      (LIST (/ (+ (- B) (SQRT (- (* B B) (* 4 A C)))) (* 2 A))
            (/ (- (- B) (SQRT (- (* B B) (* 4 A C)))) (* 2 A))))
=> RAICES
```

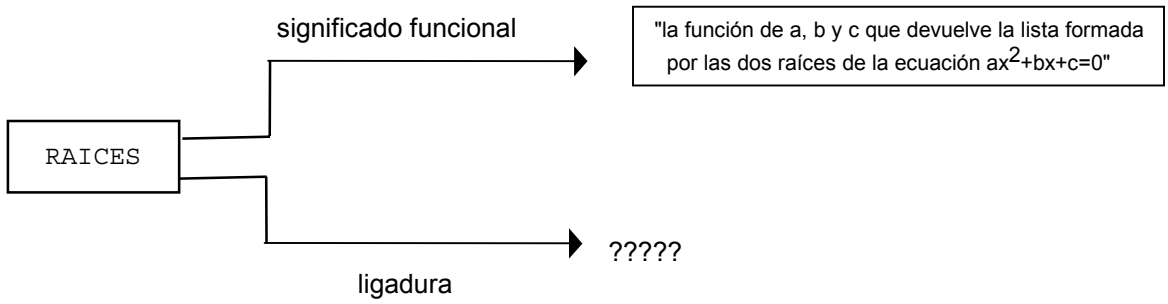
y como efecto lateral, `RAICES` tiene el significado funcional "la función de *a*, *b* y *c* que devuelve la lista formada por las dos raíces de la ecuación $ax^2+bx+c=0$". Entonces

```
(RAICES 1 0 -1) ==
      A<- 1
      B<- 0
      C<- -1
(LIST (/ (+ (- B) (SQRT (- (* B B) (* 4 A C))))(* 2 A))
      (/ (- (- B) (SQRT (- (* B B) (* 4 A C))))(* 2 A)) )
                                                    => (1 -1)
```

Por otra parte

```
RAICES
=> error: RAICES sin ligar
```

En CommonLisp son independientes el significado funcional y la ligadura del símbolo: los valores establecidos para la una no afectan a la otra. Por tanto, el significado funcional establecido para `RAICES` no es su ligadura y al intentar evaluar el símbolo `RAICES` se produce un error. Podemos visualizar la situación según el esquema siguiente:



R.1.9. *Evaluación de funciones definidas por el usuario.*
Supuestas las definiciones de R.1.8, evaluar las siguientes expresiones en el orden dado, indicando los efectos laterales que se producen:

a)
```
(DEFUN CUADRADO (N) (+ N N))
(CUADRADO 7)
```
b)
```
(DEFUN LIO2 (L) (LIST (LIO1 L L) (CAR L)))
(DEFUN LIO1 (L1 L2) (LIST (CAR L1) (CAR L2)))
(LIO2 '(DICEN DE UN SABIO))
```
c)
```
(DEFUN LIO3 (L) (LIO4 (CONS 'A L)))
(DEFUN LIO4 (M) (APPEND L M))
(LIO3 '(VECES))
```
d)
```
(DEFUN KK1 (N) 27)
(KK1 PEPE)
```
****

SOLUCION:

a)
```
(DEFUN CUADRADO (N) (+ N N))
=> CUADRADO
```
y como efecto lateral, `CUADRADO` tiene el significado funcional "la función de *n* que devuelve el duplo de *n*". Nótese que desaparece el significado funcional anteriormente asignado a `CUADRADO`.
Entonces
```
(CUADRADO 7)
=> 14
```
b)
```
(DEFUN LIO2 (L) (LIST (LIO1 L L) (CAR L)))
=> LIO2
```
y como efecto lateral, `LIO2` tiene el significado funcional "la función de *L* que devuelve una lista formada por *lio1(L, L)* y *car(L)*".
```
(DEFUN LIO1 (L1 L2) (LIST (CAR L1) (CAR L2)))
=> LIO1
```
y como efecto lateral, `LIO1` tiene el significado funcional "la función de *L1* y *L2* que devuelve una lista con los primeros elementos de *L1* y *L2*".
Entonces
```
(LIO2 '(DICEN DE UN SABIO))   ==
      L<-(DICEN DE UN SABIO)
      (LIST (LIO1 L L) (CAR L))
```
La evaluación del primer argumento de `LIST` produce una llamada a `LIO1`:
```
      L1<-(DICEN DE UN SABIO)
      L2<-(DICEN DE UN SABIO)
      (LIST (CAR L1) (CAR L2))
                                    => (DICEN DICEN)
```
y por tanto
```
      L<-(DICEN DE UN SABIO)
      (LIST (LIO1 L L) (CAR L))
                                     => ((DICEN DICEN) DICEN)
```
c)
```
(DEFUN LIO3 (L) (LIO4 (CONS 'A L)))
=> LIO3
```
y como efecto lateral, `LIO3` pasa ahora a tener el significado funcional "la función de *L* que CONSa el símbolo *A* y *L*".
```
(DEFUN LIO4 (M) (APPEND L M))
=> LIO4
```
y como efecto lateral, `LIO4` pasa ahora a tener el significado funcional "la función de *M* que APPENDa *L* y *M*". Pero, ¿quién es *L* en esta definición? Veámoslo:
```
(LIO3 '(VECES))
=> Error: L sin ligar.
```
Sigamos paso a paso esta última evaluación:

```
(LIO3 '(VECES))
==
  Entorno 1
      L<-(VECES)
      (LIO4 (CONS 'A L))
==
  Entorno 2
      M<-(A VECES)
      (APPEND L M)
=> Error: L sin ligar.
```

Nótese que el entorno 2 se ha creado independientemente del entorno 1: las ligaduras de 1 no tienen efecto en 2 y, por tanto, *L* no está ligado a (VECES). En general, la creación de entornos se rige por la regla siguiente: si al evaluar la expresión *E1* resulta necesario evaluar una subexpresión *E2*, el entorno de evaluación de *E2* es independiente del entorno de evaluación de E1. Esta regla se denomina *regla de ámbito léxico*.

d)

```
(DEFUN KK1 (N) 27)
=> KK1
(KK1 PEPE)
==
=> Error: PEPE sin ligar.
```

Siguiendo la regla de evaluación dada en R.1.1, se intenta evaluar el segundo elemento de la lista, que es un símbolo sin ligar; por tanto, se produce y señala este error. Sin embargo, si LISP fuera suficientemente astuto, habría razonado así:

```
(KK1 PEPE) == la función que siempre devuelve 27, aplicada a la ligadura
de PEPE => 27
```

pues es claro que no resulta necesario calcular el valor del argumento de KK1, ya que es irrelevante para calcular el valor que se debe devolver.

El orden de evaluación enunciado en R.1.1. es el que sigue LISP y se llama *orden aplicativo*, de *llamada por valores* o *call-by-value*. El orden de evaluación sugerido ahora -que LISP no sigue- se llama *orden normal*, de *llamada por necesidad* o *call-by-need*.

R.1.10. *Primeros ejercicios de diseño de funciones.*
Escribir en LISP las definiciones de las siguientes funciones:
a)la función de *L* que devuelve la lista formada a partir de *L* poniendo su tercer elemento en primer lugar.
b)la función de *L* (lista) y *a* (átomo) que devuelve la lista formada sustituyendo el tercer elemento de *L* por *a*.
c)la función que da el determinante de una matriz de 2x2. Supóngase que la matriz está representada como una lista de listas *((a11 a12) (a21 a22))*.
****

SOLUCION:
a)
El tercer elemento de *L* es `(CADDR L)`
Sus dos primeros elementos son `(CAR L)` y `(CADR L)`
La lista de los restantes elementos es `(CDDDR L)`
Llamemos `SUBIR-TERCERO` a la función. La definición pedida será

```
(DEFUN SUBIR-TERCERO (L)
   (CONS (CADDR L)
      (CONS (CAR L)
         (CONS (CADR L)
            (CDDDR L)))))
```

b)
Llamemos `SUST-TERCERO` a la función.
Análogamente

```
(DEFUN SUST-TERCERO (A L)
   (CONS (CAR L)
      (CONS (CADR L)
         (CONS A
            (CDDDR L)))))
```

c)
Llamemos `DET` a la función.

```
(DEFUN DET (M)
   (- (* (CAAR M) (CADADR M))
      (* (CADAR M) (CAADR M))))
```

EJERCICIOS PROPUESTOS.

P.1.1. Evaluar las siguientes expresiones:
a) `(* (/ (+ 3 3) (- 4 4)) (* 0 9))`
b) `(SQRT (EXPT (+ 1 2) 6))`
c) `(SQRT (- 0 (EXPT (+ 1 2) 6)))`
d) `(EXPT 2 (EXPT 2 (EXPT 2 (EXPT 2 2))))`
e) `(EXPT (MOD 5 3) (ABS (- 5 7)))`
f) `(LOG (EXP (+ 3 2 1)))`

P.1.2. Evaluar las siguientes expresiones:
a) `(QUOTE (NIL 'NIL T 'T))`
b) `(QUOTE (QUOTE (BUENOS DIAS)))`
c) `('+ '1 '(* 3 2))`
d) `'('+ '1 '(* 3 2))`
e) `'(+ 1 (* 3 2))`
f) `(+ '1 '(* 3 2))`
g) `(+ '1 (* 3 '2))`

P.1.3. Estudiar cuáles de las siguientes expresiones son equivalentes:
a) `(CONS 5 (T NIL))`
b) `(CONS 5 '(T NIL))`
c) `(CDDR (CONS T (CONS NIL (CONS 5 (CONS T (CONS NIL NIL))))))`
d) `(CONS 5 (LIST T NIL))`
e) `(CONS 5 (APPEND (LIST T) (LIST NIL)))`
f) `(APPEND (CONS 5 NIL) (LIST 'T) '(()))`
g) `(LIST (CADR '(3 5 7)) (CDAR '((NIL T))) (CDDDR '(NIL T)))`

P.1.4. Estudiar cuáles de las siguientes expresiones son equivalentes:
a) `(EVAL ''(BUENOS DIAS))`
b) `(EVAL (QUOTE (BUENOS DIAS)))`
c) `(QUOTE (EVAL (BUENOS DIAS)))`
d) `(EVAL (QUOTE ('BUENOS 'DIAS)))`
e) `(EVAL (QUOTE (EVAL (QUOTE '(BUENOS DIAS)))))`

P.1.5. Evaluar las siguientes expresiones:
a) `(CONS (LENGTH '(+ 3 4 5)) (LIST (+ 3 4 5) '(+ 3 4 5)))`
b) `(LENGTH (CONS (+ 3 4 5) (APPEND '(+ 3 4 5) NIL)))`
c) `(LIST (LENGTH 'PEPE) '(JUAN MARI))`

P.1.6. Sean dos listas *L1*, *L2*. Empleando únicamente CONS y NIL, dar expresiones equivalentes a
a) `(LIST L1 NIL)`
b) `(LIST L1 L2)`
c) `(APPEND NIL L1 NIL)`

P.1.7. Sean las definiciones LISP siguientes:

```
(DEFUN F1 (N) (/ (+ N 1) N))
(DEFUN F2 (N) (/ N (+ N 2)))
(DEFUN F3 (N) (+ (F1 N) (F2 N)))
```

Evaluar

```
(F3 (F2 (F1 2)))
```

indicando claramente las ligaduras que se establecen para el símbolo `N`.

P.1.8. Definir en LISP las siguientes funciones:
a)empleando CAR, CDR y CONS la función de una lista de tres símbolos *L*, que devuelve la lista formada por listas formadas por cada elemento de *L*. Por ejemplo, aplicada a `(ARTIFICIAL INTELIGENCIA ARTIFICIAL)` debe dar `((ARTIFICIAL) (INTELIGENCIA) (ARTIFICIAL))`.

b)la función *Haprox(n)* que aproxima el valor de la suma armónica parcial

$$H(n) = \sum_{1 \le k \le n} \frac{1}{k}$$

mediante la fórmula

$$\ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^3}$$

donde $\gamma = 0{,}57721...$

P.1.9. Supongamos que los vectores tridimensionales se representan en LISP por listas de tres números. Definir mediante funciones LISP las siguientes operaciones:
a)el módulo de un vector.
b)el producto escalar de dos vectores.
c)el producto vectorial de dos vectores.

P.1.10. Sean las definiciones LISP siguientes:

```
(DEFUN MI-QUOTE (E) (QUOTE E))
(DEFUN MI-CAR (E) (CAR E))
(DEFUN MI-DEFUN (NOMBRE LL E) (DEFUN NOMBRE LL E))
```

Estudiar cuáles de las siguientes equivalencias son válidas:

a) `(MI-QUOTE exp) == (QUOTE exp)`
b) `(MI-CAR exp) == (CAR exp)`
c) `(MI-DEFUN símbolo lista exp) == (DEFUN símbolo lista exp)`

NOTAS ERUDITAS Y CURIOSAS.

El lenguaje LISP fue creado por John McCarthy a finales de los años 50 (McCarthy 60) cuando se encontraba en el M.I.T.; es, después de FORTRAN, el más antiguo lenguaje de programación aún en uso. Su primera versión difundida fue LISP 1.5 (McCarthy et al. 65). La etimología oficial de LISP es "LISt Processing"; otros dicen que en realidad es un acrónimo de "Lots of Insipid Stupid Parentheses".

La notación y ciertos rasgos de la semántica de LISP están inspirados en el lambda-cálculo de A. Church (Church 41); pero hay que señalar que las diferencias entre ambos formalismos son grandes.

Desde su nacimiento, LISP se dividió en muchos dialectos mutuamente incompatibles. Los más empleados fueron MacLisp y su familia, originados alrededor del Laboratorio de Inteligencia Artificial del M.I.T., e Interlisp y su familia, originados a partir del BBN-LISP y desarrollados posteriormente en Xerox PARC. La evolución de estos dialectos produjo lenguajes muy potentes, al precio de complicar enormemente la semántica.

El lenguaje SCHEME (Steele y Sussman, 1975), por el contrario, es un LISP de sintaxis simple y semántica elegantemente definida; en cierto modo, es una vuelta a las fuentes del lambda-cálculo.

Fig. I. 1. John McCarthy en 1956.

En los años 80 la comunidad de programadores LISP empieza a pensar que quizás valdría la pena sacrificar parte de las maravillosas prestaciones de sus dialectos en aras de la portabilidad de los programas. Guy Steele lidera el movimiento y rápidamente se define un estándar, el CommonLISP (CLtL, Steele 82) que conoce ya una segunda versión familiarmente denominada CLtL2 (Steele 90). Existe también un estándar ANSI (ANSI 95).

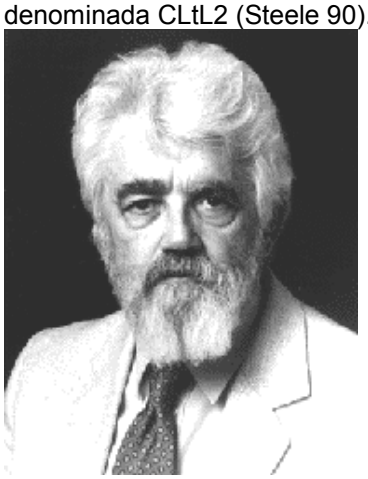


Fig. I.2. John McCarthy en la actualidad.

LISP nació para la IA y durante muchos años la identificación práctica de ambos conceptos fue completa. El heurístico "algo es IA si y sólo si está programado en LISP" servía para caracterizar este concepto, siempre escurridizo. Pero en los años 80 la situación cambió, debido a varias causas, entre ellas la aparición del Prolog como lenguaje de uso (relativamente) común. También influyó en este cambio el fracaso comercial de las máquinas LISP.

Por otra parte, las características de LISP no lo limitan a aplicaciones de IA. Su potencia expresiva y simplicidad sintáctica lo hacen apropiado para cualquier tipo de aplicación. Por ejemplo, la conocida herramienta Autocad fue originariamente desarrollada e implementada sobre LISP.

Las funciones introducidas en este capítulo provienen de LISP 1.5. En el dialecto original, así como en otros como SCHEME, se emplea `DEFINE` en lugar de `DEFUN`.

Ninguna historia de LISP puede dejar de explicar la etimología de `CAR` (*Content of the Address part of the Register*) y `CDR` (*Content of the Decrement part of the Register*). La primera máquina donde se implementó LISP fue una IBM704, cuyas palabras constaban de dos partes llamadas "dirección" y "decremento". Como puede observarse, los nombres CAR y CDR son acrónimos sus contenidos. Cuando se explique la representación intermedia de objetos LISP (células CONS) quedará justificada su elección.