INTRODUCCIÓN AL LENGUAJE COMMON LISP

Cuando se inicia la ejecución de LISP siempre aparece un prompt como > Después se puede teclear y obtener resultados

```
> (+ 2 2)
4
```

Las expresiones aritméticas siguen la notación prefija. Los resultados de los cálculos son impresos por el evaluador. La notación prefija es más compacta que la infija

```
> (+ 1 2 3 4 5 6 7 8 9 10)
55
> (- (+ 9000 900 90 9) (+ 5000 500 50 5))
4444
```

Las expresiones se pueden anidar. La regla de evaluación es mucho más simple que la usual en matemáticas u otros lenguajes de programación. Primero se evalúan los argumentos y posteriormente se aplica la función sobre los resultados de la evaluación. Una diferencia inicial para programadores de otros lenguajes de programación es que no hay diferencia entre expresión y sentencia. en C

`2 + 2` tiene un valor, en cambio `x = 2 + 2` no. En LISP toda expresión, tenga efecto o no, devuelve un valor.

Las reglas léxicas son mucho más simples en LISP. Solo hay paréntesis, comillas (sencilla, doble y backquote) espacios y comas. El punto y coma no separa expresiones (no es necesario debido a los paréntesis) sino que inicia comentarios. LISP permite la manipulación de símbolos, y la construcción de estructuras de datos complejas a partir de ellos.

```
> (append '(Carlos Antonio) '(Carmen Rosa))
(CARLOS ANTONIO CARMEN ROSA)
```

La parte extraña es el (’), que sirve para bloquear la evaluación de una expresión y devolverla literalmente.

```
> '(Carlos Antonio)
(CARLOS ANTONIO)
> 'A
A
> '2
2
> 2
2

> '(+ 2 2)
(+ 2 2)
> (+ 2 2)
4
> A
Error: A is not a bound variable
> (Carlos 'Antonio)
Error: Carlos is not a function
```

Los cálculos simbólicos y numéricos se pueden mezclar.

```
> (+ 2 (length '(a b c d)))
6
```

LISP no otorga ninguna semántica a los símbolos que manipula. CommonLISP contiene un conjunto de funciones predefinidas que es necesario conocer. Como +, append, length. Los símbolos en CommonLISP no son casesensitive. Muchos signos de puntuación son autorizados para formar parte de los símbolos: '?!$/<=>'

Variables

Los símbolos se utilizan para dar nombre a las variables. Una variable puede tomar como valor cualquier objeto LISP.

```
> (setf p '(esto es una asignacion))
> p
(esto es una asignacion)
> (setf x 10)
> (+ x x)
20
> (+ x (length p))
14
> (setf a 3 b 5)
```

Los símbolos se utilizan también para dar nombre a las funciones. Un mismo símbolo puede ser a la vez nombre de una variable y nombre de una función.

Formas especiales

En los ejemplos anteriores setf viola la regla de evaluación. No evalúa sus argumentos y luego aplica la función al resultado de esta evaluación. Setf no es una función. Se trata de uno de los componentes de la sintaxis básica de LISP. A las expresiones sintácticas de LISP se las conoce como Formas Especiales. Por ejemplo,

```
(setf x (+ 1 2) )
```

A veces se denomina Formas Especiales a los símbolos como setf y a veces a las expresiones encabezadas por estos símbolos, lo cual puede crear una cierta confusión.

Listas

Dado que las listas son esenciales en LISP, veremos unas cuantas funciones de procesado de listas.

```
> p
(esto es una asignacion)
> (first p)
esto ; equivalente a (car p)
> (second p)
es ; equivalente a (cadr p)
> (third p)
una ; equivalente a (caddr p), etc.
> (fifth p)
nil
> (length p)
4
```

Algunos detalles sobre listas. rest (o cdr) devuelve la lista resultante de eliminar el primer elemento de la lista argumento. NIL y ( ) son completamente equivalentes. NIL también se utiliza para representar el valor falso. Las listas pueden contener sublistas como elementos.

```
> (setf x '((una sublista) 2 (tercero) ((4)) 5))
((una sublista) 2 (tercero) ((4)) 5))
> (first x)
(una sublista)
>(second x)
2
> (third x)
(tercero)
> (fourth x)
((4))
> (second (first x))
sublista
```

Aparte de acceder a los elementos de las listas, también podemos construir listas.

```
> p
(esto es una asignación)
> (cons 'porque p)
(porque esto es una asignación)
> (cons (first p) (rest p))
(esto es una asignación)
> (setf ciudad (list 'Paris 'Roma))
(paris roma)
> (list p 'de (list ciudad) 'tachin)
((esto es una asignacion) de ((parisroma)) tachin)
> (cons 'Paris (cons 'Roma ()))
(paris roma)
> (append p ciudad)
(esto es una asignacion paris roma)
```

Cons viene de Construct. Estas funciones crean nuevas listas sin destruir las listas preexistentes.

```
> (last p)
(asignacion)
> (first (last p))
asignacion
```

Funciones

La forma especial defun viene de "define function". Se utiliza para definir funciones.

```
(defun last-name (name)
        "Selecciona el apellido de un nombre representado como una lista. (en formato sajón)"
        (first (last name)))
```

La función recibe un nombre last-name, tiene una lista de parámetros que contiene un único parámetro (name). También tiene una cadena de documentación que dice que es lo que la función hace. El cuerpo de la función es (first (last name)). En general

```
(defun function-name (parameter ...)
        "Documentation string"
        function-body ...)
```

Una vez definida, la función se puede utilizar como cualquier otra función LISP.

```
> (last-name '(John Q Smith))
SMITH
> (last-name '(Antonio De las Cruces Revueltas))
REVUELTAS
> (last-name '(Aristoteles))
Aristoteles
```

Podemos definir la función first-name

```
(defun first-name (name)
"Selecciona el nombre de pila de name representado como lista"
        (first name))
```

Las razones de hacerlo son: legibilidad y mantenimiento.

```
> (setf nombres '((Sir Antonio Lopez) (Señor Pepe Galvez) (Don Pepe Cuena) (Mr Carlos Sierra)))
((Antonio Lopez) (Pepe Galvez) (Pepe Cuena) (Carlos Sierra))
> (mapcar #'last-name nombres)
(LOPEZ GALVEZ CUENA SIERRA)
```

La notación #' relaciona el nombre de la función con la función misma. En este sentido es similar con la notación 'x. La llamada a mapcar anterior es equivalente a

```
(list (last-name (first nombres))
(last-name (second nombres))
...)
```

Mapcar viene de "to map" una función sobre los sucesivos car de los argumentos. Car viene de "contents of the address register" y cdr de "contents of the decrement register" las instrucciones utilizadas en la primera implementación de LISP sobre el IBM 704.

```
> (mapcar #'- '(1 2 3 4))
(-1 -2 -3 -4)
> (mapcar #'+ '(1 2 3 4) '(10 20 30 40))
(11 22 33 44)
```

Funciones Orden Superior

Las funciones en LISP se pueden crear y llamar, como ya hemos visto, pero también se pueden manipular como cualquier otro tipo de objeto. Ya hemos visto un ejemplo de programación orden superior: mapcar. Veamos un ejemplo

```
(defun mappend (fn lista)
        "Aplica fn a todos los elementos de lista y hace un append de los resultados"
        (apply #'append (mapcar fn lista)))
```

veamos como funciona el apply

```
> (apply #'+ '(1 2 3 4))
10
> (apply #'append '((1 2 3) (a b c)))
(1 2 3 a b c)
```

Definimos una función auxiliar

```
> (defun self-and-double (x) (list x (+ x x)))
> (self-and-double 3)
(3 6)
> (apply #'self-and-double '(3))
(3 6)
> (mapcar #'self-and-double '(1 10 20))
((1 2) (10 20) (20 40))
> (mappend #'self-and-double '(1 10 20))
(1 2 10 20 20 40)
```

Funcall and Lambda

funcall es similar a apply con la diferencia que no lista sus argumentos.

```
> (funcall #'+ 2 3)
5
> (apply #'+ '(2 3))
5
> (funcall #'+ '(2 3))
Error (2 3) is not a number
```

Es posible definir funciones sin darles nombre. Esto se hace con la sintaxis especial Lambda.
En general (lambda (parameters ...) body ...). Una expresión lambda es un nombre no atómico para una función. Así su uso es exacto al de cualquier otra función

```
> ((lambda (x) (+ x 2)) 4)
6
> (funcall #'(lambda (x) (+ x 2)) 4)
6
```

Más

La forma de evaluación es un poco más complicada de lo explicado hasta ahora. Si el primer elemento de una lista es una forma especial, la expresión se evalúa bajo las reglas de esa forma especial. Si no, el primer argumento es evaluado como una función, pudiendo ser un símbolo o una función lambda, el resto de elementos se evalúan como se ha explicado hasta ahora. Cuando una función aparece en una posición diferente de la primera debemos utilizar #' si no las expresiones serán evaluadas de forma normal y no serán tratadas como funciones.

```
> append
Error: APPEND is not a bound variable
> (lambda (x) (+ x 2))
Error: Lambda is not a function
```

Hay dos razones que justifican la existencia de funciones lambda (sin nombre)

• Provoca confusión crear nombres innecesarios de funciones en un programa.

• Nos permite crear funciones en tiempo de ejecución! Esta herramienta de programación es muy potente y no es posible usarla en la mayoría de lenguajes. Estas funciones de tiempo de ejecución se denominan clausuras o cierres.

Formas especiales para definiciones

Existen varias. Para definir funciones, estructuras, macros, constantes, variables, etc. Nos centraremos en las dos primeras.

• Funciones

```
(defun function-name (parameter ...) "opt. doc." body...)
```

• Estructuras

```
(defstruct structure-name "opt. doc." slot...)
```

Todas las formas def- definen símbolos globales. Para definir variables y locales es necesario utilizar let.

Estructuras

Veamos las estructuras

```
(defstruct name first (middle nil) last)
```

Se definen automáticamente la función constructora make-name, el predicado name-p, y las funciones accesoras name-first, name-middle y name-last

```
> (setf b (make-name :first 'pepe :last 'cuena))
#S(NAME :FIRST PEPE :LAST CUENA)
> (name-first b)
pepe
> (name-middle b)
nil
> (name-last b)
cuena
> (name-p b)
T
> (name-p 'pepe)
NIL
> (setf (name-middle b) 'Nolose)
nolose
> b
#S(NAME :FIRST PEPE :MIDDLE NOLOSE :LAST CUENA)
```

Formas especiales para condicionales

Hemos visto el if. Ahora veremos el cond y como las demás pueden expresarse como estas

```
(cond    (test result...)
         (test result...)
         ...)
```

Evaluación secuencial de los *tests*, cuando uno evalúa diferente de nil se evalúa el (los) *result* a continuación. Si ninguno evalúa no-nil el resultado es nil. Se devuelve el resultado de la última expresión evaluada.

```
(when test a b c)
         (if test (progn a b c))
         (cond (test a b c))

(unless test x y)
         (if (not test) (progn x y))
         (cond ((not test) x y))
(and a b c)
         (if a (if b c))
         (cond (a (cond (b c))))
(or a b c)
         (if a a (if b b c))
         (cond (a) (b) (c))
(case a (b c) (t x))
         (if (eql a 'b) c x)
         (cond ((eql a 'b) c) (t x))
```

Formas especiales para manejar variables y posiciones

La forma especial setf es la utilizada para asignar valores a una variable o a una posición (variable generalizada).

LISP	PASCAL
(setf x 0)	x := 0;
(setf (aref A i j) 0)	A[i,j] := 0;
(setf (rest list) nil)	list^.rest := nil;
(setf (name-middle b) 'Q)	b^.middle := "Q";

Para asignar variables es muy común el uso de setq. En cualquier caso la programación funcional pura no permite la asignación, y no es extraño ver programas en los que esta no aparece. La forma habitual es el uso de la vinculación en lugar de la asignación. Es decir usar parámetros de funciones o variables locales dentro de un let.

```
(let ( (x 40)
       (y (+ 1 1)))
       (+ x y))

((lambda (x y)
         (+ x y))
         40 (+ 1 1))
```

Funciones y formas especiales para repetición

Para ver las diferencias haremos versiones de la función length.

(dolist (variable list optional-result) body...)

```
(defun length1 (lista)
         (let ((len 0))
```

```
        (dolist (element list)
                (setf len (+ len 1)))
        len))
```
dolist iter. sobre los elem. de una lista
dotimes iter. sobre enteros sucesivos
do, **do*** iter. general.
loop iter. general.
mapc, **mapcar** iter. sobre elem. de una lista
some, **every** iter. sobre lista hasta condición
find, **reduce**, ... funciones más específicas
recursión repetición general.

mapc tiene dos argumentos, uno una función y el otro una lista. Aplica la función a cada elemento de la lista.

```
(defun length2 (lista)
    (let ((len 0))
        (mapc #'(lambda (element)
                (incf len))
            lista)
        len))

(dotimes (variable number optional-result) body...)
```
Dotimes no es apropiado para length, por supuesto.

```
(do ((variable initial next) ...)
    (exit-test result)
body...)

(defun length3 (lista)
    (do ((len 0 (+ len 1))
        (l list (rest list)))
    ((null l) len)))
```

Muchas formas de iteración corresponden a esquemas de programación, como hemos visto con los casos de dolist o dotimes. LISP proporciona flexibilidad para construir los esquemas más adecuados. La flexibilidad viene dada por tres elementos.

1) Aplicación a un número arbitrario de listas.

```
> (mapcar #'- '(1 2 3)
(-1 -2 -3)
> (mapcar #'+ '(1 2) '(10 20)
(11 22)
> (mapcar #'+ '(1 2) '(10 20) '(100 200))
(111 222)
```

2) Palabras clave que varían las comprobaciones

```
> (remove 1 '(1 2 3 2 1 0 -1))
(2 3 2 0 -1)
> (remove 1 '(1 2 3 2 1 0 -1) :key #'abs)
(2 3 2 0)
> (remove 1 '(1 2 3 2 1 0 -1) :test #'<)
(1 1 0 -1)
> (remove 1 '(1 2 3 2 1 0 -1) :start 4)
(1 2 3 2 0 -1)
```

3) Predicados en lugar de elementos

```
> (remove-if #'oddp '(1 2 3 2 1 0 -1))
(2 2 0)
> (remove-if-not #'oddp '(1 2 3 2 1 0 -1))
```

```
(1 3 1 -1)
> (find-if #'evenp '(1 2 3 2 1 0 -1))
2
```

Ahora usaremos los siguientes valores

```
(setf x '(a b c))
(setf y '(1 2 3))
```

Las siguientes funciones no aceptan palabras clave.

```
>(every #'oddp y)
nil ; comprueba que cada elemento satisface el predicado
>(some #'oddp y)
t ; comprueba si algún elemento satisface el predicado
>(mapcar #'- y)
(-1 -2 -3) ; devuelve la lista resultado de aplicar la función a cada elemento
```

Las siguientes funciones sí que permiten las palabras clave.

```
>(member 2 y)
(2 3) ;ver si el elemento está en la lista
>(count 'b x)
1 ;contar el número de apariciones
>(delete 1 y)
(2 3) ;eliminar elementos
>(find 2 y)
2 ;encontrar elementos
>(position 'a x)
0 ;posición de un elemento
>(remove 2 y)
(1 3) ;como delete pero con copia
>(substitute 4 2 y)
(1 4 3) ;reemplazar elementos
```

Repetición vía recursión

Una de las características que históricamente caracterizó a LISP fue la posibilidad de permitir recursión. La recursión es un mecanismo general de realizar repetición.

```
(defun length9 (lista)
    (if (null lista) 0 (+ 1 (length9 (rest list)))))
```

Las versiones recursivas sobre listas son fáciles de ver debido a la definición recursiva de las mismas. Una lista es o bien vacía o bien un elemento concatenado a una lista. La ineficiencia es una objeción clásica a la recursión. Véase

```
(defun length10 (list) (length10-aux list 0))

(defun length10-aux (sublist len-so-far)
    (if (null sublist)
    len-so-far
    (length10-aux (rest sublist) (+ 1 len-so-far))))
```

La segunda versión es recursiva por la cola. Los compiladores pueden mejorar la eficiencia de estas funciones.

Las dos versiones pueden combinarse de forma elegante en CommonLisp

```
(defun length11 (lista &optional (len-so-far 0))
    (if (null lista) len-so-far (length11 (rest lista) (+ 1 len-so-far))))
```

Otras formas especiales

Hay otras formas especiales que no encajan en los tipos anteriores hemos visto ya 'x es equivalente a (quote x) y #'f es equivalente (function f)

```
> (progn (setf x 0) (setf x (+ x 1)) x)
1
> (trace length9)
(length9)
> (length '(a b c))

(1 ENTER LENGTH: (A B C))
(2 ENTER LENGTH9: (B C))
(3 ENTER LENGTH9: (C))
(4 ENTER LENGTH9: NIL)
(4 EXIT LENGHT9: 0)
(3 EXIT LENGTH9: 1)
(2 EXIT LENGTH9: 2)
(1 EXIT LENGTH9: 3)
3
>

> (untrace length9)
(length9)
> (length9 '(a b c))
3
```

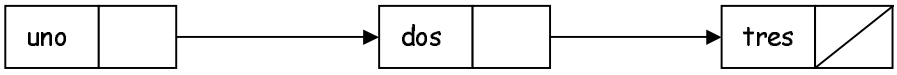
Funciones sobre listas

En la siguiente tabla usaremos

```
>(setf x '(a b c))
>(setf y '(1 2 3))
>(first x)
a ;primer elemento de las lista. (car x)  es equivalente a (first x)
>(second x)
b ;segundo elemento de la lista
>(third x)
c ;tercer elemento de la lista
>(nth 0 x)
a ;n-ésimo elemento de la lista
>(rest x)
(b c) ;todos los elementos de la lista menos primero. (cdr x) es equivalente a (rest x)
>(last x)
(c) última celda cons de una lista
>(length x)
3 número de elementos de una lista
>(reverse x)
(c b a) lista invertida
>(cons 0 y)
(0 1 2 3); operación de construcción de listas
>(append x y)
(a b c 1 2 3) ; concatenar dos listas
>(list x y)
((a b c) (1 2 3)) ; hacer una nueva lista
>(null nil)
t ;  es cierto en listas vacías
>(null x)
```

```
nil;  y falso en otro caso
>(listp x)
t ; es cierto para una lista
>(listp 3)
   nil ;y falso en otro caso
>(equal x x)
t ;cierto para listas iguales
>(equal x y)
nil; y falso para listas diferentes
>(sort y #'>)
(3 2 1) ;ordena una lista según un criterio
>(first (cons a b))
a
>(rest (cons a b))
b
```

La representación de las listas se basa en las cedas cons. Por ejemplo, la lista (1 2 3) se corresponde con:



Igualdad

LISP tiene predicados especializados para la igualdad de sus diferentes tipos de datos. Algunos de ellos son:

1) = se utiliza sólo para números.

- Comprueba si sus dos argumentos tienen el mismo valor numérico. Sólo se comprueba el valor numérico sin tener en cuenta el tipo de número (real, entero, etc.)
- En el caso de que los valores coincidan el predicado devuelve t, en caso contrario devuelve nil

```
> (= 4.0 4)
T
```

2) eq es el predicado que suele utilizarse para determinar si dos átomos no numéricos son iguales.

- Determina si dos objetos ocupan la misma posición de memoria.
- En caso afirmativo devuelve t y en otro caso nil.

```
> (eq ‘a ‘a)
T
> (eq ‘(1 2 3) ‘(1 2 3))
NIL
```

3) eql. El valor eql de dos objetos x e y, devuelve T en los siguientes casos

- Si (eq x y) es T
- Si x e y son números del mismo tipo y con el mismo valor.
- Si x e y son caracteres que representan el mismo carácter.

```
>(eql 'a 'b)
NIL
>(eql 'a 'a)
```

```
T
>(eql 3 3)
T
 >(eql 3 3.0)
NIL
> (eql 3.0 3.0)
T
> (eql #\A #\A)
T
```

4) equal. Es el predicado más adecuado para comprobar la igualdad de listas. Si los argumentos no son listas se comporta como el predicado eql, en el caso de que sean listas se comprueba el cumplimiento del predicado equal tanto con el componente car (first) como con el componente cdr (rest) de sus dos argumentos para devolver el valor verdadero.

```
>(equal 'a 'b)
NIL
>(equal 'a 'a)
T
 > (equal 3 3.0)
NIL
>(equal ‘(1 2 3) ‘(1 2 3))
T
```

Funciones numéricas

```
>(+ 4 2)
6
>(- 4 2)
2
>(* 4 2)
8
>(/ 4 2)
2
>(> 100 99)
T ; mayor que (también >=)
>(= 100 100)
T ; igual que (también /=)
>(< 99 100)
T ; menor que (también <=)
>(random 100)
42 ; números enteros aleatorios
>(expt 4 2)
16 ; exponenciación (tambien exp, y log)
>(min 2 3 4)
2 ; mínimo (también max)
>(abs -3)
3; valor absoluto
>(sqrt 4)
2 ; raíz cuadrada
```

Funciones destructivas

Las funciones matemáticas sólo calculan resultados a partir de valores pero no "hacen" nada. Algunas funciones de LISP pueden "hacer" algo aparte de la propia computación. En otros lenguajes a estas funciones se las denomina procedimientos, y pueden causar graves trastornos si no se utilizan con precaución. A veces, en cambio, son de gran utilidad.

```
> (setf x '(a b c))
(a b c)
> (setf y '(1 2 3))
(1 2 3)
> (append x y)
(a b c 1 2 3)
```
append es una función pura. no modifica ni x ni y, en cambio.
```
> (nconc x y)
(a b c 1 2 3)
> x
(a b c 1 2 3)
> y
(1 2 3)
```
calcula exactamente lo mismo que append pero modifica el primer argumento. Se llama destructiva, dado que destruye estructuras preexistentes modificándolas por nuevas. La ventaja es el espacio de almacenamiento ahorrado. Otras funciones destructivas son: nreverse, nintersection, nunion, nsetdifference y nsubst.
Un caso especial es delete, versión destructiva de remove.

Entrada/salida

La entrada en LISP es muy fácil ya que el lenguaje provee al usuario de un parser completo cuyo nombre es read. Se utiliza para leer y devolver expresiones LISP. Si la entrada se puede adaptar para que pueda escribirse en formato de expresiones LISP, los problemas de entrada se han acabado. Para leer desde el terminal las funciones read, read-char y read-line devuelven una expresión, un carácter y una cadena respectivamente. Para leer de un fichero la función with-open-stream es la más comúnmente utilizada. Asocia un stream a un fichero. Las funciones de lectura tienen un parámetro opcional que es precisamente el nombre de un stream. Las funciones de salida son print que imprime cualquier objeto en una nueva línea dejando un blanco después. prin1 es igual pero sin salto de línea ni blanco. Los formatos generados por estas funciones son legibles por las funciones de entrada. La función más utilizada es el format (todo un lenguaje de escritura).
```
> (with-open-file (stream "mifichero.txt" :direction :output)
        (print '(hello there) stream)
        (prin1 'goodbye stream))

> (with-open-file (stream "mifichero.txt" :direction :input)
        (list (read stream) (read-char stream) (read stream)
                (read stream nil 'eof)))
  ((HELLO THERE) #\Space GOODBYE EOF)

>(format t “primera linea ~&~a” ‘(1 2 3))
primera linea
(1 2 3)
NIL
```

Herramientas de depuración

En la mayoría de lenguajes hay dos estrategias de depuración:
(1) editar el programa e insertar sentencias de impresión
(2) utilizar un programa depurador que altere el estado interno en tiempo de ejecución.
En LISP hay una tercera que es (3) hacer anotaciones que no son parte de programa pero que automáticamente alteran la ejecución del mismo.
Este es el caso de trace y untrace que ya hemos visto. Otro caso es step. (step expression). Evalúa la expresión dando información de cada paso de evaluación.
Otras funciones son:
(apropos 'string) Devuelve todos los símbolos que hacen matching con string.

(describe 'make-string) Da información sobre el símbolo.

Valores múltiples

Hay funciones en Lisp que no se comportan como funciones en el sentido que no devuelven un único valor. Por ejemplo

```
>(round 5.1)
5 .1
```

Hay dos valores después de la flecha porque round devuelve el entero y el resto.

La mayoría de las veces los valores múltiples son ignorados y sólo se considera el primero de ellos.

```
(* 2 (round 5.1))
10
```

Si se quieren obtener todos se debe hacer con la forma especial multiple-value-bind.

```
(defun show-both (x)
        (multiple-value-bind (int rem)
        (round x)
        (format t "~f = ~d + ~f" x int rem)))
> (show-both 5.1)
5.1 = 5 + 0.1
```

También se pueden construir funciones que devuelvan múltiples valores.

```
>(values 1 2 3)
1 2 3
```