Guía rápida de Common Lisp

César Ignacio García Osorio, 2000*

1 Símbolos

`nil` Constante cuyo valor es ella misma (`NIL`).

`t` Constante cuyo valor es ella misma (`T`).

`(symbolp e)`
Devuelve `T` si *e* se evalúa a un símbolo, `NIL` en caso contrario.

`(boundp e)`
Devuelve `NIL` si el valor de *e* (que debe ser un símbolo) no tiene un valor ni global ni local.

`(defvar simb e)`
Define *simb* como variable global con valor inicial *e*.

`(defparameter simb e)`
Define *simb* como parámetro global cuyo valor (inicialmente *e*) puede cambiar en tiempo de ejecución pero no se espera que lo haga.

`(defconstant simb e)`
Define *simb* como constante global cuyo valor (*e*) no cambiará durante la ejecución del programa.

2 Asignación de valores

`(setf lugar e)`
Almacena el valor de *e* en el lugar especificado por *lugar*

`(setq simb e)`
Evalúa *e* y hace que su valor se almacene en *simb*; retorna el valor de *e*.

3 Entrada/Salida

`(read [stream])`
Lee de *stream* (por defecto `*standard-input*`) la representación escrita de un objeto, construye el objeto correspondiente y lo retorna.

`(read-line [stream eof-error-p eof-value recursive-p])`
Lee de *stream* (por defecto `*standard-input*`) una línea de texto terminada por los caracteres `newline` o `end-of-file` y retorna dos valores: la línea como una cadena de caracteres y un valor booleano, `T` si la línea terminaba en `end-of-line` y `NIL` si terminaba por `newline`.

`(read-char [stream eof-error-p eof-value recursive-p])`
Lee de *stream* (por defecto `*standard-input*`) un carácter, y retorna el correspondiente objeto carácter.

`(read-char-no-hang [stream eof-error-p eof-value recursive-p])`
Lee de *stream* (por defecto `*standard-input*`) un carácter si esta disponible, en caso contrario devuelve `NIL`.

`(read-byte binary-stream [eof-error-p eof-value recursive-p])`
Lee un byte del stream binario de entrada *binary-stream* y lo devuelve en forma de entero.

`(read-from-string str [eof-err eof-val rec :start st :end end :preserve-whitespace pr-wh])`
Lee y retorna una expresión, tomando como entrada la cadena *str*. Retorna un segundo valor que indica el índice del primer carácter en la cadena que no se ha leído.

`(prin1 e [stream])`
Escribe en *stream* (por defecto `*standard-output*`) una representación de *e*.

*Traduciendo, corrigiendo y ampliando la recopilación de W.Burger, WS 95/96

`(print e [stream])`
: Escribe en *stream* (por defecto `*standard-output*`) una representación de e al principio de una línea.

`(format stream control-string [`$e_1 \ldots e_k$`])`
: Escribe una salida formateada en *stream* usando *control-string* y los argumentos $e_1 \ldots e_k$.

`(fresh-line [stream])`
: Escribe un carácter de nueva línea (`newline`) en *stream* (por defecto `*standard-output*`) a no ser que *stream* este ya a principio de línea.

`(write-char char [stream])`
: Escribe el carácter *char* en *stream* y devuelve *char*.

`(write-byte integer binary-stream)`
: Escribe el byte de valor *integer* en *binary-stream*.

`(with-open-file (stream-var filename options)` $e_1 \ldots e_k$`)`
: Fija un stream de entrada (por defecto) o de salida como el fichero *filename*. A continuación evalúa $e_1, \ldots, e_k$ con *stream-var* ligado al stream abierto y por último cierra el stream.

`(load filename)`
: Carga el fichero *filename* en el interprete.

4 Listas

`(list [`$e_1 \ldots e_k$`])`
: Crea una lista que contiene los elementos $e_1, \ldots, e_k$.

`(cons e lst)`
: Crea una lista con e como primer elemento y *lst* como restantes elementos.

`(make-list size)`
: Crea una lista con *size* elementos inicializados a `NIL` (hay otras opciones disponibles).

`(length lst)`
: Devuelve el número de elementos (de alto nivel) de la lista *lst*.

`(listp lst)`
: Devuelve `T` si *lst* es una lista (incluso si es la lista vacía).

`(first lst)`
: Devuelve el primer elemento de la lista *lst* (`NIL` si se trata de la lista vacía); es equivalente a `car`.

`(second lst)`
: Devuelve el segundo elemento de la lista *lst* (`NIL` si se trata de la lista vacía)[1].

`(rest lst)`
: Devuelve la lista que contiene todos los elementos de *lst* salvo el primero (`NIL` si la longitud de *lst* es menor de 2); es equivalente a `cdr`[2].

`(nth n lst)`
: Devuelve el elemento de *lst* que se encuentre en la posición n-ésima (el primer elemento tiene la posición 0). Retorna `NIL` si n se sale del rango.

`(last lst [n])`
: Devuelve una lista que contiene los n últimos elementos (si no se específica el número sólo 1) de la lista *lst* o `NIL` si *lst* es la lista vacía.

`(butlast lst [n])`
: Devuelve una copia de la lista *lst* pero sin los n (por defecto 1) últimos elementos.

`(append` lst_1 $lst_2 \ldots lst_k$ `)`
: Devuelve la lista concatenación de las listas lst_1, $lst_1, \ldots,$ lst_1.

`(member item lst )`
: Devuelve `T` (en realidad la porción de *lst* que comienza por *item*) si *item* ocurre como elemento de alto nivel en cualquier posición de *lst*.

`(remove item lst )`
: Devuelve una lista similar a *lst* pero con todos los elementos que son `eql` con *item* eliminados.

`(remove item lst )`
: Devuelve una lista similar a *lst* pero con todos los elementos que son `eql` con *item* eliminados.

`(copy-list lst )`
: Devuelve una copiar de alto nivel de *lst*.

[1] Otras funciones análogas son `third`, `fourth`... `tenth`.

[2] Existen abreviaturas para todas las combinaciones posibles de `car` y `cdr` hasta un número de cuatro, algunos ejemplos: `caar`, `cadr`, `cdar`, ..., `cddddr`.

`(subseq` *lst start* `[`*end*`])`
: Crea una nueva lista usando los valores de *lst*. Si *start* se evalúa a 0 y no se da *end* se copia toda la lista.

`(reverse` *lst*`)`
: Devuelve una lista con los mismos elementos pero en orden inverso.

`(push` *item place*`)`
: Inserta *item* al principio de la lista almacenada en *place* y la vuelve a almacenar de nuevo en *place*.

`(pop` *place*`)`
: Devuelve el primer elemento de la lista almacenada en *place* y almacena la lista sin su primer elemento de nuevo en *place*.

5 Predicados

`(atom` *e*`)`
: Devuelve `T` si *e* se evalúa a un átomo y `NIL` en caso contrario. Por tanto `atom` es `T` si el valor de *e* no es un cons.

`(null` *e*`)`
: Devuelve `T` si *e* se evalúa a `NIL`, en caso contrario devuelve `NIL`.

`(consp` *lst*`)`
: Devuelve `T` si *lst* es realmente una lista (es decir una lista no vacía).

`(endp` *lst*`)`
: Devuelve `T` si *lst* se evalúa a `NIL`, y `NIL` si es una lista no vacía.

`(typep` *object type*`)`
: Devuelve `T` si el valor en *object* es de tipo *type*, `NIL` en caso contrario.

6 Tests de igualdad

`(eq` e_1 e_2`)`
: Devuelve `T` si y sólo si e_1 y e_2 son el mismo objeto (misma posición de memoria). Un test muy eficiente para comprobar la igualdad de símbolos, pero inútil cuando se pretende comprobar la igualdad de listas, números u otros objetos de Lisp. Los objetos que tienen la misma representación escrita no son necesariamente `eq`, números con el mismo valor no tienen porque ser `eq`, y dos listas de igual representación no tienen porque ser la misma lista y por tanto no son `eq`.

`(eql` e_1 e_2`)`
: Devuelve `T` si e_1 y e_2 son `eq` o si son número del mismo tipo y valor, o si son objetos carácter que representan el mismo carácter.

`(equal` e_1 e_2`)`
: Devuelve `T` si e_1 y e_2 son estructuralmente similares. Dos objetos son `equal` si su representación escrita es la misma. Cuando se comparan cadenas este test diferencia mayúsculas y minúsculas.

`(equalp` e_1 e_2`)`
: Devuelve `T` si e_1 y e_2 son `equal`, son caracteres que satisfacen el test `char-equal` o si son números con el mismo valor aunque sean de tipos diferentes. Se aplican reglas especiales para el caso de los arrays, las tablas de hash y las estructuras (consultar la especificación completa de Common Lisp).

7 Aritmética

`(+` n_1 $n_2 \ldots n_k$`)`
: Devuelve $n_1 + n_2 \ldots + n_k$.

`(-` n_1 $n_2 \ldots n_k$`)`
: Devuelve n_1 - n_2 ...- n_k.

`(*` n_1 $n_2 \ldots n_k$`)`
: Devuelve n_1 * n_2 ...* n_k.

`(/` n_1 $n_2 \ldots n_k$`)`
: Devuelve $(\ldots(n_1 \,/\, n_2) \ldots / \, n_k)$.

`(decf` *place* `[`*delta*`])`
: Sustrae *delta* (que por defecto vale 1) al valor almacenado en *place* y almacena el nuevo valor en la misma localización.

`(incf` *`place`* `[`*`delta`*`])`
Como `decf` pero añadiendo el valor de *delta*.

`(max` n_1 `[`$n_2 \ldots n_k$`])`
Devuelve el valor máximo de todos sus argumentos.

`(min` n_1 `[`$n_2 \ldots n_k$`])`
Devuelve el valor mínimo de todos sus argumentos.

`(signum` n`)`
Devuelve 1 si el valor de n es positivo, 0 si su valor es 0 y -1 si es negativo. El resultado es del mismo tipo que n.

`(abs` n`)`
Devuelve el valor absoluto de n.

`(sqrt` n`)`
Devuelve la raíz cuadrada de n.

`(exp` n`)`
Devuelve e^n.

`(expt` *`base`* n`)`
Devuelve $base^n$.

`(log` n `[`*`base`*`])`
Devuelve el logaritmo en base *base* (que por defecto vale e) de n.

`(cos` n`)(sin` n`)(tan` n`)`
Devuelven el coseno, el seno y la tangente de n (que se supone dado en radianes).

`(acos` n`)(asin` n`)(atan` n`)`
Devuelven en radianes el arcocoseno, el arcoseno y el arcotangente de n.

`(float` n`)`
Devuelve el valor de n como un número en punto flotante.

`(random` n`)`
Devuelve un número aleatorio entre 0 (inclusive) y el valor de n (pero nunca n).

`(mod` *`número divisor`*`)`
Lleva a cabo la operación `floor` sobre sus argumentos y devuelve el segundo valor de `floor`.

`(rem` *`número divisor`*`)`
Lleva a cabo la operación `truncate` sobre sus dos argumentos y devuelve el segundo valor de `truncate` como su único valor. La diferencia con `mod` se muestra a continuación:

```
Argumento1  Argumento2    mod    rem
----------------------------------------
 13           4            1      1
-13          -4            3     -1
 13          -4           -3      1
-13          -4           -1     -1
 13.4         1          0.4    0.4
-13.4         1          0.6   -0.4
----------------------------------------
```

`(floor` n `[`*`dvsor`*`]) (ceiling` n `[`*`dvsor`*`]) (truncate` n `[`*`dvsor`*`]) (round` n `[`*`dvsor`*`])`
Todas estas si toman un sólo argumento y es entero lo devuelven tal cual. Si es real o racional devuelve el entero "mas próximo" siguiendo distintos criterios[3]. Si hay un segundo argumento antes de llevar a cabo la aproximación se efectúa la división entre dicho argumento[4].

```
Argumento   floor       ceiling     truncate    round
-------------------------------------------------------
 2.6          2           3           2           3
 2.5          2           3           2           2
 2.4          2           3           2           2
 0.7          0           1           0           1
 0.3          0           1           0           0
-0.3         -1           0           0           0
-0.7         -1           0           0          -1
-2.4         -3          -2          -2          -2
-2.5         -3          -2          -2          -2
-2.6         -3          -2          -2          -3
-------------------------------------------------------
```

`pi` Una constante cuyo valor es la aproximación en punto flotante a π.

[3]En realidad devuelven un segundo valor que es la diferencia entre el argumento y el primer valor.
[4]Por ejemplo, `(floor 5 2) == (floor (/ 5 2))`.

8 Comparación y predicados sobre números

`(= `$n_1 \ldots n_k$`)`
: Devuelve `T` si todos sus argumentos son numéricamente iguales, en caso contrario `NIL`.

`(/= `$n_1 \ldots n_k$`)`
: Devuelve `T` si todos sus argumentos son diferentes, en caso contrario `NIL`.

`(< `$n_1 \ldots n_k$`)`
: Devuelve `T` si cada argumento es menor que el siguiente, en caso contrario `NIL`.

`(<= `$n_1 \ldots n_k$`)`
: Devuelve `T` si cada argumento es menor o igual que el siguiente, en caso contrario `NIL`.

`(> `$n_1 \ldots n_k$`)`
: Devuelve `T` si cada argumento es mayor que el siguiente, en caso contrario `NIL`.

`(>= `$n_1 \ldots n_k$`)`
: Devuelve `T` si cada argumento es mayor o igual que el siguiente, en caso contrario `NIL`.

`(zerop `*n*`)`
: Devuelve `T` si el número n es cero (independientemente de su tipo, entero, punto flotante o complejo), `NIL` en caso contrario[5].

`(plusp `*n*`)`
: Devuelve `T` si el número n es estrictamente mayor que cero, `NIL` en caso contrario.

`(minusp `*n*`)`
: Devuelve `T` si el número n es estrictamente menor que cero, `NIL` en caso contrario.

`(numberp `*e*`)`
: Devuelve `T` si e se evalúa a cualquier tipo de número de Common Lisp.

9 Operadores lógicos

`(and [`$e_1 \ldots e_k$`])`
: Evalúa cada argumento e_i secuencialmente. Si se llega a un argumento que devuelve `NIL`, devuelve `NIL` sin evaluar el resto de argumentos. Si llega al *último* argumento devuelve el valor de su evaluación.

`(or [`$e_1 \ldots e_k$`])`
: Evalúa cada argumento e_i secuencialmente. Si se llega a un argumento que no devuelve `NIL`, devuelve el valor de dicho argumento sin evaluar el resto de argumentos. Si llega al *último* argumento devuelve el valor de su evaluación.

`(not `*e*`)`
: Devuelve `T` si el valor de e es `NIL`, en caso contrario devuelve `NIL`.

10 Constructores de bloques

`(progn [`$e_1 \ldots e_n$`])`
: Evalúa las formas e_1 a e_n y retorna el valor de e_n.

`(prog1 `e_1 `[`$e_2 \ldots e_n$`])`
: Evalúa las formas e_1 a e_n y retorna el valor de e_1.

`(let (`$lvl_1 \ldots lvl_k$`) `$e_1 \ldots e_n$`)`
: Lleva a cabo las ligaduras de variable local $lvl_1 \ldots lvl_k$, evalúa e_1 a e_n y retorna el valor de e_n (como si hubiera un `progn` implícito).

`(let* (`$lvl_1 \ldots lvl_k$`) `$e_1 \ldots e_n$`)`
: Lleva a cabo las ligaduras de variable local $lvl_1 \ldots lvl_k$ *secuencialmente*, el resto como `let`.

11 Constructores de condicionales

`(if `*test* e_1 `[`e_2`])`
: Evalúa *test* y si no es `NIL` evalúa e_1. En caso contrario evalúa e_2 si existe.

`(when `*test* $e_1 \ldots e_n$`)`
: Evalúa *test* y si no es `NIL` evalúa las formas e_1 a e_n. Si *test* es `NIL` devuelve `NIL` en caso contrario el resultado de evaluar e_n (como si hubiera un `progn` implícito).

[5] `(zerop -0.0)` es siempre cierto

`(unless` *test* $e_1 \ldots e_n$`)`
Evalúa *test* y es `NIL` evalúa las formas e_1 a e_n. Si *test* no es `NIL` devuelve `NIL` en caso contrario el resultado de evaluar e_n (como si hubiera un `progn` implícito).

`(cond (`$test_1$ $e_{11} \ldots e_{1n}$`)...(`$test_k$ $e_{k1} \ldots e_{kn}$`))`
Evalúa $test_1$, ..., $test_k$ hasta que algún $test_i$ se evalúa a algo distinto `NIL`, entonces evalúa $e_{i1}, \ldots, e_{in}$ como si hubiera un `progn` implícito.

`(case` *keyform* `(`key_1 $e_{11} \ldots e_{1n}$`)...(`key_k $e_{k1} \ldots e_{kn}$`))`
Evalúa *keyform*, a continuación evalua como si hubiera un `progn` implícito las formas e_{ij} para las cuales key_i coincide con el resultado de evaluar *keyform* o se trata de una lista dentro de la que existe el valor al que se evalúo *keyform*. Devuelve la última forma evaluada. Permite la aparición de una última cláusula con clave `otherwise` o `t` cuyas formas se evaluaran si no ha habido coincidencia con ninguna de las otras cláusulas.

12 Constructores de iteraciones

`(dolist (`*var lst* `[`*result*`]) `e_1 e_k`)`
Evalúa $e_1 \ldots e_k$ para cada elemento en la lista *lst*. La variable *var* se liga al valor del elemento en curso de la lista. Al finalizar, si existe, se evalúa la forma *result* y su valor es devuelto.

`(dotimes (`*var count* `[`*result*`]) `e_1 e_k`)`
Evalúa $e_1 \ldots e_k$ *count* veces. La variable *var* es la variable contador del bucle y se liga a los valores que van de 0 a *count*-1. Finalmente, si se ha especificado la forma *result* se evalúa y devuelve su resultado.

`(do ((`var_1 `[`ini_1 `[`act_1`]]])...(`var_k `[`ini_k `[`act_k`]]]))(`*test* t_1 `...`t_n`) `e_1 `...`e_m`)`
En primer lugar, las variables var_i del índice quedan localmente ligadas al valor de ini_i (por defecto `NIL`). La ligadura y asignación se realiza en paralelo. A continuación comienza la iteración. Cada ciclo realiza los siguientes pasos:

1. Se evalúa test de fin *test*. Si el resultado no es `NIL` finaliza la iteración. Se evalúan secuencialmente las formas $t_1 \ldots t_k$ y se devuelve el valor de t_k (`NIL` si no hay ninguna).
2. Si el valor de *test* es `NIL` se evalúan secuencialmente las formas del cuerpo $e_1 \ldots e_m$.
3. Cuando se alcanza el fin del cuerpo, se asigna a cada var_i el valor de act_i. Si no se proporciona act_i, var_i no es actualizada.

`(do* ((`var_1 `[`ini_1`[`act_1`]])...)(`*test* `...)  ...)`
Como `do`, pero las variables de índice se ligan secuencialmente.

`(mapcar (`*fun* $arglst_1$ `[`$arglst_1$ $arglst_k$`]))`
Aplica la función *fun* a los sucesivos elementos de las listas de argumentos $arglst_i$ y retorna la lista de resultados. Si hay k listas de argumentos, *fun* debe ser una función con k argumentos.

`(mapcan (`*fun* $arglst_1$ `[`$arglst_1$ $arglst_k$`]) )`
Funciona como `mapcar` pero utiliza `nconc` para construir la lista con los resultados, por tanto, el resultado de aplicar *fun* a sus argumentos debe ser de tipo lista. Se da la siguiente equivalencia:

`(mapcon` *fun* $lst_1 \ldots lst_n$`)` $\equiv$ `(apply #'nconc (maplist` *fun* $lst_1 \ldots lst_n$`))`

13 Definición de funciones

`(defun` *name* `(`$a_1 \ldots a_k$`) `$e_1 \ldots e_n$`)`
Define una función con nombre con argumentos $a_1 \ldots a_k$ y cuerpo $e_1 \ldots e_n$. En la especificación de los argumentos son posibles multitud de opciones.

`(lambda (`$a_1 \ldots a_k$`) `$e_1 \ldots e_n$`)`
Define una función sin nombre con argumentos $a_1 \ldots a_k$ y cuerpo $e_1 \ldots e_n$. En la especificación de los argumentos son posibles multitud de opciones.

`(function` *fun*`)`
Devuelve el objeto función asociado con *fun*. Si *fun* es un símbolo, devuelve la definición de la función asociada. Si *fun* es una expresión lambda devuelve su cerradura léxica. Como se usa con frecuencia se puede abreviar como `#'`, por tanto, `#'f` es lo mismo que `(function f)`.

14 Sobre la evaluación

`(apply` *function* `[`$a_1 \ldots a_k$`]`*arglist*`)`
Aplica el valor de *function* a la lista de todos los argumentos, *arglist* si no se especifico ningún a_i, o a la lista de $a_1 \ldots a_k$ añadida a *arglist*.

`(eval` *e*`)`
Evalúa *e* por ser su argumento y a continuación evalúa el resultado de dicha evaluación.

`(funcall` *function* $a_1 \ldots a_k$`)`
Aplica el valor de *function* a los argumentos $a_1 \ldots a_k$. Se da la siguiente equivalencia:

`(apply #'`*fn* `(list` $arg_1 \ldots$ arg_n`))` $\equiv$ `(funcall #'`*fn* $arg_1 \ldots$ arg_n`)`

`(quote` *e*`)`
Simplemente retorna su argumentos sin evaluar. Esto permite que cualquier objeto Lisp se pueda escribir como un valor constante en un programa. Como se usa con frecuencia se puede abreviar como `'`, por tanto, `'x` es lo mismo que `(quote x)`.

15 Miscelánea

`(load` *pathname*`)`
Carga el archivo *pathname* en el interprete.

`(trace` $function_1 \ldots function_k$`)`
Habilita el trazado de las funciones que aparecen como argumentos (los nombres de las funciones no es necesario que vayan precedidos por el quote).

`(untrace` $function_1 \ldots function_k$`)`
Deshabilita el trazado de las funciones que aparecen como argumentos (los nombres de las funciones no es necesario que vayan precedidos por el quote). Si se invoca sin argumentos deshabilita el trazado de todas las funciones.

`(time` *e*`)`
Evalúa *e* pero además devuelve información del tiempo de ejecución.

`(dribble [`*pathname*`])`
Imprime las salidas y entradas en el fichero *pathname* permitiendo crear un registro de una sesión interactiva. Si se llama sin argumentos finaliza la creación del registro y cierra el fichero correspondiente.

`(documentation` *sbl tipo-doc*`)`
Muestra la cadena de documentación asociada al símbolo *sbl* si no existe dicha cadena devuelve `NIL`. El valor de *tipo-doc* debe uno de los siguientes ser: `function`, `variable` o `type`.

`(lisp-implementation-version)`
Devuelve una cadena que identifica la versión de la implementación de Common Lisp que se este usando.

`(sleep` *n*`)`
Causa que la ejecución se suspenda durante aproximadamente *n* segundos. Devuelve `NIL`.

`(values [`$e_1 \ldots e_k$`])`
devuelve los k valores resultado de evaluar los k argumentos. La sentencia values al final de una función es la forma estándar de lograr que una función devuelva más de un valor. Sin argumentos no devuelve ningún valor.

`(multiple-value-setq ([`$var_1 \ldots var_k$`]) `*forma-val*`)`
Se asignan (no se ligan) sucesivamente los valores devueltos por la evaluación de *forma-val* a las variables $var_1 \ldots var_k$ incluidas en la lista de variables. Si hubiera más variables que valores, a estas se les asigna nil. Si hay mas valores que variables, los valores en exceso no se pierden.

`(multiple-value-bind ([`$var_1 \ldots var_k$`]) `*forma-val*` `$e_1 \ldots e_n$`)`
Funciona de forma análoga al let. Los valores devueltos por *forma-val* se asignan localmente a las variables incluidas en la lista de variables y luego se ejecutan sucesivamente el resto de las formas $e_1 \ldots e_n$.

16 Tipos de argumentos en una definición funcional

El formato general de definición funcional es el siguiente:

`(defun` *nombre lambda-lista* {*declaración*}* {*forma*}*`)`

Que asocia al símbolo *nombre* la función cuyo cuerpo es {*forma*}* y cuya lista de parámetros viene dada por *lambda-lista*. La sintaxis de una *lambda-lista* es:

```
({var}*
        [&optional {var|(var [initform [svar]])}*]
        [&rest var]
        [&key {var|({var|(keyword var)}[initform [svar]])}*]
        [&aux {var|(var [initform])}*])
```

Para llamar a una función hay que aplicar el nombre sobre sus argumentos (evaluación). La evaluación pasa por las siguientes etapas:

1. evaluar los argumentos
2. asignar a los parámetros de la lambda-lista los valores de los argumentos
3. evaluar, secuencialmente, las formas del cuerpo
4. devolver la última forma evaluada

Una lambda-lista tiene cuatro partes:

1. parámetros requeridos: todos los que están antes de la primera clave
2. parámetros opcionales: entre `&optional` y la siguiente clave
3. parámetro rest: un único parámetro `&rest`
4. parámetros clave: entre `&key` y la siguiente clave
5. variables auxiliares: a partir de `&aux`

El proceso de aplicación de los argumentos es el siguiente:

1. Si hay n parámetros requeridos, hay que suministrar al menos n argumentos. Los parámetros quedan ligados al valor de los n primeros argumentos.
2. Si se ha incluido `&optional` y hay argumentos sin procesar, los parámetros opcionales quedan ligados a los siguientes argumentos, si no hay argumentos por procesar, los parámetros opcionales se ligan a `NIL` o al valor de *initform*.
3. Si se ha incluido `&rest`, el parámetro *rest* queda ligado a la lista formada con todos los argumentos que quedan por procesar.
4. Si se ha incluido `&key` las variables se ligan al valor que siga a la clave en la lista de argumentos (que no sean requeridos u opcionales), si no aparece se ligan a `NIL` o al valor de *initform*.
5. Si se ha incluido `&aux` los siguientes elementos de la *lambda-lista* no son en realidad parámetros, sino variables auxiliares que quedan localmente ligadas en la función.