Paradigmas y Lenguajes

Funcional Parte 2

Msc. Ricardo Monzón

FUNCIONES PARA OPERAR SOBRE LISTAS (1)

EXPRESIONES SIMBOLICAS Y FORMAS

De acuerdo a lo visto hasta el momento, los átomos y listas en Lisp son conocidos como **expresiones simbólicas**. Cualquier átomo válido o lista es una expresión simbólica. Por ejemplo:

ATOMOS: **5, A, "A ES UN STRING"**

LISTAS: **(UNA LISTA DE ELEMENTOS)**
(((A)))
(+ 3 (* X (/ 100 (- Y Z))))
(DEFUN ADD2 (X) (+ X 2))

Una forma en Lisp es un objeto que puede ser evaluado. A un átomo no se le considera como una forma tipo lista.

FUNCIONES PARA OPERAR SOBRE LISTAS (2)

Las listas son conjuntos ordenados de átomos o listas conectados.

Para realizar la manipulación de los elementos de una lista es importante tener acceso a sus elementos.

Existen tres tipos importantes de funciones de selección de elementos:

- **CAR -** devuelve el primer elemento de la lista.
- **CDR** - devuelve toda la lista menos el primer elemento.
- **C****R** - permite la concatenación de funciones CAR Y CDR. Es decir, CADR, CDDR, CADADR, etc.

FUNCIONES PARA OPERAR SOBRE LISTAS (3)

• La función **CAR**, devuelve el primer elemento de una lista.
Su sintaxis es **(CAR LISTA)**, o de una manera más técnica sería,

(CAR CONS).

Por ejemplo podríamos tener la siguiente llamada

(CAR (1 2 3))

Sin embargo, esto no funciona, ya que Lisp intentará evaluar la lista (1 2 3) y dará un ERROR. Intentará buscar una función definida con el nombre **1** y cuyos argumentos tomen valores **2** y **3** respectivamente. Como ya habíamos comentado y para evitar que LISP evalúe una lista de datos (y en general cualquier objeto) utilizaremos quote **'**. Así para el ejemplo anterior tendremos

> (CAR '(1 2 3))
> 1

FUNCIONES PARA OPERAR SOBRE LISTAS (4)

Ejemplos de CAR

> (CAR '(A B C))	**==>**	**A**
> (CAR '(A B (C D) E))	**==>**	**A**
> (CAR '((A B) C))	**==>**	**(A B)**
> (CAR '((A B C)))	**==>**	**(A B C)**
> (CAR 'A)	**==>**	**ERROR**

No se puede obtener el CAR de un ATOMO. En general no se puede aplicar ninguna función de LISTA a los ATOMOS.

Es importante destacar que la función CAR no genera un nuevo valor, sino que da como resultado la dirección de un elemento que existe.

FUNCIONES PARA OPERAR SOBRE LISTAS (5)

- La función **CDR**, para listas propias, de verdad o no punteadas, **devuelve la lista sin el primer elemento**.

La sintaxis de esta función es **(CDR LISTA)** o de forma técnica

(CDR CONS)

> (CDR '(A B C)) **==>** **(B C)**
> (CDR '(A B (C D) E)) **==>** **(B (C D) E)**
> (CDR '((A B) C)) **==>** **(C)**
> (CDR '((A B C))) **==>** **NIL**
> (CDR 'A) **==>** **ERROR**

Los argumentos del CDR deben ser del tipo CONS.

Para una lista punteada, el CDR de una estructura CONS puede ser un átomo. Por tanto, en estos casos se devolverá un átomo y no una estructura CONS.

> (CDR '(A . B)) **==>** **B**

FUNCIONES PARA OPERAR SOBRE LISTAS (6)

• Las funciones **C****R**, no son más que una abreviación de las formas CAR y CDR. Podemos invocar a las llamadas (C*R LISTA), (C**R LISTA), (C***R LISTA), (C****R LISTA), donde el * podrá sustituirse por A, para invocar a CAR, o por D para invocar a CDR. Solo pueden sustituirse cuatro posiciones en la misma función.

Los caracteres incluidos invocan a las funciones respectivas en orden de derecha a izquierda.

> (CADR '(A B C))
(CAR (CDR '(A B C))) ==> B
> (CDAR '((A B C) D E))
(CDR (CAR '(A B C))) ==> (B C)
> (CADDR '(A B C D E)) ==> C
> (CAR (CDDDDR '(A B C D E))) ==> E

FUNCIONES PARA OPERAR SOBRE LISTAS (7)

- **LAST** LIST. Esta función devolverá la última estructura CONS de la lista. **Si hay un solo elemento, se toma como ultimo**. Por ejemplo,

```
> (LAST '(A B C))        ==>    (C)
> (LAST '(A (B C)))      ==>    ((B C))
> (LAST '(1 2 3.5))      ==>    (3.5)
> (LAST '((A B C))       ==>    ((A B C))
```

- **ELT** LISTA INDICE. Esta función devolverá el elemento de la lista que está en la posición **índice**. El primer elemento de la secuencia tiene el índice en la posición 0. Ejemplos,

```
> (ELT '(A (B C)) 1)          ==>    (B C)
> (ELT '(A B) 3)              ==>    ERROR
> (ELT '(A B C D E) 2)        ==>    C
```

En el ultimo ejemplo, las posiciones de los elementos son 0,1,2,3,4. El 2 representa la tercera posición.

FUNCION DE ASIGNACION (1)

- La función **SETQ** asigna un valor a una variable. Su sintaxis es:

(SETQ {VAR FORM}+).

La Q de SETQ es un mnemotécnico de la función quote. Las llaves y el signo mas, representan repetición del par VAR FORM.

(SETQ VAR FORM VAR FORM VAR FORM …)

DEBE haber un número par de argumentos. Los argumentos impares de la función no son evaluados.

Ej. > (setq x 4 y 6 z 9)
> 9

Esto asigna 4 a x, 6 a y, 9 a z. Siempre las evaluaciones de expresiones, devuelven el resultado de la ultima evaluación. En este caso, la ultima evaluación fue z 9.

FUNCION DE ASIGNACION (2)

Las evaluaciones de las formas y las asignaciones (primero se evalúa y luego se asigna), se realizan secuencialmente.

Ejemplos,

```
> (SETQ X 5)                                  ==>   5
> (SETQ NOMBRE "MARIA")                       ==>   "MARIA"
> (SETQ FRUTA (CAR ´(MANZANA PERA )))         ==>   MANZANA
> (SETQ LISTA1 ‘(A B C))                      ==>   (A B C)
> (SETQ X 5 Y 6 Z 7)                          ==>   7
> (SETQ X 5 Y 6 Z)                  Error: too few arguments
```

En el último ejemplo, si el número de argumentos es impar, devuelve un error.

FUNCIONES DE CONSTRUCCION DE LISTAS (1)

Las funciones con las que Lisp permite construir listas son

- **CONS**
- **LIST**
- **APPEND**.

Con ellas deberemos ser capaces de construir listas de elementos, crear estructuras CONS, añadir más elementos a una lista, concatenar listas, quitar elementos de una lista, etc.

FUNCIONES DE CONSTRUCCION DE LISTAS (2)

- **CONS** SEXP SEXP

La función **CONS**, crea una nueva estructura cons. Sintácticamente podemos expresarlos como:

(CONS SEXP SEXP)

donde SEXP son expresiones simbólicas, pueden ser átomos o listas.

El CAR de la nueva lista será la primera **SEXP** y el CDR la segunda.

> (CONS 'A '(B C D))	**==>**	**(A B C D)**
> (CONS '(X Y) '(B C D))	**==>**	**((X Y) B C D)**
> (CONS '((X Y) (W Z)) '((A B)))	**==>**	**(((X Y) (W Z)) (A B))**
> (CONS 'A NIL)	**==>**	**(A)**
> (CONS NIL '(A))	**==>**	**(NIL A)**

FUNCIONES DE CONSTRUCCION DE LISTAS (3)

Obsérvese que si la segunda SEXP corresponde con un átomo entonces se creará una lista punteada ("impropia").

> (CONS 'A 'B) ==> (A . B)
> (CONS '(A B) 'C) ==> ((A B) . C)

La función CONS puede utilizarse para también para realizar las siguientes operaciones:

Función CONS	*Ejemplo*	*Resultado*	*Sirve para...*
(CONS átomo lista)	(CONS 1 '(2 3 4))	(1 2 3 4)	Añadir un elemento a una lista
(CONS átomo átomo)	(CONS 1 2)	(1 . 2)	Crear una lista impropia
(CONS lista lista)	(CONS (1 2) (3 4))	((1 2) 3 4)	Añadir una sublista a una lista
(CONS lista átomo)	(CONS '(1 2) 4)	((1 2) . 4)	Añadir sublista y crear lista impropia

FUNCIONES DE CONSTRUCCION DE LISTAS (4)

- **LIST** &REST ARGS

La función **LIST** devuelve una lista formada con todos los elementos pasados como argumentos. Los argumentos deberán ser expresiones simbólicas válidas.

La notación &REST ARG implica que se permite cualquier número de argumentos. Es decir, se le puede pasar a la llamada de la función cualquier número de parámetros.

La sintaxis corresponderá con llamadas del estilo

(LIST SEXP SEXP ...)

LIST se utiliza para crear listas propias. Las listas impropias sólo pueden crearse a partir de la función CONS.

> (LIST 'A 'B 'C) ==> (A B C)
> (LIST ‘(A) ‘(B) ‘(C)) ==> ((A) (B) (C))
> (LIST ‘A ‘(B C)) ==> (A (B C))

FUNCIONES DE CONSTRUCCION DE LISTAS (5)

- **APPEND** &REST ARGS

Permite crear una nueva lista concatenando dos o más listas dadas. Mientras que LIST y CONS aceptan listas y átomos como argumentos, **APPEND sólo acepta listas, excepto en el último argumento**. También acepta un número indefinido de argumentos.

La función **APPEND** concatena los argumentos en una lista. Todos los argumentos, excepto el último deben ser listas. Los argumentos que no sean listas no serán incluidos.

Sintácticamente lo indicaríamos como

(APPEND LISTA LISTA … LISTA SEXP)

FUNCIONES DE CONSTRUCCION DE LISTAS (6)

- **APPEND** &REST ARGS

> (APPEND ‘(A) ‘(B) ‘(C)) **==>** **(A B C)**
> (APPEND ‘(A) ‘(B C)) **==>** **(A B C)**
> (APPEND ‘((A)) ‘((B)) ‘((C))) **==>** **((A) (B) (C))**
> (APPEND ‘(A) NIL) **==>** **(A)**
> (APPEND NIL ‘(B C)) **==>** **(B C)**

NIL es una lista y por tanto cumple los requerimientos de la definición de APPEND.

> (APPEND ‘A ‘(B C)) **==>** **ERROR**

> (APPEND ‘(A) ‘(B) ‘C) **==>** **(A B . C)**

FUNCIONES DE CONSTRUCCION DE LISTAS (7)

Otras funciones de listas

(BUTLAST lista n), devuelve una nueva lista con los n últimos elementos de la lista pasada como parámetro eliminados. No se modifica el contenido de la lista original

Ejemplos,

> (setq b '(3 4 5 6))
> **(butlast b 2)**
> **(3 4)**
> **b**
> **(3 4 5 6)**

Si quisiéramos que el resultado se asigne a una nueva lista, haríamos:

(setq lista (butlast lista n))

FUNCIONES DE CONSTRUCCION DE LISTAS (8)

- **NTH <n> <list>**

Permite extraer el elemento “n” de una lista “list”. El primer elemento es n=0.

Ejemplos,

> (setq b ‘(4 5 6))
> (nth 1 b)
> 5
> (nth 0 b)
> 4

- **NTHCDR <n> <list>**

Permite extraer el cdr de una lista “list” a partir del elemento “n”.

Ejemplos,

> (setq b ‘(4 5 6 7 8))
> (nthcdr 3 b)
> (7 8)
> (nthcdr 1 b)
> (5 6 7 8)

FUNCIONES DE CONSTRUCCION DE LISTAS (9)

- **REVERSE <list>**

  Permite obtener el reverso de una lista. No modifica el contenido de la lista original.

Ejemplos,

```
> (setq b '(4 5 6 7 8))
> (reverse b)
> (8 7 6 5 4)
> b
> (4 5 6 7 8)
```

Si quisiéramos guardar la lista resultado debemos hacer:

```
> (setq lista1 (reverse b))
```

FUNCIONES DESTRUCTIVAS (1)

A diferencia de las funciones vistas anteriormente, que son llamadas NO DESTRUCTUIVAS, las funciones **destructivas** son **aquellas que modifican el contenido de algún parámetro**.

- **(RPLACA lista elem)**, sustituye el car de la lista con elem.

Ejemplos:

> (setq a '(1 2 3))
> (1 2 3)
> (rplaca a 7)
> (7 2 3)
> a
> (7 2 3)

FUNCIONES DESTRUCTIVAS (2)

- **(RPLACD lista elem),** sustituye el **cdr** de la lista con elem.

Ejemplos:

```
> (setq a ‘(7 2 3))
> (7 2 3)
> (rplacd a '(9))
> (7 9)
> a
> (7 9)
> (rplacd a 6)
> (7.6)
> a
> (7.6)
```

FUNCIONES DESTRUCTIVAS (3)

- **(NCONC lista1 ... listan),** a diferencia de APPEND, NCONC devuelve su valor modificando el símbolo que expresa **lista1, y modifica el valor de las sucesivas listas hasta n-1.**

Ejemplos:

```
 > (setq a '(1 2) b '(3 4) c '(5 6) d '(7 8))
(7 8)
> (nconc a b c d)
(1 2 3 4 5 6 7 8)
> a
(1 2 3 4 5 6 7 8)
> b
(3 4 5 6 7 8)
> c
(5 6 7 8)
> d
(7 8)
>
```

FUNCIONES DESTRUCTIVAS (4)

- **(PUSH item lista),** añade el elemento item a **lista al principio.**

  (setq lista (cons item lista)) ,función equivalente

```
> (setq b '(4 5 6))
(4 5 6)
> (push 9 b)
(9 4 5 6)
> b
(9 4 5 6)
```

- **(POP lista),** elimina el primer elemento de **lista**

  (setq lista (cdr lista)) ,función equivalente

```
> (pop b)
9
> b
(4 5 6)
```

OTRAS FUNCIONES (1)

- **LENGTH** SECUENCIA. Devuelve el número de elementos existentes en el nivel superior de la lista (o secuencia).

Por ejemplo,

> (LENGTH '(A B C)) **==>** **3**

> (LENGTH '(A B . C)) **==>** **2**

(en este caso devuelve 2, porque el segundo elemento es uno solo llamado "Par punteado").

> (LENGTH '()) **==>** **0**

OTRAS FUNCIONES (2)

- **MEMBER** ITEM LISTA {&KEY {:TEST / :TEST-NOT / :KEY}}

La función MEMBER busca en el nivel superior de la LISTA un elemento que sea igual, **EQL**, que el ITEM. Si se encuentra un elemento que cumple esta condición, la función devolverá una lista cuyo CAR es el elemento buscado y el CDR es el resto de la lista.

Por el contrario si no se encuentra ningún elemento se devolverá NIL. Por defecto, el test que se realiza es EQL. Sin embargo, se puede expresar otro tipo de comparación en la opción KEY de la lista de parámetros.

La opción :TEST permite realizar comparaciones sucesivas del ITEM con cada uno de los elementos de la lista a través del operador señalado después de :TEST, hasta encontrar el primer elemento que cumple dicha condición, es decir, que de NO-NIL.

OTRAS FUNCIONES (3)

La opción :TEST-NOT es semejante a la anterior, salvo que se detiene al encontrar el primer elemento en cuya evaluación se obtenga NIL. Finalmente la opción :KEY hace lo mismo que las anteriores pero aplica la función indicada en clave.
Veamos algunos ejemplos:

> (MEMBER 'C '(A B C D E F)) **==> (C D E F)**
> (MEMBER 'Z '(A B C D E F)) **==> NIL**
> (MEMBER 'J '(A B (I J) F)) **==> NIL**
> (MEMBER '(X Y) '(A B C (X Y) D)) **==> NIL**

; Su valor es NIL por que compara con EQL

> (MEMBER '(X Y) '(A B C (X Y) D) :TEST #'EQUAL) **==> ((X Y) D)**
> (MEMBER '7 '(3 5 7 9) :TEST-NOT #'>) **==> (7 9)**

OTRAS FUNCIONES (4)

FMAKUNBOUND <sym>

Permite desligar un símbolo de una función.

>> **(fmakunbound mi-suma)** ; desliga la función mi-suma
>> **mi-suma** **==>** Error: The variable MI-SUMA is unbound

MAKUNBOUND <sym>

Permite desligar un símbolo de su valor.

>> **(setq a 4)** ; liga el símbolo a con 4
>> A **==>** 4
>> **(makunbound 'a)** ; desliga el símbolo a
>> A **==>** Error: The variable A is unbound.

OTROS PREDICADOS (5)

BOUNDP <sym>

El predicado boundp chequea si el símbolo sym, tiene un valor ligado a el. Retorna T si tiene un valor ligado, o NIL en caso contrario.

> **(setq a 1)** ; liga la variable a con el valor 1
> **(boundp 'a)** → **T** ; retorna T – el valor es 1

FBOUNDP <sym>

El predicado fboundp chequea si el símbolo sym, tiene una definición de función ligada a el. Retorna T si tiene un valor ligado, o NIL en caso contrario.

> (defun f1 (x) (print x)) ; set up function F1
> (fboundp 'f1) → T
> (fboundp 'car) → T
> (fboundp 'f2) → Nil

PREDICADOS Y OPERADORES LOGICOS P/LISTAS (1)

Predicados sobre tipos de datos.

Todos los predicados que se muestran a continuación devuelven T si son ciertos o NIL sino lo son.

ATOM OBJETO. Devuelve true si el objeto NO es una construcción CONS.

CONSP OBJETO. Devuelve cierto si el objeto es CONS.

LISTP OBJETO. Devuelve cierto si el objeto es CONS o NIL (la lista vacía).

NULL OBJETO. Será cierto si objeto es NIL.

TYPEP OBJETO TIPO-ESPECIFICADO. Es cierto cuando el objeto pertenece al tipo definido en tipo especificado.

Predicados de igualdad.

Los objetos Lisp pueden verificarse a diferentes niveles de igualdad:

- Igualdad numérica **=**
- Identidad referencial **EQ.**
- Identidad representacional **EQL**. (acepta solo dos argumentos)
- Igualdad estructural **EQUAL** (acepta solo dos argumentos)
- Igualdad de valores **EQUALP**.

PREDICADOS DE IGUALDAD (1)

EQ X Y.

La función EQ testea si X e Y están referenciados por punteros iguales.

Solo se utiliza para caracteres, símbolos y enteros pequeños menores o iguales a 255.

Sintácticamente tendríamos **(EQL SEXPR SEXPR)**, donde la sexpr se deberá evaluar como un número o variable.

> **(eq ‘a ‘a)**	→	T
> **(eq 1 1)**	→	T
> (eq 256 256)	→	Nil (solo hasta 255)
> (eq 1 1.0)	→	Nil
> (eq ‘’a’’ ‘’a’’)	→	Nil

PREDICADOS DE IGUALDAD (2)

EQL X Y.

La función EQL testea si X e Y representan el mismo valor. Esta función puede devolver false aunque el valor imprimible de los objetos sea igual. Esto sucede cuando los objetos apuntados por variables parecen los mismos pero tienen diferentes posiciones en memoria.

Este predicado no conviene utilizarlo con strings, ya que su respuesta no será fiable. Por el contrario se deberá utilizar con variables y números cuando sea preciso conocer si tienen la misma posición en memoria.

Sintácticamente tendríamos **(EQL SEXPR SEXPR)**, donde la sexpr se deberá evaluar como un número o variable.

PREDICADOS DE IGUALDAD (3)

EQL X Y

Ejemplos

> (EQL 5 5) **==> T**

> (EQL 5 5.0) **==> NIL**

> (SETQ A ´WORD)

> (SETQ B ´WORD)

> (EQL A B) **==> NIL**

> (SETQ L ´(A B C))

> (EQL ´(A B C) L) **==> NIL (NO FIABLE)**

> (SETQ M L)

- **(EQL L M)** **==> T** (dos variables apuntan al mismo sitio)

> (SETQ N ´(A B C))

> (EQL L N) **==> NIL**

> (EQUAL L A) **==> NIL**

PREDICADOS DE IGUALDAD (4)

EQUAL X Y.

Verifica si los objetos X e Y, son estructuralmente iguales. Es decir, los valores imprimibles de los objetos son iguales. Sintácticamente tendríamos que expresarlo (EQUAL SEXPR SEXPR).

Ejemplos

```
> (EQUAL 5.0 5)                    ==> NIL
> (EQUAL NIL ())                   ==> T
> (SETQ A ´WORD  B ´WORD)
> (EQUAL A B)                      ==> T
> (SETQ L ´(A B C))
> (EQUAL ´(A B C) L)               ==> T
> (SETQ M L)
> (EQUAL L M)                      ==> T
> (SETQ N ´(A B C))
> (EQUAL L N)                      ==> T
> (EQUAL L A)                      ==> NIL
```

PREDICADOS DE IGUALDAD (5)

EQUALP X Y

La función EQUALP testea si X e Y representan el mismo valor, pero a diferencia de EQUAL, esta función se denomina Case Insensitive. A diferencia de Eql, puede aplicarse a los string.

```
> (EQUALP 5 5)                ==> T
> (EQUALP 5 5.0)              ==> T
> (EQUALP  ‘‘a’’ ‘‘A’’)       ==> T
```

PREDICADOS DE IGUALDAD (6)

Resumen:

- **Eq** – Punteros idénticos. Palabras con caracteres, símbolos y pequeños enteros.
- **Eql** – Solo números del mismo tipo.
- **Equal** – Listas y Strings.
- **Equalp** – Caracteres y strings con identificación de mayúsculas, números de diferentes tipos, arreglos.

PREDICADOS DE IGUALDAD (7)

	funcion fn			
Argumentos	**eq**	**eql**	**equal**	**equalp**
(fn 'a 'a)	T	T	T	T
(fn 1 1)	T	T	T	T
(fn 1 1.0)	NIL	NIL	NIL	T
(fn 1.0 1.0)	NIL	T	T	T
(fn "a" "a")	NIL	NIL	T	T
(fn '(a b) '(a b))	NIL	NIL	T	T
(setq a '(a b))				
(setq b a)				
(setq c '(a b))				
(fn a b)	T	T	T	T
(fn a c)	NIL	NIL	T	T
(fn '(a b) '(A B))	NIL	NIL	T	T
(fn '(a b) '(c d))	NIL	NIL	NIL	NIL
(fn "a" "A")	NIL	NIL	NIL	T
(fn "abc" "abcD")	NIL	NIL	NIL	NIL

OPERADORES LOGICOS (1)

Lisp tiene tres de los operadores lógicos más comunes como primitivas, **AND, OR y NOT**.

Los demás pueden crearse a partir de estos. Los funciones AND y OR devuelven un valor cuando las condiciones son ciertas, en otro caso devuelven nil.

Estas funciones no evalúan necesariamente todos los argumentos. Una vez que se verifica la condición se devuelve NO-NIL o T.

Por ello es importante el orden en el que se colocan los argumentos.

OPERADORES LOGICOS (2)

AND {FORM}*

La función AND evalúa sus argumentos en orden. Si cualquiera de los argumentos se evalúa a NIL, se detiene la evaluación y se devuelve el valor NIL. Por el contrario si todos los argumentos son NO-NIL, devolverá el resultado de la última evaluación. Sintácticamente **(AND SEXPR SEXPR …)**, está permitido cualquier número de argumentos.

Ejemplos,

```
> (AND T  (< 2 5)  (> 7 4)  (* 2 5))                ==> 10
> (SETQ X 3 CONT 0)
> (INCF CONT)
> (AND (<= CONT 10) (NUMBERP X) (* 2 X))            ==> 6
> (AND (EVENP X) (/ X 2))                           ==> NIL
```

OPERADORES LOGICOS (3)

OR {FORM}*

La función OR evalúa sus argumentos en orden. Si cualquiera de los argumentos se evalúa a NO-NIL, se detiene la evaluación y se devuelve T. Por el contrario si todos los argumentos son NIL, la función OR devolverá el resultado de la ultima sexp.

Sintácticamente (OR SEXPR SEXPR …), está permitido cualquier número de argumentos.

Ejemplos,

> (OR NIL (= 4 5) (NULL ´(A B)) (REM 23 13)) **==> 10**

> (SETQ X 10)

> (OR (< 0 X) (DECF X)) **==> T**

> (OR (> 0 X) (DECF X)) **==> 9**

> (OR (CONSP X) (EVENP X) (/ X 2)) **==> 5**

Preguntamos si X es una lista, luego si X es Par, ambas dan NIL.

OPERADORES LOGICOS (4)

NOT FORM

La función NOT evalúa un único argumento. La función NOT devuelve T o NIL. Si los argumentos se evalúan a NIL, entonces devuelve T, en otro caso NIL.

Sintácticamente (NOT SEXPR), está permitido sólo un argumento.

Ejemplos:

```
> (NOT NIL)                    ==> T
> (NOT T)                      ==> NIL
> (NOT (EQUAL ´A ´A) )         ==> NIL
> (SETQ X ´(A B C) )
> (NOT (ATOM X))               ==> T
```

SECUENCIA DE ACCIONES (1)

Se llama ***bloque*** a una secuencia de sentencias que deben ser ejecutadas secuencialmente en el orden en el que aparecen.

LISP proporciona diferentes formas de crear estos bloques.

Algunos de ellos se crean de forma implícita, como veremos mas adelante (loop y return).

LISP también proporciona mecanismos para crear bloques de forma explícita.

Para ello utilizaremos los bloques creados por la familia **prog**.

SECUENCIA DE ACCIONES (2)

progn

permite crear un bloque de sentencias que serán evaluadas secuencialmente. Se toma el valor devuelto por la última de ellas como valor de retorno de la forma progn completa.

(progn [< sentencia 1 > ... < sentencia n>])

Donde hay que observar que se podría crear un bloque sin sentencias.

Proporciona una forma de agrupar una serie de expresiones LISP. Cada una corresponde con una forma determinada y serán evaluadas en orden.

La sintaxis completa es:

PROGN {((VAR {INIT})/ VAR*)} DECLARACIONES CUERPO).

SECUENCIA DE ACCIONES (3)

El valor devuelto por PROGN corresponde al de la última forma evaluada.

> (progn (setq a 23) (setq b 35) (setq c (* a b))) → 805

PROG1 y PROG2, devuelven el valor de la primera y segunda forma respectivamente, mientras se evalúan el resto de las formas.

> (prog1 (setq a 23) (setq b 35) (setq c (* a b))) → 23

> (prog2 (setq a 23) (setq b 35) (setq c (* a b))) → 35

> c → 805

SECUENCIA DE ACCIONES (4)

Si en la declaración de un PROGN se incluyen variables locales, la asignación de sus valores se hace en paralelo, al igual que sucede con LET que veremos mas adelantet.
LISP da la posibilidad de que la asignación de valores a las variables se realice secuencialmente mediante PROG*
(PROG* ...)

La devolución de un valor para una forma puede hacerse explícito mediante RETURN.

(RETURN resultado)