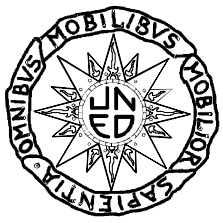
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Introducción al Lenguaje Common Lisp

JESÚS GONZÁLEZ BOTICARIO

DEPARTAMENTO DE INTELIGENCIA ARTIFICIAL



Dpto. de Inteligencia Artificial

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Introducción al Lenguaje Common Lisp

Jesús González Boticario

ÍNDICE

CURSO DE COMMON LISP

1. INTRODUCCIÓN

1.1 Objetivos del curso

Este material pretende proporcionar una guía teórica para fijar y ampliar los aspectos fundamentales del lenguaje LISP, cuyo contenido puede ser contrastado con el texto incluido en la documentación básica [1]. Dado que el único objetivo es resaltar los aspectos más relevantes del lenguaje, esta guía también intenta ser un esquema organizado de las principales sentencias del lenguaje, de tal forma que puede llegar a ser, sobre todo para alumnos principiantes, una guía de consulta útil a la hora de plantear la solución de problemas concretos.

No debe por tanto considerarse este material como una explicación detallada de todas las sentencias del lenguaje, más bien debe entenderse **como un curso de referencia**, al cual se puede acudir para recordar rápidamente las construcciones más significativas del lenguaje.

1.2 Notación

En la explicación de cada una de las primitivas del lenguaje se va a adoptar la siguiente notación

(nombre_primitiva *argumentos*) ⇐ los nombres de las primitivas son invariantes (la única forma de invocar una determinada función es escribir correctamente su nombre)

El objetivo de esta notación es facilitar la comprensión de las diferentes posibilidades de combinación de argumentos en las llamadas respectivas

arg_1 ⇐ los argumentos cuyo valor pueda ser variable se representan mediante letra en itálica, o también mediante <arg>

[arg_1] ⇐ el argumento que aparece entre corchetes es opcional

[arg_1]* ⇐ el argumento que aparece entre corchetes seguido del carácter (“*”) puede aparecer de 0 a N veces en la llamada a la función

{arg_1 | arg_2 | arg_3}* ⇐ los argumentos que aparecen entre llaves indican que hay que elegir una entre varias posibilidades, si además aparece el carácter asterisco entonces el significado de dichas llaves también puede repetirse hasta N veces

arg_1 arg_2 … arg_n ⇐ expresa que puede haber de 1 a N argumentos

1.3 Breve Reseña Histórica

- El LISP es el segundo de los lenguajes más antiguos, junto con el FORTRAN, que sigue utilizándose y ampliándose en la actualidad
- Diseñado para la manipulación de símbolos
- Evoluciona adaptándose a las necesidades
- Precursores:
  - IBM 704: los nombres de su arquitectura (p.ej., "content address register") se importan como primitivas "CAR"
  - IPL: contiene operaciones primitivas para manejar símbolos y listas
  - Cálculo LAMBDA: A. Church creó una notación funcional adoptada por McCarthy para el LISP
  - Funciones recursivas: LISP fue el primer lenguaje que permite su definición
- Años 50: Creado a mediados de estos años en el laboratorio de Inteligencia Artificial (IA) del MIT (Massachusetts Institute of Technology)
  - Su creador principal fue John McCarthy
  - Las investigaciones en el campo de la IA requerían técnicas efectivas de manipulación de símbolos
- Años 60: Es la época del LISP 1.5
  - Lenguaje poco evolucionado con unas 100 primitivas
- Años 70: Diversificación del LISP 1.5

- Dos proyectos alternativos

  El proyecto MAC del MIT fue la base del ZETALISP

  El BBNLISP terminó originando el INTERLISP

- Años 80: Época de divulgación y estandarización
  - Consolidación y estandarización del lenguaje: COMMON LISP [2]
  - Se incrementa notablemente su utilización en el desarrollo de aplicaciones reales
  - Ampliaciones: Facilidades para la creación de procesos paralelos, ventanas, “flavors”, etc)
- Actualmente:
  - Normalización: subcomité X3J13 de ANSI para producir un estándar de COMMON LISP [3]
  - Portabilidad: cualquier intérprete/compilador de COMMON LISP debe ser capaz de ejecutar el mismo código con leves modificaciones
  - Eficiencia: posibilidad de optimizar el código compilado
  - Potencia: una gran cantidad de funciones incorporadas
  - Programación orientada a objetos: CLOS (clases, mecanismos de herencia, métodos declarativos, protocolos, funciones genéricas, etc.)

1.4 Características básicas

- Uso de símbolos y listas como estructuras principales

- Funciones sencillas para componer funciones más complejas
- Lenguaje funcional: “no tiene sintaxis”
- No hay diferencia formal entre datos y programas:
  - Para un programa LISP otro programa LISP es un dato
  - Facilita la construcción de: depuradores, trazadores, formateadores de programas, editores de estructuras
- Repertorio de funciones básicas
- Flexibilidad
- Fácilmente modificable: numerosos dialectos
- Difícil de leer y de escribir sin herramientas (debido a que su sintaxis viene determinada exclusivamente por la utilización de listas de símbolos)
- Entorno de desarrollo/validación mediante intérprete
- Entorno de ejecución/aplicación mediante compilador
- Facilidades para la gestión de la memoria utilizada
  - Reserva dinámica de almacenamiento
  - Sistema automático de recuperación de memoria (llamado “Garbage Collection”)

1.5 Objetivos del COMMON LISP

- Creación de un lenguaje común

- Portabilidad
- Consistencia
- Expresividad
- Compatibilidad
- Eficiencia
- Potencia
- Estabilidad

1.6 Advertencias

- Debe considerarse que la filosofía de programación en LISP es muy diferente a otros lenguajes
- Olvidar algunas de las ideas clásicas de programación:
  - No es necesario declarar las variables
  - Uso de paréntesis como única sintaxis
  - Funciones con notación prefija
  - Composición de funciones
  - Evaluación inmediata utilizando el intérprete

Problema: Supongamos que se desea simular una conversación entre un psiquiatra (la máquina) y un supuesto paciente (el hombre). Para ello, una de las funciones que habrá que definir será la siguiente:

Función básica → comparar-frases

```
> (comparar-frases '(soy una frase) '(soy una
                                      frase))
```

```
T

> (comparar-frases '(soy una frase)
                   '(soy una persona))

NIL
```

Solución:

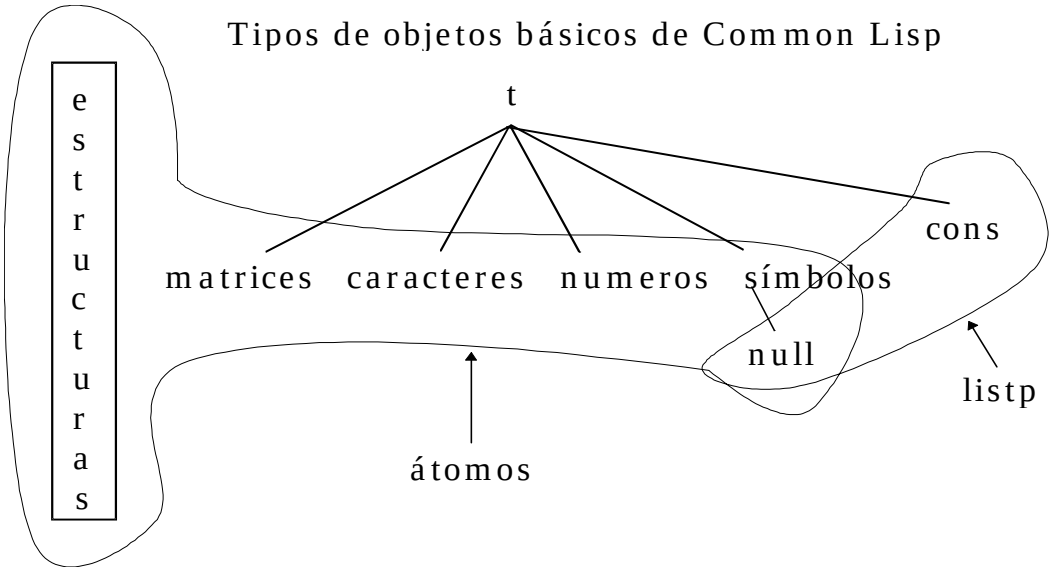
```
(defun comparar-frases (patron frase)
  (cond ((and (null patron) (null frase)) t)
        ((or (null patron) (null frase)) nil)
        ((equal (car patron) (car frase))
            (comparar (cdr patron) (cdr frase)))
        (t nil)))
```

Solución en otros lenguajes: representar palabras como matrices de caracteres, representar la frase como una matriz de caracteres, descomponer la entrada en palabras individuales explorando la matriz carácter a carácter, etc.

2. TIPOS DE DATOS

2.1 Organización jerárquica de objetos

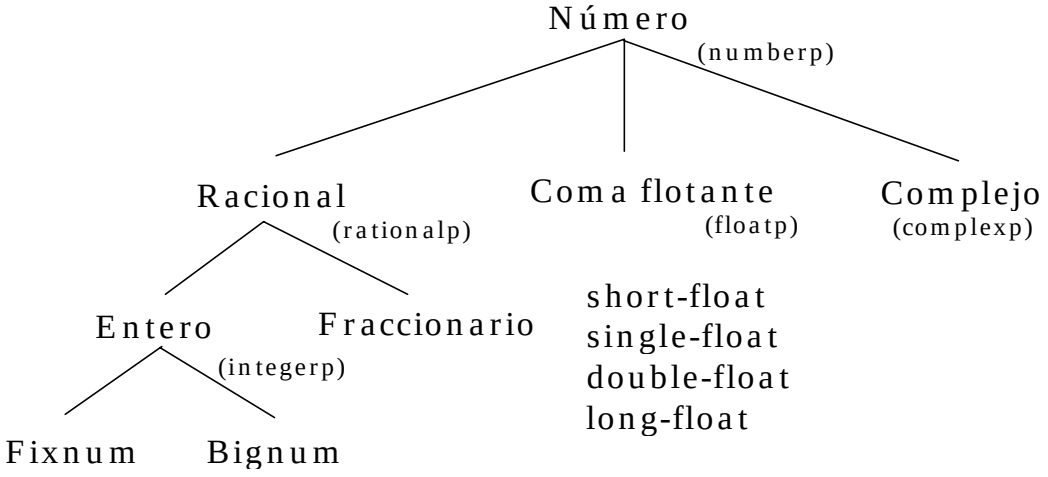
- Los tipos básicos de datos incluidos en el lenguaje se organizan formando una jerarquía de tipos que define una relación de pertenencia a un conjunto



2.2 Tipos básicos de datos

- **Números:** COMMON LISP facilita diferentes formas de representación para los números. Éstos se dividen en cuatro categorías:

- Enteros	+ 1024
- Fraccionarios	- 17/23
- Coma flotante	2.53
- Complejos	#C(5/3 7.0)



- **Símbolos:** Cada símbolo tiene asociado su nombre impreso

  Juan, JUAN, JuaN → JUAN

- **Listas:** Secuencia de elementos de cualquier tipo

  (a b c)

  (uno (dos tres) cuatro)

  (mi telefono es (2 59 43 75)

- **Cadenas de caracteres:** Secuencia de caracteres entre dobles comillas

  “Juan” → 4 caracteres

  “” → vacía

  “Vector con \” comillas“ → 21 caracteres

- **Vectores:** Matrices de elementos cualesquiera con una dimensión

#(a b c)

#(1 3 5 7 9)

- **Otros tipos:** Matrices, paquetes, flujos de entrada/salida, pathnames, etc.

3. Intérprete de LISP (read-eval-print)

- Durante la etapa de operación con el intérprete de LISP, continuamente se repite el siguiente bucle:

1. Lee una expresión → READ
2. Evalúa dicha expresión → EVAL
3. Devuelve el resultado → PRINT

>“Esto es un string”

“Esto es un string”

>(+ 534 789)

1323

> 5.389687

5.389687

> #\E

#\E

Funciones: Cada argumentos se evalúa por orden (de izquierda a derecha) y luego se aplica la función al resultado de la evaluación de los argumentos

> (car ´(primeros segundos terceros))

PRIMEROS

> (cons ‘los ‘(primeros segundos terceros))

(LOS PRIMEROS SEGUNDOS TERCEROS)

Excepciones: Algunas formas especiales (if let setq quote let* declare) no siguen la regla general de evaluación de funciones.

> (setq opciones ‘(mayor igual menor))

(MAYOR IGUAL MENOR)

> opciones

(MAYOR IGUAL MENOR)

Objetos: Precedidos de una comilla se evalúan a sí mismos

> ‘pepe

PEPE

```
> pepe
PEPE: unbound variable
```

Funciones con nombre: defun

```
> (defun mi-cuadrado (num)
        (if (numberp num) (* num num)))
MI-CUADRADO
> (mi-cuadrado 11)
121
```

Funciones sin nombre: lambda

```
> ((lambda (x)  (* x  x))  11)
121
```

4. Representación interna de objetos

4.1 Símbolos

- Objetos con varios componentes
- Cada componente puede accederse invocando una determinada función

Componente	**Función de acceso**
nombre impreso	symbol-name
valor	symbol-value
función	symbol-function
lista de propiedades	symbol-plist
paquete	symbol-package

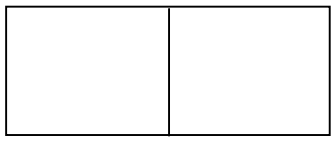
Advertencia: el nombre impreso identifica al objeto (la función que devuelve el nombre impreso se denomina “symbol-name”)

```
> (symbol-name ‘primer-simbolo)
“PRIMER-SIMBOLO”
```

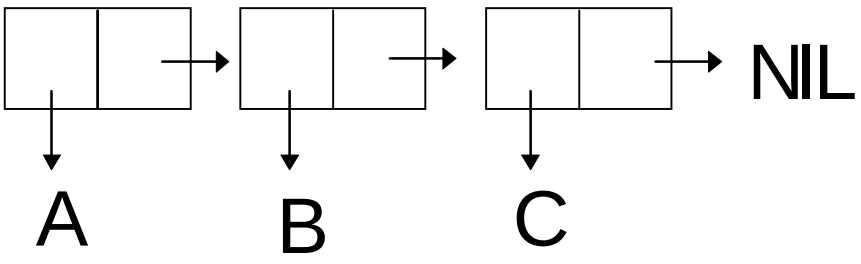
4.2 Listas

- LISP es un lenguaje diseñado para el tratamiento de listas
- En una lista se puede acceder fácilmente a los elementos de la misma
- Las listas se construyen internamente utilizando “celdas cons”.

Cada celda está formada por dos punteros:



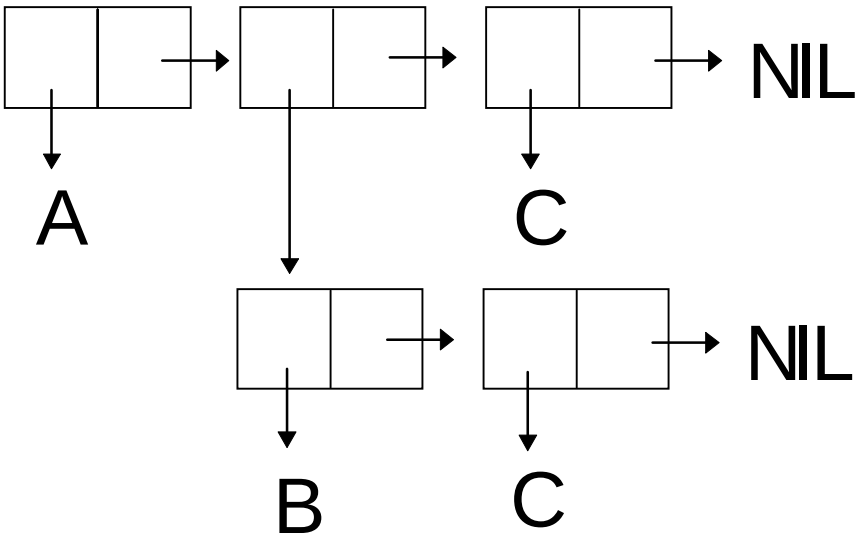
Internamente una lista cualquiera (A B C) tiene la siguiente representación:



NIL → Símbolo especial que indica el final de la lista

- Las listas pueden anidarse

Ejemplo: (A (B C) C)



CAR → nombre de la función que devuelve el primer elemento de la lista

CDR → nombre de la función que devuelve el resto de la lista

5. Variables

- Existen dos tipos de variables:

léxicas (estáticas) variables locales

- definidas por el contexto en el cual están escritas dentro del programa

especiales (dinámicas) variables globales

- definidas por el contexto en el cual se han creado durante la ejecución del programa

• Los símbolos **T** y **NIL** no pueden utilizarse como variables o tener valores asignados a ellos

T se interpreta como CIERTO

NIL se interpreta como FALSO (también representa la lista vacía)

Funciones de acceso para variables:

(setq *var1 form1 var2 form2* . . .)

forms son evaluadas y el resultado es almacenado en las variables *vars*

vars no son evaluadas

setq devuelve el último valor asignado

```
> (setq x “literal”  y ‘(1 2 3 4))
(1 2 3 4)
> (setq a ‘value)
value
```

(set *symbol value*)

modifica el valor de una variable especial

no puede modificar el valor de una variable local (asignación léxica)

devuelve *value*

• Existe una función para modificar el valor asignado a una variable, a un elemento de una lista, a un elemento de una matriz, etc.

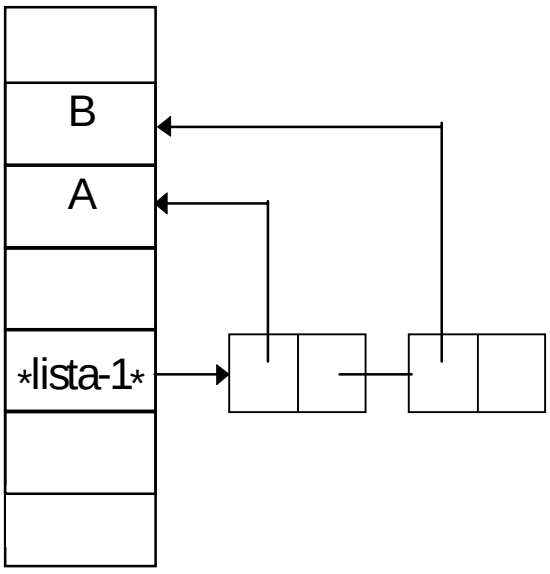
Función de actualización universal: setf

(setf *lugar valornuevo*)

- Evalúa *lugar* para acceder a un objeto
- Almacena el resultado de evaluar *valornuevo* en el lugar apuntado por la función de acceso *lugar*
- setf puede tener el mismo efecto que setq cuando *lugar* es el nombre de una variable
- setf va más allá de setq

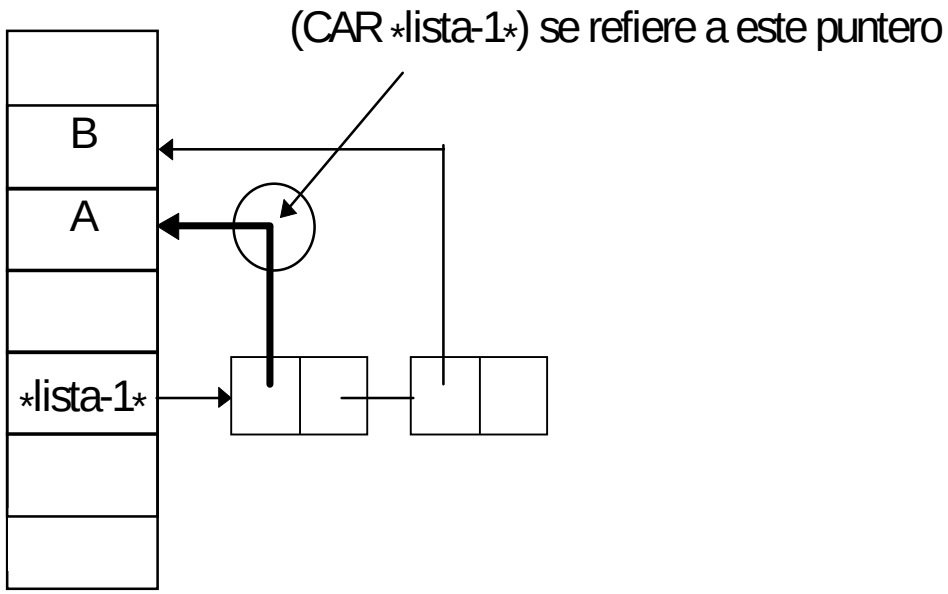
  puede modificar cualquier posición de memoria

```
> (setq *lista-1* '(A B))
(A B)
```




```
> (setf (car lista-1) 1)
1
```

• Las expresiones que devuelven la posición de un objeto LISP se denominan **"variables generalizadas"**



> lista-1

(1 B)

6. Funciones y predicados básicos

6.1 Operaciones con listas

(car *lista*) Devuelve el primer elemento de la lista

(car '(a b c)) → a

(car '()) → ()

(car '((1 2) 3 4) → (1 2)

(cdr *lista*) Devuelve el resto de los elementos de la lista (todos menos el primero)

(cdr '(a b c)) → (b c)

(cdr '()) → ()

(cdr '((1 2) 3 4)) → (3 4)

• Ambas evalúan sus argumentos sin modificarlos (aquellas funciones que modifican el valor de los argumentos recibidos se denominan **"funciones destructivas"**)

• Pueden invocarse combinaciones de ambas, juntándose en un mismo nombre de función hasta cuatro llamadas consecutivas

```
(setq oficina '(finanzas
                (parte-de gestion)
                (lugar madrid)
                (numero-empleados 350)
                (numeros-tf (2513398 2513399)))
```

(cadadr oficina) → gestion

(car (cdaddr oficina) → madrid

(cadr (cadddr oficina) → 350

(cadar (cddddr oficina) → (2513398 2513399)

(cons *x y*) Crea una "celda cons" cuyo car es *x* y cuyo cdr es *y*

(cons 'a (cons 'b (cons 'c '()))) → (a b c)

(car 'a '(b c d)) → (a b c d)

(car '((1 2) 3 4) → (1 2)

- cons generalmente se utiliza para añadir un nuevo elemento al principio de una lista

- evalúa sus argumentos sin modificarlos

(append *list1 list2 … listN*) Devuelve una lista como resultado de la concatenación de

todos sus argumentos (que deben ser listas)

(append ‘(a b c) ‘(d)) → (a b c d)

(append ‘(a b c) ‘(d e f) ‘() ‘(g)) → (a b c d e f g)

- evalúa sus argumentos
- La nueva lista está formada por una copia de los n - 1 primeros argumentos. El último se añade como cdr de la lista devuelta

(list *arg1 arg2 … argN*) Crea una lista a partir de sus argumentos

(list ‘a ‘b (+ 3 4) 8 ‘(d)) → (a b 7 8 (d))

- evalúa sus argumentos

Ejemplos:

(cons ‘(a b) ‘(c d)) → ((a b) c d)

(list ‘(a b) ‘(c d)) → ((a b) (c d))

(append ‘(a b) ‘(c d)) → (a b c d)

(cons ‘(a) ‘()) → ((a))

(list ‘(a) ‘()) → ((a) nil)

(append ‘(a) ‘()) → (a)

(cons ‘() ‘(a)) → (nil a)

(list ‘() ‘(a)) → (nil (a))

(append ‘() ‘(a)) → (a)

- Otras funciones de manipulación de listas:

(nth *n list*) Devuelve el *n*-ésimo elemento en *list*

(nth 3 ‘(1 2 3 4)) → 4

(nthcdr *n list*) Devuelve el resultado de *n* cdrs en *list*

(nth 3 ‘(1 2 3 4)) → (4)

(last *list*) Devuelve la última “celda cons”

(last ‘(1 2 (3 4))) → (3 4)

(length *list*) Devuelve el número de elementos en una secuencia (listas y vectores)

(length ‘(1 (2 3) 4)) → 3

(length “que longitud tengo”) → 18

(reverse *list*) Devuelve una copia de *list* con sus elementos en orden inverso

(reverse ‘(1 (2 3) 4)) → (4 (2 3) 1)

(butlast *list* &optional *n*) Devuelve una copia de *list* con todos sus elementos excepto los últimos *n* (ver más adelante “parámetros opcionales”). Por omisión *n* se considera que tiene el valor 1

(butlast ‘(1 (2 3) 4)) → (1 (2 3))

6.2 Operaciones con números

- Operaciones aritméticas: +, –, *, /

(*operador arg1 … argN*) Devuelve el resultado de la aplicación sucesiva del *operador* sobre los primeros *i* argumentos junto con la

aplicación del *operador* sobre cada nuevo argumento *i + 1*

(/ 3 4 5) → 3/20

(- 1 2 3 4 5) → -13

(*operador número*)

(1+ 5) → 6

equivale a (+ 1 5)

(1- 5) → 4

(incf *posición* [*valor*]) Aumenta en *valor* (el *valor* por omisión es 1) una variable generalizada. Equivale a la expresión:

(setf *posición* (+ *posición valor*))

(setq contador 4) → 4

(incf contador) → 5

(decf *posición* [*valor*]) Se mantiene lo indicado en incf, pero ahora la operación es "disminuir" en lugar de "aumentar".

- Operaciones trigonométricas: sin, cos, tan, atan, y las correspondientes hiperbólicas: sinh, cosh, tanh, atanh

Por ejemplo: (sin *radianes*) Devuelve el resultado del seno sobre un argumento que se supone expresado en radianes

```
> (sin pi)
1.22514845490862E-16
> (cos pi)
```

-1.0

6.3 Predicados

• Un predicado es una función que devuelve CIERTO o FALSO

NIL → Siempre es falso
T → cierto
Cualquier valor distinto de NIL se considera cierto

• **Predicados sobre tipos de datos:**

- Se utilizan para determinar si un objeto pertenece a un determinado tipo

(null *objeto*) T si el argumento es () "la lista vacía", si no devuelve NIL

(symbolp *objeto*) T si el argumento es un símbolo, si no devuelve NIL

(atom *objeto*) T si el *objeto* no es una "celda cons", si no devuelve NIL

(listp *objeto*) T si el *objeto* es una "celda cons" o la lista vacía, si no devuelve NIL

(consp *objeto*) T si el *objeto* es una "celda cons", si no devuelve NIL

(numberp *objeto*) T si el *objeto* es cualquier tipo de número, si no devuelve NIL

Otros predicados sobre tipos: integerp, floatp, characterp, stringp, vertorp, arrayp, functionp, etc.

- **Predicados numéricos:**
  - Sólo pueden aplicarse sobre argumentos que sean números

(zerop *número*) T si el argumento es cero, si no devuelve NIL

(plusp *número*) T si *número* es estrictamente mayor que cero, si no devuelve NIL

(minusp *número*) T si *número* es estrictamente menor que cero, si no devuelve NIL

(oddp *número*) T si *número* es un entero impar, si no devuelve NIL

(evenp *número*) T si *número* es un entero par, si no devuelve NIL

Otros predicados numéricos: rationalp, complexp, numberp

Ejemplos:

(typep '(1 2 3) 'cons) → t

(typep 'a 'symbol) → t

(setq a 3)

(typep a 'number) → t

(typep '() 'cons) → nil

(typep '() 'list) → t

(null ()) → t

(setq a 4)

(null a) → nil

(null ‘a) → nil

(setq a 2)

(symbolp a) → nil

(symbolp ‘a) → t

(setq a ‘b)

(symbolp a) → t

(atom ‘a) → t

(setq a 2)

(atom a) → t

(atom ‘()) → t

- **Predicados de igualdad:**

  - Presentamos los distintos predicados ordenados de menor a mayor grado de exigencia de igualdad entre los argumentos recibidos. Así, equal se utiliza para comparaciones genéricas, eql para números y caracteres y eq para objetos idénticos

(equal *x y*) T si los argumentos son estructuralmente iguales (es decir, isomorfos). Para que dos objetos sean equal basta con que su representación impresa es la misma

(equal ‘a ‘b) → nil

(equal ‘a ‘a) → t

(equal 3 3) → t

(equal 3 3.0) → t

(equal (cons ‘a ‘(b)) (cons ‘a ‘(c))) → nil

(equal (cons ‘a ‘(b)) (cons ‘a ‘(b))) → t

(eql *x y*) T si los objetos son conceptualmente idénticos. Es decir, si son números del mismo valor y tipo o si son caracteres iguales o si son eq

(eql ‘a ‘a) → t

(equal 2 2) → t

(eql #\a #\a) → t

(equal (cons ‘a ‘(b)) (cons ‘a ‘(b))) → nil

(eq *x y*) Comprueba si sus argumentos son el mismo objeto (misma posición de memoria). Los números y caracteres no deben ser comparados con este predicado

(eq ‘a ‘a) → t

(equal (cons ‘a nil) (cons ‘a nil)) → nil

- **Predicados de igualdad entre números:**
  - Sólo se aplican sobre argumentos que devuelven valores numéricos

    = T si los argumentos tienen el mismo valor

    /= T si los argumentos tienen distinto valor

  Otros: < > <= >=

  (= 3 3) → t

  (/= 5 5 8) → t

  (< 5) → nil

  (> 3 2 2 1) → nil

(>= 3 2 2 1) → t

- **Predicados para listas:**

(member *elemento lista* [:test *pred*]) Comprueba si un elemento pertenece a una lista. Si *elemento* se encuentra en la *lista* (comparándolos con eql) devuelve el resto de la lista a partir de dicho *elemento*. El parámetro "clave" :test (ver la explicación de los **parámetros clave** más adelante), si se especifica en la llamada a la función, sirve para cambiar el criterio de comparación

```
> (member 'buscado '(el (buscado) mas buscado
                                    de la lista))

(BUSCADO DE LA LISTA)

> (member '(buscado) '(el (buscado) mas buscado
                                    de la lista) :test #'equal)

((BUSCADO) MAS BUSCADO DE LA LISTA)
```

6.4 Funciones lógicas

(not *objeto*) Devuelve t si el *objeto* es nil o la lista vacía (), en caso contrario devuelve falso (o sea nil)

(not (oddp 5)) → nil

(and *arg1 arg2 … argN*) Devuelve el resultado de evaluar el último argumento cierto (/= nil). Es decir, evalúa sus argumentos de izquierda a derecha hasta que uno

devuelve nil o hasta que ya no quedan más argumentos

(and 5) → t

(or *arg1 arg2 … argN*) Devuelve el resultado de evaluar el primer argumento cierto. Es decir, evalúa sus argumentos de izquierda a derecha hasta que uno devuelve algo diferente a nil

(or (not 5) (+ 4 5)) → 9

- Pueden utilizarse como estructuras de control

  - Dadas una secuencia de acciones independientes A_1, A_2, A_3, … A_n, (llamadas a funciones), se desea que se ejecute cada una de ellas sí y sólo sí las anteriores se han realizado con éxito (ver más adelante la explicación de la primitiva cond)

```
(and a b … z) ≡ (cond ((not a) nil)
                      (not b) nil)
                      …
                      (t  z))
```

  - Dadas una secuencia de acciones independientes A_1, A_2, A_3, … A_n, (llamadas a funciones), se desea que se ejecuten una a una dichas acciones hasta que alguna devuelva un valor distinto de nil, dejando sin evaluar el resto de las funciones en la secuencia

```
(or a b … z) ≡ (cond (a)
                     (b)
                     …
                     (t  z))
```

Algunos ejemplos:

```
> (not (null (member 1 '(3 1 5))))

t

> (or  (member 1 '(3 2 5))
       (member 1 '(3 1 5))

(1 5)

> (and (or  (member 1 '(3 2 5))
            (member 1 '(3 5 8))
            (list (member 1 '(3 1 5))))

(NIL)

> (not (and (or  (member 1 '(3 2 5))
                 (member 1 '(3 5 8))
                (member 2 '(3 1 5))))

T
```

Advertencia: aunque not y null se comportan exactamente igual, por claridad conviene distinguir su utilización

not → determina si el valor es falso

null → determina si la lista está vacía

```
> (not (member 'b '(a b c))

NIL

> (null (cddr (member 'b '(a b c))))

t
```

7. Definición de funciones

Las funciones permiten realizar una secuencia ordenada de acciones (llamado “cuerpo de la función”) a partir de una serie de argumentos.

La definición de funciones requiere especificar el nombre de los argumentos dentro de la misma. A los nombres de dichos argumentos se les conoce como “parámetros”.

Una característica esencial es que los argumentos de las funciones, al contrario que los de las macros (ver más adelante), son evaluados antes de ser asignados a los parámetros respectivos

Existen dos tipos de funciones en COMMON LISP

- **Definición de funciones anónimas:**

- Las llamadas expresiones lambda constituyen la forma básica de representar funciones

  (lambda (*parámetro*$_1$ *parámetro*$_1$ … *parámetro*$_n$)
  cuerpo-de-la-función)

- Una función anónima es aquella expresión lambda que permite definir in situ un conjunto de acciones que se quieren realizar a partir de una serie de argumentos

- Tiene que haber el mismo número de parámetros que de argumentos en la llamada a la función

- Los argumentos "se cogen" uno a uno, de izquierda a derecha de los valores devueltos por la evaluación de las expresiones situadas a la derecha de la propia expresión lambda
- Devuelve la evaluación de la última expresión dentro del cuerpo de la función (conjunto de expresiones que la forman)

```
> ((lambda (x y) (list x y) (cons (car x) (cdr y)))
        '(ana ha acertado) '(juan ha fallado))

(MARIA HA FALLADO)
```

- **Definición de funciones con nombre**
  - Construcción de funciones que serán invocadas fuera del contexto donde se definieron

(defun nombre_función (*parámetro*$_1$ *parámetro*$_2$… *parámetro*$_n$) cuerpo-de-la-función)

  - Devuelve el nombre de la función

```
> (defun rotate (L) (append (last L) (butlast L)))

ROTATE

> (rotate '(a b c))

(C A B)
```

  - Al escribir algo como (rotate '(a b c)) ocurren una serie de acciones dentro del bucle read-eval-print:

1. se evalúa el argumento: (quote (a b c)) → (a b c)
2. al parámetro L se le asigna temporalmente el valor del argumento
3. las expresiones que forman el cuerpo de rotate se evalúan en orden

4. rotate devuelve el valor de la última expresión evaluada dentro de su cuerpo y el parámetro L vuelve a su estado original

```
> (setq a '(1 3 5)
(1 3 5)
> (defun test-argumentos (a b) (print a) (print b)
                                        (* a b))
TEST-ARGUMENTOS
> (test-argumentos 3 4)
3
4
12
> b
UNBOUND VARIABLE: B
> a
(1 3 5)
```

Nota: utilizar "a" y "b" como nombres de los parámetros de test-argumentos no afecta los valores que pudieran tener variables con dichos nombres (p.ej. "a") fuera del contexto de la definición de la función

```
> (defun cuenta-atomos (L)
        (or (and (null L) 0)
            (and (atom L) 1)
            (+ (cuenta-atomos (car L))
               (cuenta-atomos (cdr L)))))
CUENTA-ATOMOS
> (cuenta-atomos '(1 5 7 9 11))
5
```

```
> (cuenta-atomos '(esto (es (una)) lista
                        (con (siete (elementos)))))
7
```

- **Definición de funciones con un número variable de argumentos:**

1. Construcción de funciones que pueden tener cero o un *número determinado* de argumentos opcionales:

(defun nombre_función (*parámetro*$_1$ *parámetro*$_2$ … *parámetro*$_n$ &optional *par-opt*$_1$ (*par-opt*$_2$ *valor-inicial*$_2$) … *par-opt*$_n$) cuerpo-de-la-función)

- Los argumentos recibidos en la llamada a la función se asignan secuencialmente, de izquierda a derecha, a los parámetros especificados en su definición. Si hubiera parámetros opcionales (aquellos que aparecen en su definición a partir del identificativo &optional) se le asignan el resto de los argumentos

```
> (defun dentro (elem L1 &optional L2)
        (or (and L2 (member elem L1)
                    (member elem L2))
            (member elem L1)))
DENTRO
> (dentro 4 '(1 2 3 4 5))
(4 5)
> (dentro 4 '(1 2 3 4 5) '(1 4 A C))
(4 A C)
```

- Cuando no hay argumentos para los parámetros opcionales, se les asignan, bien los valores iniciales correspondientes a dichos parámetros (*valor-*

inicial$_n$), o bien NIL cuando no existan dichas inicializaciones.

```
> (defun introduce-atomo (L &optional (elem 'a))
                         (cons elem L))

INTRODUCE-ATOMO

> (introduce-atomo '(x y z) 'b)

(B X Y Z)

> (introduce-atomo '(x y z))

(A X Y Z)
```

2. Construcción de funciones que pueden tener un *número indeterminado* de argumentos opcionales

- Para poder recoger un número variable de argumentos opcionales en la llamada a una función se debe introducir en su definición un nombre de variable que se sitúa a la derecha de &rest.
- La variable así especificada aglutinará en una sola lista todos los argumentos que todavía no hayan sido asignados a parámetros anteriores en la lista de parámetros de la definición de la función
- Si no quedan argumentos sin procesar (es decir, sin evaluar y asignar a parámetros previos) entonces se le asigna NIL a la variable que señala "el resto"

```
> (defun introduce-algunos (L &rest atomos)
                         (append atomos L))

INTRODUCE-ALGUNOS

> (introduce-algunos '(a b c) 1 2 3 4)

(1 2 3 4 A B C)

> (defun suma-algunos (L &rest numeros)
                         (apply #'+ numeros))
```

```
SUMA-ALGUNOS
> (suma-algunos 1 2 3 4 5)
15
```

Nota: #'+ es una abreviatura que internamente se traduce en (symbol-function '+), que devuelve el valor de la definición de la función

3. Construcción de funciones que pueden tener *argumentos identificados por su nombre* (y no sólo por su posición en la llamada a la función)

- Los parámetros que permiten identificar argumentos por su nombre se denominan "clave". Son aquellos que aparecen a la derecha de &key

- Los argumentos de este tipo se especifican de forma explícita indicando su nombre seguido de su valor en la llamada a la función (luego tiene que haber un número par de argumentos referidos a los parámetros clave). No importa el orden, lo único que es necesario es nombrarlos mediante "*:nombre-argumento valor-argumento*"

```
> ((lambda (a b &key c d)
                    (list a b c d))
              1  2  :d  8 :c  5)
(1 2 5 8)
```

- Si hay más de un argumento referido al mismo parámetro clave se utiliza el valor obtenido por el primer argumento evaluado

```
> ((lambda (a b &key c d)
                    (list a b c d))
              1  2  :d  8 :d  5)
(1 2 NIL 8)
```

- Los parámetros clave también pueden ser inicializados con un valor por omisión (de forma análoga a los parámetros opcionales)

```
> ((lambda (a b &key c (d 0)) (list a b c d))
                                    1  2  :c  0)
(1 2 5 0)
> ((lambda (a b &key c d) (list a b c d)) 1 2 :c 8)
(1 2 8 NIL)
> ((lambda (a b &key c d) (list a b c d)) :a :b :c :d)
(:A :B :D NIL)
```

4. Construcción de funciones con diversos tipos de argumentos

- Todos los tipos de parámetros (requeridos, opcionales, resto, clave, etc.) pueden ser especificados simultáneamente en la definición de una función

```
(defun nombre_función ({var}*
        [&optional {var | (var [valor-inicio])}*]
        [&rest var]
        [&key {key-var | (key-var [valor-inicio])}*])
        cuerpo-de-la-función)
```

- Cada uno de los tipos anteriores de parámetros puede aparecer o no en la definición de la función

- El procesamiento de los argumentos con respecto a los parámetros siempre se realiza de izquierda a derecha. Así, al primer parámetro de la función se le asignará el valor devuelto al evaluar el primer argumento

- La asignación de parámetros se realiza siguiendo un proceso predeterminado:

1. En primer lugar se comprueba si hay **parámetros requeridos**, en cuyo caso se les asignan el resultado de la evaluación de los argumentos. Después de haber procesado este tipo de parámetros puede o no haber otros tipos

2. Si existe &optional, los parámetros especificados a continuación, y hasta que no aparezca un nuevo tipo de parámetros (&rest, &key, etc.), se consideran **opcionales** y se evalúan conforme a las normas ya señaladas. Después de haber procesado este tipo de parámetros puede o no haber otros tipos

3. Si existe &rest, el **parámetro resto** especificado a continuación debe ser único (hasta que no aparezca un nuevo tipo de parámetros) y se evalúa conforme a las normas ya señaladas. Después de haber procesado este tipo de parámetros puede o no haber otros tipos

4. Si existe &key, los parámetros especificados a continuación se consideran **parámetros clave** y se procesan conforme a las normas ya señaladas. Después de haber procesado este tipo de parámetros puede o no haber otros tipos (p.ej. &allow-other-keys, &aux)

Nota: si aparecen &rest y &key, todos los argumentos que no hayan sido previamente procesados se utilizan para ambos

Ejemplos:

```
> ((lambda (a &optional (b 3) &rest x
                &key c (d 1)) (list a b c d x)) 1)
(1 3 NIL 1 ())
```

```
> ((lambda (a &optional (b 3) &rest x
                &key c (d 1)) (list a b c d x)) 5 'b :c 8)

(5 B 8 1 (:C 8))

> ((lambda (a &optional (b 3) &rest x
                &key c (d 1)) (list a b c d x)) 1 6 :c 7)

(1 6 7 1 (:C 7))

> ((lambda (a &optional (b 3) &rest x
                &key c (d 1)) (list a b c d x))
                1 6 :d 8 :c 9 :d 10)

(1 6 9 8 (:D 8 :C 9 :D 10))
```

8. Documentación del código

Los programas en Common Lisp se almacenan en ficheros

Los nombres de los ficheros dependen del "sistema de ficheros" proporcionado por el sistema operativo sobre el cual se esté trabajando. Una convención muy frecuente es utilizar la extensión ".lsp" para ficheros con el código fuente y ".fas" o ".fasl" para los que contengan el compilado

La claridad del código permite detectar errores de contenido, a la vez que facilita su corrección, actualización y divulgación

La documentación es el primer paso para lograr diseñar un código claro y efectivo a la vez que eficiente

Los comentarios en los ficheros de Common Lisp se designan poniendo ";" delante del texto correspondiente.

El evaluador de expresiones ignora cualquier texto que aparezca entre un punto y un carácter de “retorno” (fin de línea en el fichero)

- Se siguen las siguientes convenciones para documentar el código:

  1. Las cabeceras, secciones y subsecciones se documentan escribiendo delante del comentario correspondiente la secuencia “;;;;”

  - Por ejemplo, al principio de un fichero podrían aparecer los siguientes comentarios:

```
;;;;    -*- Mode: LISP; Package: USER -*-
;;;; ------------------------------------------------------------------
;;;;                        © Juan Nihonkoku
;;;;                   Creado: Fri Oct 10 13:09:04 1996
;;;; ------------------------------------------------------------------
;;;; Nombre del fichero: EJERLISP.LSP
;;;; Autor: Juan Nihonkoku
;;;;
;;;; Objetivo: este fichero contiene las soluciones
;;;;   de los problemas de los ejercicios del curso de
;;;;   Common Lisp del año 1996
;;;;
;;;; Historia
;;;; 10/10/96 Juan N Crea una primera version de los
;;;;                        ejercicios basicos
;;;;
;;;;  Fin de cabecera
```

  2. La secuencia “;;;” precede partes significativas del programa (secciones, funciones, etc.)

  - Por ejemplo:

```
;;; Esta seccion contiene las funciones que resuelven
```

```
;;; los problemas basicos relativos a la manipulacion de
;;; listas
;;;
;;; Ejercicio 1.7
;;;
;;; Funcion que introduce un elemento nuevo en una
;;; lista. Para ello comprueba primero si el elemento ya
;;; ya pertenece a dicha lista
(defun introduce-nuevo (elem lista)
                        … … … …)
```

2. La secuencia ";;" se utiliza para documentar el código que se encuentra justo debajo del comentario

- Por ejemplo:

```
(defun introduce-nuevo (elem lista)
… … … …
;; basta con que el elemento tenga la misma apariencia
            (or (member elem lista :test #'equal)
                (cons elem lista))
… … … …)
```

3. Con ";" se documentan las expresiones incluidas en la misma línea

- Por ejemplo:

```
(defun introduce-nuevo (elem lista)
 (or (and (null lista) (list elem)) ; si la lista estaba vacía
 … … … …)
```

9. Condicionales y Estructuras de Control

9.1 Condicionales

Hasta ahora hemos visto la utilidad de los operadores lógicos (and, or y not) para seleccionar la evaluación de secuencias de sentencias alternativas

• Existen una serie de primitivas en Common Lisp especialmente pensadas para controlar la lógica del programa:

(if *test parte-then* [*parte-else*])

- Si el resultado de la evaluación de *test* no es NIL (es cierto), entonces evalúa *parte-then*, si no evalúa *parte-else*

```
> (defun mi-consp (L) (if (atom L) nil t))
MI-CONSP
> (mi-consp 'a)
NIL
> (mi-consp '(esto es una lista))
T
```

(when *test* $sentencia_1$ $sentencia_2$ … $sentencia_n$)

- Primero evalúa *test*, si el resultado es NIL, entonces no se evalúa ninguna de las *sentencias* y se devuelve NIL. En caso contrario, se evalúan todas las *sentencias* de izquierda a derecha y se devuelve el valor de la última

```
> (setq x 5/3)
5/3
> (when (ratiop x) (setq x (numerator x)) (* x x))
25
```

(unless *test* $sentencia_1$ $sentencia_2$ … $sentencia_n$)

- Primero evalúa *test*, si el resultado no es NIL, entonces no se evalúa ninguna de las *sentencias* y se devuelve NIL. En caso contrario, se evalúan todas las *sentencias* de izquierda a derecha y se devuelve el valor de la última

```
> (defun introduce-nuevo (elem lista)
        (unless (member elem lista)
                (push elem lista)))
INTRODUCE-NUEVO
> (setq primera-lista '(a b c d)
(A B C D)
> (introduce-nuevo 'x '(a b c d)
(X A B C D)
> primera-lista
(X A B C D)
```

Nota: push es una función **destructiva** que introduce un elemento al principio de una lista, es decir, modifica el contenido de la lista que recibe como argumento. El efecto sobre la lista (push elem lista) equivale a (setq lista (cons elem lista)). Por tanto, **es muy importante tener en cuenta si la función utilizada es o no destructiva**

(when (and x (consp x)) (cuenta-atomos x))
es equivalente a:
(unless (or (null x) (atom x)) (cuenta-atomos x))

(cond (*test*$_1$ *sentencia*$_{11}$ *sentencia*$_{12}$ … *sentencia*$_{1n}$)
(*test*$_2$ *sentencia*$_{21}$ *sentencia*$_{22}$ … *sentencia*$_{2n}$)
(…) …)

- cond es la sentencia condicional del lenguaje por excelencia. Como su propio nombre indica se utiliza para evaluar cláusulas de forma condicional

Nota: se llaman "cláusulas" a cada una de las listas (*test* *sentencia*$_1$ *sentencia*$_2$ … *sentencia*$_n$)

- Devuelve el valor de la última *sentencia* de la primera cláusula cuyo *test* sea distinto de NIL. Las *sentencias* se evalúan secuencialmente de izquierda a derecha

- Es correcto incluir una última cláusula (t nil), que será seleccionada cuando todas las anteriores hayan fallado

```
(cond ((atom L) 'atomo)
      ((listp L) 'lista)
      ((numberp L) 'número)
      (t nil))
```

Ejemplos:

```
> (defun or-logico (arg1 arg2 arg3)
      (cond ((atom L) 'atomo)
            ((listp L) 'lista)
            ((numberp L) 'número)
            (t nil))

OR-LOGICO
```

```
> (or-logico 'x '(a b c d)
        (consp "esto es una ristra de caracteres")
        (cons 'esto '(es una lista))
        (integerp 8))

(ESTO ES UNA LISTA)

> (defun and-logico (arg1 arg2 arg3)
        (cond (arg1 (cond arg2
                          (cond (arg3 arg3))
                                (t nil)))
                    (t nil)))
                (t nil)))

AND-LOGICO

> (and-logico (setq a '(1 2 3)
                (nconc a '(a))
                (reverse a))

(A 3 2 1)
```

Nota: nconc es la versión **destructiva** de append

(case *forma-clave*
(*clave*$_1$ *sentencia*$_{11}$ *sentencia*$_{12}$ … *sentencia*$_{1n}$)
(*clave*$_2$ *sentencia*$_{21}$ *sentencia*$_{22}$ … *sentencia*$_{2n}$)
(…) …)

forma-clave sí se evalúa

las *clave*s no se evalúan

- *case* actúa de forma parecida a *cond*, en este caso se contrasta *forma-clave* con cada una de las *clave*s
- Si la *clave* es un átomo entonces se utiliza la función eql para compararlas
- Si la *clave* es una lista se utiliza la función member para compararlas

```
> (defun dias-mes (mes ano)
      (case mes
       ((enero marzo mayo julio agosto
                          octubre diciembre) 31)
       ((abril junio septiembre noviembre) 31)
       (febrero (if (zerop (mod year 4)) 29 28)))
DIAS-MES
> (dias-mes 'febrero 1996)
29
```

(typecase *forma-clave*
(*tipo*$_1$ *sentencia*$_{11}$ *sentencia*$_{12}$ … *sentencia*$_{1n}$)
(*tipo*$_2$ *sentencia*$_{21}$ *sentencia*$_{22}$ … *sentencia*$_{2n}$)
(…) …)

- Su funcionamiento es muy parecido a *case* pero las comparaciones se hacen con los *tipo*s que pudiera tener *forma-clave*

```
(typecase x
      (string "es una ristra de caracteres")
      (integer "es un entero")
      (symbol "es un simbolo")
      (otherwise "no es un tipo controlado")
```

9.2 Estructuras de Control

9.2.1 Asignación de variables

• Algunas estructuras de control sirven para crear asignaciones de valores a variables en un determinado contexto (es decir, crean variables locales)

(let ((*var*$_1$ *valor*$_1$) (*var*$_2$ *valor*$_2$) …)
cuerpo)

- Ejecuta el *cuerpo* con cada variable ligada a su correspondiente *valor*
- El establecimiento de las asociaciones (variable *valor*) se realiza simultáneamente ("en paralelo") en todas las variables, lo cual impide que una variable pueda depender del valor de una variable que fuera anterior en la secuencia
- Devuelve el valor de la última expresión evaluada en el *cuerpo*
- Es muy útil para almacenar resultados intermedios

(let* ((*var*$_1$ *valor*$_1$) (*var*$_2$ *valor*$_2$) …)
cuerpo)

- let* se comporta igual que let excepto en que el establecimiento de las asociaciones (variable *valor*) se realiza secuencialmente, lo cual permite que una variable pueda depender del valor de una variable que fuera anterior en la secuencia

Ejemplos:

```
> (setq x 15)
15
> (let ((x 3) (y x)) y)
15
> (let* ((x 3) (y x)) y)
3
> (defun introduce-nuevo (primero segundo)
        (let ((elem (cond ((listp primero) segundo)
                          (t primero)))
              (lista (cond ((listp segundo) segundo)
                          (t primero)))) (cond
```

```
(cond ((member elem lista) lista)
      (t (cons elem lista)))))
```

```
INTRODUCE-NUEVO
> (introduce-nuevo 'w '(c d e a f w1)
 (W C D E A F W1)
```

9.2.2 Establecimiento de secuencias

(prog (*var*$_1$ *var*$_2$ (*var*$_3$ *inicio*$_3$) *var*$_4$ (*var*$_5$ *inicio*$_5$) …)
cuerpo)

- El establecimiento de las asociaciones (variable valor de *inicio*) se realiza en paralelo en todas las variables, dejando de tener vigencia dichas asociaciones fuera del contexto del prog
- Cuando las variables aparecen solas (p.ej. *var*$_1$) se les asigna el valor nil
- prog devuelve nil después de haber evaluado la última expresión del *cuerpo*, a no ser que dentro del cuerpo de prog se evalúe alguna sentencia return

(return *resultado*)

- cuando una de las sentencias del *cuerpo* es return no se evalúan el resto de las sentencias y cualquiera que sea el valor obtenido al evaluar *resultado* (se puede omitir "(return)", en cuyo caso se devolvería nil) es el valor finalmente devuelto por la sentencia prog
- prog* realiza las mismas funciones que prog con la salvedad de que la asignación de valor a las variables se realiza en secuencia en lugar de simultáneamente

(progn *sentencia*$_1$ *sentencia*$_2$ … *sentencia*$_n$ …)

- Evalúa las *sentencia*s de izquierda a derecha y devuelve el valor obtenido al evaluar la última sentencia (si esta estuviera formada por otras tantas devolvería el valor de la última evaluada)

- Iteración: do

Descripción general:

(do ((*inicializacion-variables-y-resto-actualizaciones*)
(*condicion-parada resultado-final*)
cuerpo)

Descripción más detallada:

(do ((*var*$_1$ *inicio*$_1$ *paso*$_1$)
… … …
(*var*$_n$ *inicio*$_n$ *paso*$_n$))
(*test-final resultado*)
cuerpo)

El "bucle" definido por esta sentencia es el siguiente:

1. Si es la primera iteración se evalúan todas las sentencias *inicio* (aquellas que no aparezcan "(*var*)" se consideran nil) y sus valores se les asignan a las variables correspondientes

2. Si no es la primera iteración se evalúan las sentencias *paso* y se actualizan con los valores devueltos las variables correspondientes. Pudiera ocurrir que no se hubiera especificado *paso* para alguna variable, en cuyo caso su valor no se actualizaría en cada iteración

3. Se evalúa el *test-final* y si su valor es distinto de nil se devuelve *resultado* y se abandona el bucle

4. Se evalúa las sentencias que forman el cuerpo y se regresa de nuevo al punto 2.

Ejemplos:

```
> (defun exponente** (m n)
      (do   ((result 1)
             (exponente n))
            ((zerop exponente) result)
          (setq result (* m result))
          (setq exponente (– exponente 1))))

EXPONENTE**

> (exponente** 3 5)

243

> (defun exponente* (m n)
      (do   ((result 1 (* m result))
             (exponente n) (– exponente 1))
            ((zerop exponente) result)))
              ;; exponente sin cuerpo

EXPONENTE*

> (defun nuevo-reverse (lista)
      (do   ((L lista (cdr L))
             (result nil) (cons (car l) result)))
           ((null l) result)))

NUEVO-REVERSE

> (nuevo-reverse '(a b c d))

(D C B A)
```

- Iteración: do*

- Tiene la misma sintaxis que do pero la asignación de valores a las variables se realiza secuencialmente

```
> (defun exponente (m n)
    (if (zerop n) 1
      (do   ((result m (* m result))
             (exponente n) (– exponente 1))
             (contador (– exponente 1)
                       (– exponente 1))
        ((zerop contador) result)))
EXPONENTE
```

- Iteración: dotimes

(dotimes ((*var contador* [*resultado*])
cuerpo)

- Realiza el *cuerpo* un número determinado *contador* de veces devolviendo el valor de *resultado*, excepto cuando en el cuerpo se evalúa alguna sentencia return, en cuyo caso se devuelve el valor indicado por ésta
- La variable *var* en la primera iteración vale 0 y en la última vale (– *contador* 1)

Ejemplo:

Teniendo en cuenta que:

(floor *numero divisor*)

- Devuelve el mayor entero menor o igual que el cociente entre el *numero* y el *divisor*

```
> (defun palindromop (L &optional (inicio 0)
                                  (final (length L)))
      (dotimes (k (floor (– final inicio) 2) t)
            (unless
               ;; elementos desde el principio
               (equal (nth (+ inicio k) L)
               ;; elementos desde el final
                     (nth (– final k 1) L))
               (return nil))))

PALINDROMOP

> (palindromop '(esto es esto))

 T
```

- Iteración: dolist

(dolist ((*var lista* [*resultado*])
    cuerpo)

- La evaluación de *lista* debe ser una lista
- Realiza tantas iteraciones como elementos (de más alto nivel) tenga la *lista*
- En cada iteración *var* tiene asignado como valor el elemento de la lista cuya posición se corresponde con dicha iteración
- Al terminar el proceso devuelve el valor *resultado*, excepto cuando en el cuerpo se evalúa alguna sentencia return, en cuyo caso se devuelve el valor indicado por ésta

```
> (dolist (x '(a b c))
      (print x))
A
B
C

NIL
```

9.2.3 Funciones de transformación de listas

• Son funciones que acceden iterativamente a los elementos o "colas sucesivas" que componen una lista, realizan acciones en función de dicho elemento o cola y devuelven una lista con los resultados de dichas acciones o devuelven la lista original (en este caso lo importante es el resultado de cada acción y no el valor devuelto)

• Funciones que actúan sobre cada elemento de la lista

(mapcar *función* $list_1$ $list_2$ …)

- Aplica la *función* a los car de cada lista luego a los cadr y así sucesivamente hasta que se alcance el final de la lista más corta

- La *función* debe tener tantos argumentos como listas

- Devuelve una lista con los resultados dados por las sucesivas aplicaciones de *función*

```
> (mapcar #'+ '(7 8 9) '(1 2 3))
(8 10 12)
> (mapcar #'oddp '(7 8 9))
(T NIL T)
```

(mapc *función* $list_1$ $list_2$ …)

- Actúa como mapcar salvo que no acumula los resultados de las sucesivas aplicaciones de *función*

- Devuelve su segundo argumento

```
> (mapc #'(lambda (arg)
          (print (list arg arg))) '(1 2 3))
(1 1)
(2 2)
(3 3)

(1 2 3)
```

(mapcan *función* $list_1$ $list_2$ …)

- Actúa como mapcar pero utiliza la función nconc para combinar los resultados de las sucesivas aplicaciones de *función*

```
> (mapcan #'(lambda (x)
                (and (numberp x) (list x)))
          '(a 1 b c 3 4 d 5))

(1 3 4 5)
```

- Funciones que actúan sobre las colas sucesivas de la lista

(maplist *función* $list_1$ $list_2$ …)

- Aplica la *función* a las listas y a sus sucesivos cdrs (cddr, cddr, etc.)
- La *función* debe tener tantos argumentos como listas
- Devuelve una lista con los resultados dados por las sucesivas aplicaciones de *función*

```
> (maplist #'(lambda (x y)  (list x y))
          '(a b c) '(1 2 3))
(((A B C) (1 2 3)) ((B C) (2 3)) ((C) (3)))
```

(mapl *función* $list_1$ $list_2$ …)

- Actúa como maplist salvo que no acumula los resultados de las sucesivas aplicaciones de *función*
- Devuelve su segundo argumento

```
> (mapl #'(lambda (x y) (print (list x y)))
        '(a b c) '(1 2 3))
((A B C) (1 2 3))
((B C) (2 3))
((C) (3))
(A B C)
```

(mapcon *función* $list_1$ $list_2$ …)

- Actúa como maplist pero utiliza la función nconc para combinar los resultados de las sucesivas aplicaciones de *función*

```
> (mapcon #'(lambda (x y)  (list x y))
        '(a b c) '(1 2 3))
((A B C) (1 2 3) (B C) (2 3) (C) (3))
```

Resumen de funciones de transformación de listas:

	Opera sobre CARs	Opera sobre CDRs
Efectos laterales	MAPC	MAPL
La lista devuelta se construye con LIST	MAPCAR	MAPLIST
La lista devuelta se construye con NCONC	MAPCAN	MAPCON

9.2.4 Invocación de funciones

(apply *función* arg_1 arg_2 …)

- Aplica una *función* a una lista de argumentos
- El último argumento debe ser una lista
- La lista de argumentos se construye concatenando el último *arg* (como con append) a la lista formada con todos los anteriores *arg*s

```
> (apply '+ 1 2 '(3 4))
10
```

(funcall *función* arg_1 arg_2 …)

- Aplica una *función* a una serie de argumentos
- La *función* no puede ser una macro ni una forma especial (if, let, setq, etc.)

```
> (setq funciones-lista '(append list nconc))
(APPEND LIST NCONC)
> (funcall (caddr lista) '(a b c d) '(1 2 3) '(x y z))
(A B C D 1 2 3 X Y Z)
```

9.2.5 Valores múltiples

- A veces es conveniente que las funciones devuelvan más de un valor, sin que para ello se deba recurrir a su agrupación en una lista
  - Anteriormente hemos utilizado la función numérica floor

(floor *numerador* [*divisor*])

- Además de devolver el mayor entero menor o igual que el cociente entre numerador y divisor, devuelve como segundo valor el resto de la división

- Por omisión divisor es 1

```
> (floor 3 2)
1
1
```

(values &rest *args*)

- Devuelve un sólo valor por cada uno de los resultados de las evaluaciones de cada argumento

```
> (defun doble-exponente (x y)
        (values (expt x y) (expt y x)))
DOBLE-EXPONENTE
> (doble-exponente 2 3)
8
9
```

- La sentencia (values) al final de una función es la forma estándar de lograr que una función no devuelva valor alguno

(multiple-value-setq (*var*$_1$ *var*$_2$ …) *sentencia*)

- Se asignan sucesivamente los valores devueltos por *sentencia* a las variables incluidas en la lista de variables

- Si hubiera más variables que valores, a estas se les asigna nil

```
> (multiple-value-setq (cociente resto)
                       (floor 17 3))
cociente → 5
resto → 2
```

(multiple-value-bind (*var*$_1$ *var*$_2$ …) *sentencia*
sentencia$_1$ *sentencia*$_2$ … *sentencia*$_n$)

- Funciona de forma análoga a let. Los valores devueltos por *sentencia* se asignan localmente a las variables incluidas en la lista de variables y luego se ejecutan sucesivamente el resto de las sentencias (*sentencia*$_1$ *sentencia*$_2$, etc.)

- Si hubiera más variables que valores, a estas se les asigna nil

```
> (multiple-value-bind (cociente resto)
                       (floor 17 3) (* cociente resto))
10
```

10. Macros y Corrección de Errores

10.1 Macros

Ventajas:

- Hacen que el código sea más claro y sencillo de escribir y entender. Permiten definir funciones que convierten cierto código Lisp en otro, generalmente más prolijo. Esta conversión se realiza antes de que el código sea evaluado o compilado
- Al igual que en el resto de los lenguajes de programación, las macros se diferencian de las funciones en que a la hora de generar el código compilado las macros “se expanden en línea” (es decir, cada llamada a una macro obliga al compilador a repetir el código de la macro in situ)
- Ayuda a escribir código modular
- Son útiles para definir ampliaciones en las sentencias incluidas en el lenguaje
- Eliminan tener que escribir repetidas veces partes complicadas del código a lo largo del programa

Desventajas:

- Si se está ejecutando código compilado deben recompilarse todas las llamadas a las macros que se hayan modificado
- Son difíciles de depurar

- En las llamadas a funciones se evalúan todos los argumentos y posteriormente la función se aplica a los resultados de dichas evaluaciones

- ¡ Las macros provocan una doble evaluación!

  - En la primera “pasada” se produce una forma o sentencia de Lisp
  - En la segunda se evalúa de nuevo el resultado producido en la primera pasada y se produce el valor correspondiente

(defmacro *nombre lista-lambda* {*sentencias*}*)

- No se evalúan los argumentos
- Pueden tener una gran variedad de argumentos; la lista-lambda es semejante a la de las funciones
- Dado que se ubican en la mismo espacio de memoria (celda) que se reserva para el código de la función de un determinado *nombre*, no puede haber una macro y una función que tengan el mismo nombre (recordar la función “(symbol-function *nombre*)”)

```
> (defmacro mi-unless (test &rest cuerpo)
            (list ‘cond (cons (list ‘not test) cuerpo)
                        (list t nil)))

MI-UNLESS

> (setf a 3)

NIL

> (mi-unless (list a) (list a))

(3)
```

```
> (macroexpand ‘(mi-unless (list a) (list a)))
(IF (NOT (ATOM A)) (LIST A) NIL);
T
```

10.2 Utilización del Backquote

• El carácter de comilla invertida (“`”) simplifica la escritura del código que es doblemente evaluado dentro del cuerpo de las macros

• Se utiliza para realizar una evaluación selectiva del código que aparece después de dicho carácter

- En la primera “pasada”, el código que se encuentra a la derecha del carácter de comilla invertida (“`”) deja de ser evaluado, por tanto su contenido se deja intacto (para ser nuevamente evaluado en el segundo paso si aparece dentro de una macro)
- Si a la derecha del carácter de comilla invertida (“`”) apareciera el carácter de coma (“**,**”) delante de alguna expresión, dicha expresión sí que es evaluada en la primera pasada, y el objeto devuelto en dicha evaluación se introduce en el lugar donde aparecía dicha expresión

  `(a b c) → (a b c)

  (setf c 4)

  `(a b ,c) → (a b 4)

- Si a la derecha del carácter de comilla invertida (“`”) apareciera la combinación de caracteres de coma (“**,**”) seguido de arroba (“**@**”), es decir:

(“**,@**”) entonces la expresión que siga a dicha combinación se evalúa y su valor se introduce en lugar de dicha expresión de tal forma que si su valor es una lista de elementos, dichos elementos se introducen como nuevos elementos en la estructura que contuviera a la expresión y no se introduce como tal lista de elementos

```
(setf b ‘(d e))
`(a ,@b ,c) → (a d e 4)
`(a ,@(cdr b) ,c) → (a e 4)
```

• Dicho de otra forma, podríamos considerar el carácter de comilla invertida (“`”) como el de comienzo de una plantilla en la que nada es evaluado excepto aquellas partes en las que aparezca el carácter de coma (“**,**”) o la combinación de caracteres (“**,@**”)

- Como puede apreciarse en el siguiente ejemplo, la definición de macros se simplifica con su utilización

```
(defmacro mi-unless (test &rest cuerpo)
        `(cond ((not ,test) ,@cuerpo)
               (t nil)))
```

Ejemplos:

```
> (defmacro suma (&rest args)
      `(+ ,@args))
SUMA
> (macroexpand ‘(suma 3 4 5))
(+ 3 4 5);
T
```

```
> (defmacro mi-if (test val-cierto
                                &optional val-falso)
        `(cond (,test ,val-cierto)
               (t      ,val-falso)))

MI-IF

> (mi-if (listp 'a) "es una lista" "no es una lista")

"no es una lista"
```

10.3 Corrección de errores

- En el lenguaje existen algunas primitivas que permiten seguir paso a paso la ejecución del código
- Existen además otras utilidades que dependen de cada entorno, como los depuradores

(trace *nombre-funcion*)

- Activa una traza sobre la función, es decir, cada vez que dicha *función* se invoque en el código se mostrarán sus argumentos y cuando termine de ejecutarse se presentará el valor devuelto
- La *función* no se evalúa por lo que no hace falta añadirle el carácter de coma (“**'**”)
- Si no recibe ningún argumento muestra las funciones cuya traza siga activa en ese momento
- Para desactivar la traza de una función:

(untrace *nombre-funcion*)

- untrace sin argumentos se desactivan todas las trazas que estuvieran activas en dicho instante

```
> (defun factorial (num)
      (if (= num 0) 1 (* num (factorial (1- num)))))
FACTORIAL
> (factorial 5)
1. Trace: (FACT '5)
2. Trace: (FACT '4)
3. Trace: (FACT '3)
4. Trace: (FACT '2)
5. Trace: (FACT '1)
6. Trace: (FACT '0)
6. Trace: FACT ==> 1
5. Trace: FACT ==> 1
4. Trace: FACT ==> 2
3. Trace: FACT ==> 6
2. Trace: FACT ==> 24
1. Trace: FACT ==> 120
120
```

(step *sentencia*)

- Permite interactuar paso a paso con cada una de las funciones incluidas en el código de *sentencia*
- La interacción depende del entorno de ejecución, pero generalmente el carácter de retorno o el de espacio provocan el avance a lo largo de los diferentes pasos

(apropos *string*)

- Muestra todos los símbolos (primitivas, funciones, macros, variables, etc.) cuyo nombre impreso contenga dicho *string* como un *substring*
- Devuelve información sobre dicho símbolo

(describe *objeto*)

- Muestra información sobre *objeto*

(documentation *símbolo tipo-doc*)

- Devuelve el string de documentación asociado a un *símbolo*
- Las variables y funciones pueden documentarse con el fin de obtener dicha documentación en el intérprete de Lisp

```
> (defun palindromop (L &optional (inicio 0)
                                  (final (length L)))
      "Comprueba si una secuencia es simetrica"
      (dotimes (k (floor (– final inicio) 2) t)
            (unless
               ;; elementos desde el principio
               (equal (nth (+ inicio k) L)
               ;; elementos desde el final
                     (nth (– final k 1) L))
               (return nil))))

PALINDROMOP

> (documentation 'palindromop 'function)

"Comprueba si una secuencia es simetrica"
```

- defvar es la sentencia general para crear variables globales al principio del programa

```
> (defvar *tiempo-fact* (time (factorial 5))
      "Tiempo en que se tarda en evaluar el
      factorial de 5")

Execution time: 0.00 seconds (0 clock ticks)

120
```

```
> (documentation '*tiempo-fact* 'variable)
  "Tiempo en que se tarda en evaluar el factorial
       de 5"
```

11. Estructuras de datos

11.1 Funciones destructivas

- Ya se ha comentado el efecto de algunas funciones destructivas (p.ej. push), es decir, aquellas que alteran el valor de sus argumentos (los cambios permanecen después de haber terminado de ejecutarse)

(rplaca *x y*) cambia el car de *x* por *y*

(rplacd *x y*) cambia el cdr de *x* por *y*

(push *elem lista*) introduce *elem* al principio de lista

(pop *lista*) función para extraer el primer elemento de la *lista*

(nbutlast *list* &optional *n*) versión destructiva de butlast

(nconc &rest *listas*) crea una lista con la concatenación de *listas* y modifica las listas originales

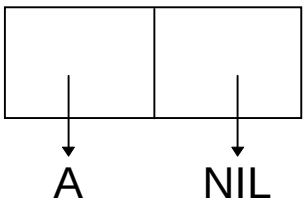
```
> (setq var1 '(a b c d))
(A B C D)
> (setq var2 '(1 2 3 4))
(1 2 3 4)
```

```
> (nconc var1 var2)
(A B C D 1 2 3 4)
> var1
(A B C D 1 2 3 4)
> var2
(1 2 3 4)
> (nconc var1 var2)
(A B C D 1 2 3 4)
> (nconc var1 var1) → stack overflow: circular list
```

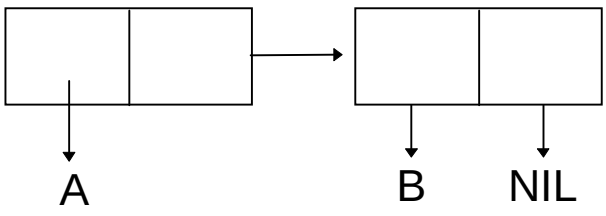
11.2 Par punteado

- Una celda cons es una estructura de registro que contiene dos componentes llamados car y cdr

(A)



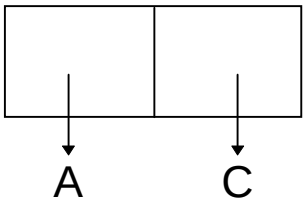
(A B)



- Como un caso especial, un par punteado es una única celda cons y se representa mediante el car y el cdr encerrado entre paréntesis y separados por un punto rodeado por dos espacios en blanco (“ **.** ”)

• Un par punteado es por tanto una celda cons en la que su cdr no es nil

(A C)



- Para crear un par punteado

```
> (setq L (cons ‘a ‘b))
(A . B)
> (setq M (append ‘(1 2) L))
(1 2 A . B)
> (last M)
(A . B)
> (append M ‘(3 4))
(1 2 A 3 4)
```

11.3 Listas de Asociación

• Una lista de asociación, o a-list, es una lista compuestas por pares de elementos (asociados)

• El car de una a-list se llama *clave* y el cdr se conoce como *dato* (*clave* . *dato*)

```
> (setq x (list (cons 1 ‘a) (cons 2 ‘b) (cons 3 ‘c)))
((1 . A) (2 . B) (3 . C))
```

(assoc *clave a-list* [:test *pred*])

- El valor devuelto es el primer par en *a-list* cuyo car sea igual (eql por omisión) a la clave

```
> (assoc 2 x)
(2 . B)
> (assoc 2 x :test #'<)
(3 . C)
```

11.4 Listas de Propiedades

• Uno de los componentes básicos de cualquier símbolo en la memoria es el puntero a su lista de propiedades (plist)

• Una lista de propiedades es una celda de memoria que contiene una lista con un número par de elementos (puede ser 0)

• Cada par de elementos se dice que constituye una entrada de la lista; el primer componente del par se llama nombre de la propiedad y el segundo es su valor

- Para acceder a cada entrada se utiliza la función get

(get *símbolo propiedad*)

- Devuelve el valor asociado a la *propiedad* en la lista de asociación del *símbolo*
- La búsqueda se realiza tomando como base de la comparación el predicado eq
- Si el par no existiera se devuelve nil

(symbol-plist *símbolo*)

- Devuelve la lista de propiedades completa del *símbolo*

(setf (get *símbolo propiedad*) *valor*)

- *get* accede a la posición donde se encuentra el valor actual de la *propiedad* del *símbolo*

- setf es la forma genérica de cambiar el contenido de posiciones que apuntan a objetos de Lisp

```
> (setf (get ‘coche ‘color) ‘rojo)

ROJO

> (symbol-plist ‘coche)

(COLOR ROJO)

>(describe 'coche)


Description of
COCHE
This is the symbol COCHE, has the property
 COLOR.
The symbol lies in #<PACKAGE USER> and is
 accessible in the package USER.
For more information, evaluate
 (SYMBOL-PLIST 'COCHE).

> (setf (get ‘coche ‘categoria) ‘lujo)
        (get ‘coche ‘consumo) ‘alto))

ALTO

> (symbol-plist ‘coche)

(CONSUMO ALTO CATEGORIA LUJO
                    COLOR ROJO)
```

11.5 Estructuras

• Este tipo de datos es semejante a los registros de otros lenguajes de programación

• Las estructuras permiten utilizar una técnica de representación simple y eficiente

• El Common Lisp crea automáticamente funciones de acceso y actualización para cada una de las instancias de las estructuras creadas por el usuario

• Las estructuras proporcionan una extensión de la representación utilizada en las listas de propiedades

- Las programación basada en estructuras constituye una técnica más depurada que la programación basada en listas de propiedades por las siguientes razones:

  Optimización del espacio de almacenamiento

  Facilidad de manipulación

  Flexibilidad

  Segmentación de los datos

• Definición básica de estructuras

(defstruct *nombreestructura*
nombrecampo$_1$
nombrecampo$_2$
... ...
nombrecampo$_n$)

- Todas las funciones de acceso y de creación de instancias de la estructura se crean al mismo tiempo

Creación de instancias:

make-*nombreestructura*

Acceso a los valores de los campos ("slots" [1]):

nombreestructura-nombrecampo

- Para actualizar los valores de los campos de una estructura se utiliza la función genérica de actualización setf

Creación:

```
>(defstruct empleado nombre depto posicion)
EMPLEADO
```

Acceso:

```
>(empleado-depto emp1)
NIL
```

Modificación:

```
>(setq emp1 (make-empleado :nombre 'juan))
#S(EMPLEADO :NOMBRE JUAN :DEPTO NIL
                         :POSICION NIL)
>(empleado-nombre emp1)
JUAN
>(setf (empleado-nombre emp1) 'antonio)
ANTONIO
> emp1
#S(EMPLEADO :NOMBRE ANTONIO :DEPTO
                        NIL :POSICION NIL)
```

[1] Aunque utilicemos la nomenclatura "campo", proveniente del mundo más conocido de las Bases de Datos, conviene tener presente que en el entorno de los problemas de Inteligencia Artificial deberíamos hablar de "slots" [Mira *et al*., 1995].

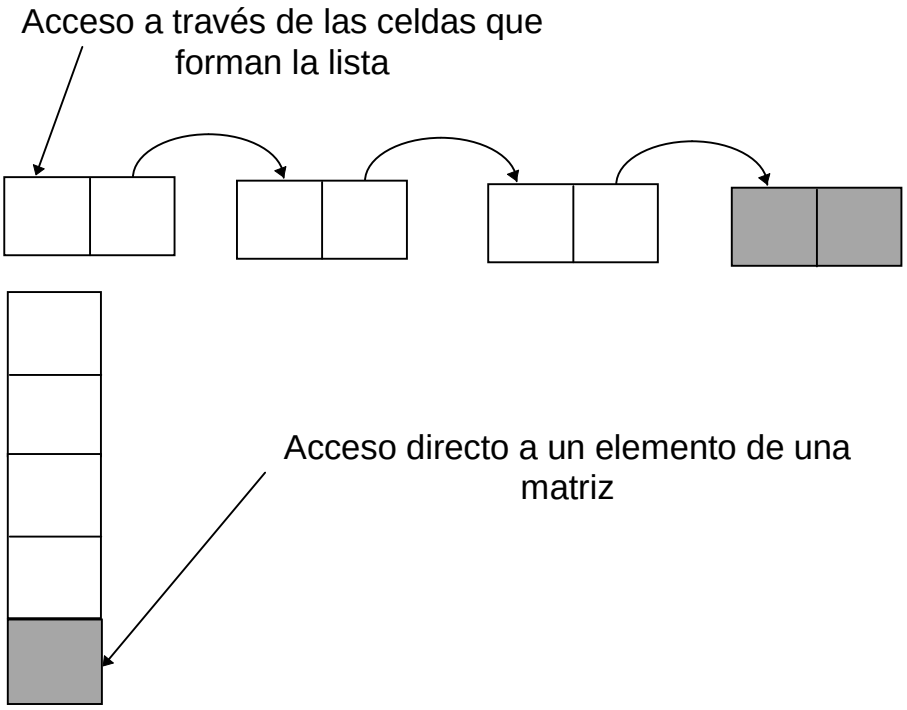
11.6 Matrices y vectores

- Son estructuras de datos con un tamaño prefijado (existen opciones de creación para permitir reajustar el tamaño dinámicamente)

- La gran ventaja es que facilitan el acceso directo al elemento requerido a partir de su posición

- El tiempo de acceso a un elemento concreto es independiente de su posición (al contrario que en las listas)

- En general se puede afirmar que el acceso y la actualización (dada la forma en como se almacenan internamente) es más rápido con las matrices que con las listas



- En general se puede afirmar que la inserción de un nuevo elemento es más rápida con las listas (dada la

flexibilidad de su estructura interna) que con las matrices

Creación de matrices:

(make-array *dimensiones*
[:adjustable *valor*]
[:element-type *tipo*]
{[:initial-element *valorcomun*] |
[:initial-contents *listadeelementos*]})

- *dimensiones* es una lista de enteros no negativos que representa las dimensiones de la matriz. Para el caso de las matrices de una dimensión (vectores) basta con proporcionar un entero
- [:adjustable *valor*] si valor es distinto de nil señala que puede ajustarse dinámicamente el tamaño de la matriz
- [:element-type *tipo*] indica la posibilidad de especificar el *tipo* de elementos. Si *tipo* es t entonces puede ser cualquier tipo (es el valor por omisión)
- [:initial-element *valorcomun*] Crea una matriz con todos sus elementos iguales a *valorcomun*
- [:initial-contents *listadeelementos*] *listadeelementos* es una lista cuya estructura debe corresponderse con el valor dado en *dimensiones*
- Si no se le proporciona ningún argumento, los elementos de la matriz se rellenan con nil
- A cada una de las “dimensiones” de la matriz se le llama eje y tanto los ejes como los elementos

incluidos en cada uno de éstos comienzan a numerarse en el 0

```
>(make-array '(2 2) :initial-element 'a)
#2A((A A) (A A))
>(let ((a (make-array '(2 3)
                :initial-contents '((a b c) (1 2 3))))
      (b (make-array '2
                :initial-element 'a)))
      (values a b))
#2A((A B C) (1 2 3)) ;
#(A A)
```

Acceso y actualización:

(array-dimension *array numero-eje*)

- devuelve el número asociado a la *dimension* del *numero-eje* especificado

```
>(let ((a (make-array '(2 3)
                :initial-contents '((a b c) (1 2 3)))))
      (values (array-dimension a 0)
                (array-dimension a 1)))
2 ;
3
```

(array-rank *array*)

- devuelve el número de dimensiones de la matriz

```
>(array-rank (make-array '(2 3)))
2
>(array-rank (make-array '(2 3 4)))
3
```

(aref *array* $dimen_1$ $dimen_2$ … $dimen_n$)

- es la función general de acceso a los elementos de una matriz

- cada uno de los valores de representa la posición del elemento que se quiere recuperar en el eje correspondiente, tiene que haber tantos valores como indique el rango de la matriz

```
>(let ((a (make-array '(2 3)
                :initial-contents '((a b c) (1 2 3)))))
        (aref A 1 2))

3

>(let ((a (make-array '(2)
                :initial-contents '((a b c) (1 2 3)))))
        (aref A 0))

(A B C)

>(setq a (make-array '(2 3)))

#2A((NIL NIL NIL) (NIL NIL NIL))

>(setf (aref a 1 2) 1)

1

> a

#2A((NIL NIL NIL) (NIL NIL 1))
```

- Un vector es una matriz de una sola dimensión

Creación de vectores:

(vector $elem_1$ $elem_2$ … $elem_n$)

- crea un vector cuyos elementos son $elem_i$

```
>(vector 'juan 1 'x)

# (JUAN 1 X)
```

12. Entrada/Salida

• Los flujos de datos (“streams”) permiten leer y escribir datos en un fichero

• Un flujo es un objeto que se utiliza como fuente o sumidero de datos

• El flujo puede ser de caracteres, enteros, binario

• La comunicación del bucle read-eval-print con el usuario se realiza a través de flujos de datos (o canales)

- Existen variables que contienen los streams estándares que permiten dicha comunicación

standard-input es la fuente de donde se leen los datos del usuario

standard-output es el stream donde se escriben los datos que recibe el usuario

• El usuario también puede crear sus propios flujos de datos

- Una de sus principales funciones es permitir la lectura y escritura de datos en un fichero

(with-open-file (*stream nombrefichero* {opciones}*)
{*sentencias*}*)

- Esta función se encarga de abrir un flujo de datos con un fichero para permitir realizar acciones con su contenido a través de las *sentencias* evaluadas como si hubiera un progn implícito

- Tiene la gran ventaja que al terminar de ejecutarse cierra automáticamente el flujo con el fichero correspondiente

Las opciones posibles se especifican a través de argumentos de tipo clave (precedidos de (“:”), se evalúan a sí mismos, **nota:** probar a escribir en el intérprete :prueba)

:direction

Este argumento especifica si el canal o flujo es de entrada (valor :input, es el valor por omisión), de salida (valor :output) o bidireccional (valor :io)

:element-type

Este argumento especifica el tipo de datos intercambiados en la comunicación. Puede ser character (en cuyo caso se utilizarán las funciones read-char y write-char), bit, etc.

:if-exists

Este argumento especifica la acción que deberá realizarse si la dirección de apertura es :output o :io

- Algunos de sus posibles valores son: :new-version (crea una nueva versión del fichero), :overwrite (se modifica destructivamente el contenido), :append (añade al final), :supersede (crea un nuevo fichero que reemplaza al actual)

:if-does-not-exists

Puede tener dos valores :error o :create que indican si se señala un error (por omisión si la :direction es :input) o si se crea un fichero nuevo con el nombre especificado

```
>(with-open-file (mi-flujo “test”
                                :direction :output
                                :if-exists :supersede)
                    (prin1 “primero” mi-flujo))

“primero”

>(with-open-file (mi-flujo “test”
                                :direction :input)
                    (read mi-flujo))

“primero”
```

- Las funciones básicas de lectura en streams son:

  (read [*flujo-entrada* [*eof-error-p* [*eof-value*]]])

  - *flujo-entrada* debe ser un stream del cual se obtienen los datos (si no se especifica nada se supone que provienen de *standard-input*)

  - eof-error-p controla lo que sucede cuando se alcanza el carácter de fin de fichero, el valor por omisión es t y en dicho caso se señala un error. Si su valor es nil la función read devuelve al encontrar dicho carácter el valor indicado en el argumento eof-value

  - La función read-line con los mismos argumentos permite leer el contenido de un fichero línea a línea

```
>(with-open-file (mi-flujo “nuevo-f”
                                :direction :output
                                :if-exists :overwrite
                                :if-does-not-exist :create)
                    (princ “Mi nombre es nuevo-f” mi-flujo)
                    ;; la función terpri añade un carácter de
                    ;; fin de línea al final de un stream
                    (terpri mi-flujo)
                    (princ “esto es otra línea”))

NIL
```

```
>(setq en (open “nuevo-f” :direction :input)
#<File INPUT Stream NUEVO-F>
>(read-line en nil ‘fin)
“Mi nombre es nuevo-f”
NIL
>(read-line en nil ‘fin)
“esto es otra línea”
NIL
>(read-line en nil ‘fin)
FIN
T
>(close en)
NIL
```

- Las funciones básicas de escritura en streams son:

(print *objeto stream*)

- escribe primero un carácter de nueva línea sobre el *stream*, luego el *objeto* de tal forma que pueda ser leído más adelante (por tanto incluye los caracteres de escape)debe ser un stream del cual se obtienen los datos (si no se especifica nada se supone que provienen de *standard-input*) y finalmente escribe un espacio en blanco

(princ *objeto stream*)

- escribe el *objeto* sobre el *stream* sin incluir los caracteres de escape

(prin1 *objeto stream*)

- escribe el *objeto* sobre el *stream* con todos los caracteres de escape

```
> (print "hola")
"hola"
"hola"
> (princ "hola")hola
"hola"
> (prin1 "hola")"hola"
"hola"
```

- Para proporcionar datos con una apariencia (formato) determinado se utiliza la sentencia format

(format *stream-destino string-de-control* &rest *args*)

- Escribe los *argumentos* sobre el *stream-destino* teniendo en cuenta las “directrices” especificadas en el *string-de-control*
- Si el *stream-destino* es nil entonces devuelve el formato de la salida como un string en caso contrario devuelve nil
- Si el *stream-destino* es t la salida se escribe en *standard-output*
- Las “directrices” especificadas en el *string-de-control* se representan precedidas del carácter tilde (“~”). La salida muestra el contenido de dicho string excepto en los lugares donde aparecen dichas directrices
- Existen una gran variedad de posibilidades con format, especificamos sólo algunas de las más

significativas mediante ejemplos concretos (ver el manual de referencia [2])

```
> (format nil "La respuesta es ~D." 5)
"La respuesta es 5."
; ~D escribe un entero decimal
> (format nil "La respuesta es ~3D." 5)
"La respuesta es   5."
; ~3D deja tres espacios antes de escribir
> (format t "&La respuesta es: ~%~D." 5)
La respuesta es
5
; ~& escribe un carácter de nueva línea si no
; estuviera ya en una nueva
; ~% escribe un carácter de nueva línea
```

- Existen dos funciones básicas que permiten cargar el contenido de un fichero en memoria y compilarlo (posteriormente habrá que comprobar si siguen quedando errores de ejecución)

(load *fichero*)

- Esta función lee el contenido de un fichero evaluando todas las sentencias en el contenidas
- El fichero puede ser previamente compilado con la función

(compile-file *fichero* [:output-file *fichero-salida*])

Bibliografía

[1] Winston, P.H., Horn, B.K. (1991). *Lisp*. Editorial Adisson-Wesley Iberoamericana, 3ª Edición (versión traducida).

[2] Guy L. Steele Jr. (1984). *Common Lisp: the language*. Digital Press, Massachusetts (MA).

[3] Guy L. Steele Jr. (1990). *Common Lisp: the language*. 2ª edición. Digital Press, Massachusetts (MA).