Práctica
1

# Introducción al Common Lisp.

La primera sesión de prácticas consiste en una introducción al Common Lisp, empezando por lo más básico: el concepto de CONS. Asimismo se realiza una aproximación al entorno del intérprete que se esté utilizando.

1. **Notación.**
2. **Introducción al lenguaje de programación LISP.**
3. **Concepto de CONS.**
4. **Lisp: lenguaje funcional.**
5. **El intérprete.**
6. **Definición de variables y constantes.**
7. **Sistema de ayudas.**
8. **Evaluación.**



## 1. Notación.

En éste y los siguientes capítulos se utilizará la siguiente notación:

- La línea de comandos del intérprete de Common Lisp se representará por el símbolo "?".
- Las expresiones simbólicas en Lisp irán en letra `courier`: en negrita las expresiones que se escriban para que el intérprete las evalúe y en letra

normal las expresiones que escribe el intérprete como resultado de la evaluación. Por ejemplo:

```
? (LIST 1 2 3 4)
(1 2 3 4)
```

- Las expresiones simbólicas que se nombran en el texto así como las secuencias de expresiones que aparecen sin resultado de la evaluación se pondrán en letra `courier` normal.

# 2. Introducción al lenguaje de programación Lisp.

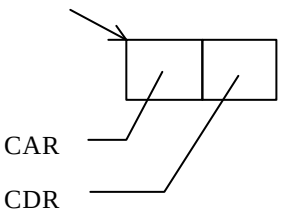
Para aprender Lisp hay que empezar por conocer cuáles son los objetos que podemos manejar:

- Los objetos fundamentales que existen en Lisp son objetos semejantes a una palabra del lenguaje natural y se llaman *átomos*. Por ejemplo: `4, "¡Salud!", AGUACERO, CUANTO?, HORA-DE-COMER.`
- Los grupos de átomos forman objetos (tipo frase) llamados *listas*. Las listas asimismo se pueden agrupar formando listas de nivel superior (listas de listas). Ejemplos de listas simples son `(LA HORA DE COMER)` y `(A B C).` Ejemplos de listas de listas son `((A B C))` y `(CUANTO-VALE? (+ 17 (/ 3456.43 12.25))).`
- Los átomos y las listas conjuntamente se llaman *expresiones simbólicas*, o para abreviar, *expresiones*.

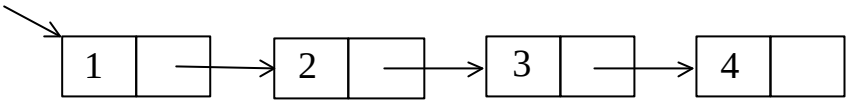
Lisp es un lenguaje capaz de trabajar con expresiones simbólicas (átomos o listas): Todo en Lisp son expresiones simbólicas, desde la definición de funciones hasta el almacenamiento de los datos. Se pueden distinguir diversos tipos de átomos, básicamente: los *números* (como el `4` del ejemplo), los strings literales, (como `"¡Salud!"`) y los *símbolos* (todos los demás del ejemplo). Y dado que un átomo es un elemento tan simple como una palabra o un número, sólo nos queda saber cómo se forman las listas.

# 3. Concepto de CONS.

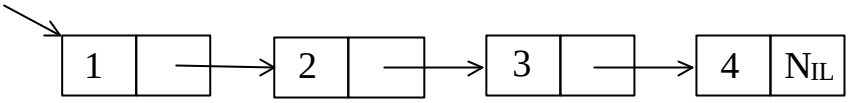
Un CONS es la unidad de almacenamiento básica que consta de dos partes: el CAR y el CDR.




Para organizar una lista simple, necesitamos de al menos dos cosas: los átomos que la forman y algún método de enlace entre ellos. Por ejemplo, una lista como `(1 2 3 4)`, ¿cómo podría organizarse basándose en conses? Muy fácilmente:




Pero, ¿qué poner en el CDR de la última celda CONS para indicar que ahí finaliza la lista? En Lisp existe un átomo especial llamado `NIL` que significa lista vacía. Por tanto, `NIL` es la única expresión simbólica que es a la vez un *átomo* y una *lista*.




Esta es la forma básica de almacenamiento de una lista en Lisp. ¿Como se construyen estos enlaces entre conses con el lenguaje de programación? Con una función del mismo nombre, `CONS` (de `CONS`truir), que admite dos argumentos:

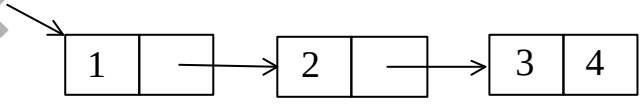
1. lo que se va a poner en el CAR, y
2. lo que se va a poner en el CDR.

La lista anterior se construiría enlazando tantas funciones `CONS` como conses tiene la lista.

```
? (CONS 1 (CONS 2 (CONS 3 (CONS 4 NIL))))
(1 2 3 4)
```

Esta es una definición recursiva de la lista: nótese que los conses no son consecutivos sino anidados. Es algo así como decir que: *salvo la lista vacía (que es `NIL`), una lista está formada por un elemento + una lista.*

Ya conocemos cómo se forma una lista simple, pero aún quedan muchas dudas por despejar. El lector se habrá preguntado: ¿hay necesidad de utilizar 4 conses y un `NIL` en el ejemplo anterior? ¿no se podría utilizar solamente 3 conses, poniendo el último número en el CDR del tercer cons?. Efectivamente, se puede.




Pero también es evidente que aunque esta lista contiene los mismos átomos que la anterior, éstos están dispuestos de diferente manera. Este tipo de listas se llaman *impropias*, mientras que las listas acabadas en ***`NIL`*** como la anterior se llaman listas *propias*. El nombre viene del hecho de que cuando la lista es de más de dos elementos, se suele emplear una lista propia porque es más fácil de manejar debido a que `NIL` marca siempre el final de la lista.

¿Cómo se crean las listas impropias? Igual que las propias:

```
? (CONS 1 (CONS 2 (CONS 3 4)))
(1 2 3 . 4)
```

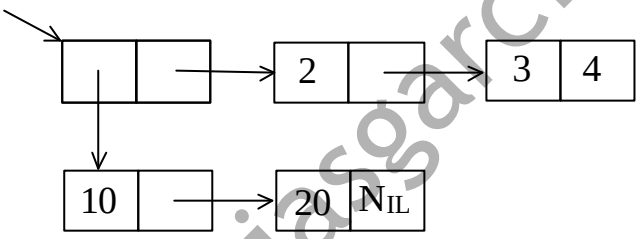
Nótese que de nuevo, hay una función `CONS` por cada celda cons. Como es una lista "diferente" se emplea una notación diferente, de ahí que se señale que la lista es impropia con un punto entre los dos elementos que comparten un mismo cons.

Existe una regla muy sencilla para distinguir una lista propia de una impropia:

- La lista propia tiene tantos conses como elementos.
- La lista impropia de *n* elementos tiene *n*-1 conses.

El punto de la lista impropia debe estar separado de ambos átomos. Por ejemplo, `(1.2)` es una lista propia con un número real, mientras que `(1 . 2)` es una lista impropia con dos números enteros. Con los símbolos ocurre lo mismo, `A.B` es un sólo símbolo y no dos, puesto que el punto se puede emplear en los nombres de los símbolos.

Por último, el lector se preguntará cómo es el hecho de que una lista ¡pueda contener listas!. Para que alguno de los elementos de una lista sea otra lista, simplemente hay que poner la sublista en el CAR del cons adecuado. Por ejemplo:




Es decir, disponemos de dos listas `(2 3 . 4)` y `(10 20)` y queremos añadir la segunda como elemento de la primera. Por tanto la segunda se asigna al CAR y la primera al CDR.

```
? (CONS (CONS 10 (CONS 20 NIL)) (CONS 2 (CONS 3 4)))
((10 20) 2 3 . 4)
```

Nótese que la función `CONS` sirve para añadir un elemento al principio de una lista: según sea el elemento y la lista en cuestión, se crean listas propias, listas impropias o listas de listas. En el siguiente cuadro se resumen todos los casos posibles:

| *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 |

A partir de ahora, una lista no será una sucesión más o menos enrevesada de átomos sino que conocemos perfectamente cómo es por dentro.

***Ejercicio:***

Dibuja los CONSES de las siguientes listas:

```
(MIGUEL DE CERVANTES Y SAAVEDRA)
(+ 2 (* 4 5))
((3 POR 4) SON 12)
((UN . PAR) (DOS . PARES))
((1 . A) (2 (B) C) FIN)
```

# 4. Lisp: lenguaje funcional.

La programación en Lisp se basa en funciones. En el apartado anterior hemos visto la función `CONS`. Una función tiene una lista de argumentos y devuelve un valor. Por ejemplo, la división tiene como argumentos el numerador y el denominador y devuelve el cociente de los mismos. La sintaxis es la siguiente:

```
? (/ 6 3)
2
```

Hay funciones que admiten un número ilimitado de argumentos. Por ejemplo, la suma:

```
? (+ 2 3 4 5)
14
```

Para combinar distintas funciones, las llamadas se anidan de manera que los resultados de unas funciones sirvan como argumentos de otras:

```
? (+ 2 3 (* 2 2) (- 7 2))
14
```

# 5. El intérprete.

El intérprete es un programa que acepta expresiones, las evalúa y escribe el resultado de dicha evaluación. A este bucle de funcionamiento se le llama bucle de lee-evalúa-escribe (*read-eval-print loop*, en inglés). Todas las definiciones que se evalúen son recordadas por el intérprete mientras dure la sesión.

Cuando durante la evaluación se encuentra un error, el Intérprete entra en un bucle de ruptura (*break loop*, en inglés). Se llama así, porque se rompe el funcionamiento normal del intérprete. Para cancelar el error y volver al bucle de funcionamiento normal basta con abortar el bucle de ruptura.

A continuación se introduce el intérprete que se va a utilizar durante las sesiones de prácticas.

### *El intérprete `CLISP`:*

`CLISP` es un intérprete y compilador de Common Lisp de libre distribución. Para iniciar `CLISP` basta con teclear su nombre en la línea de comandos. Además permite especificar el lenguaje en el que se desea que el intérprete escriba los mensajes:

```
$ clisp -L español
```

Así se inicia el bucle lee-evalúa-escribe y el intérprete queda a la espera de que leer expresiones simbólicas. Cuando escribimos una expresión simbólica, el intérprete no hace distinción entre mayúsculas y minúsculas.

```
[1]> (CONS '10 (cons '20 (Cons '30 (cOnS '40 niL))))
(10 20 30 40)
```

Añadir espacios en blanco, tabuladores o retornos de carro no influye en la evaluación de lo que se escribe. Hasta que no se ha terminado de escribir la sentencia por completo, el intérprete no comienza a evaluarla. Por ejemplo:

```
[2]> (+ 1 2 3 4 5
        6 7 8 9 10)
55
```

Cuando al evaluar la expresión ocurre un error, el intérprete indica cuál es, y el nivel de bucle de ruptura:

```
[3]> (+ 2 a)

*** - EVAL: la variable A no tiene ningún valor
1. Break [4]>
```

El bucle de ruptura se aborta tecleando **unwind**.

Aunque las definiciones son recordadas por el intérprete hasta que se finalice la sesión, no quedan registradas en ningún sitio, salvo el propio intérprete. Si queremos tener una copia de aquello que le hemos hecho aprender al Intérprete, *necesitamos escribirlo en un fichero*. Muchos intérpretes modernos incorporan menús para manejo y edición de ficheros entre otras utilidades, pero este intérprete de libre distribución no dispone de facilidades gráficas todavía.

Cargar un fichero Common Lisp, de modo que el intérprete evalúe (y por tanto, recuerde) todas las expresiones simbólicas que contenga, se hace mediante la función `load`.

```
[5]>  (load "/home/inf/lopeza/lisp/ejemplo.lsp")
;; Cargando el fichero /home/inf/lopeza/lisp/ejemplo.lsp ...
;; La carga del fichero /home/inf/lopeza/lisp/ejemplo.lsp ha finalizado
T
```

De este modo, podemos escribir el programa mediante cualquier editor de texto y evaluarlo en el intérprete de diversas maneras.

Para finalizar la sesión, se puede hacer mediante las funciones `(quit)`, o `(exit)`.

```
[6]> (quit)
Adiós.
```

Para más información sobre `clisp`, se puede consultar la documentación disponible, bien en la línea de comandos:

```
$ man clisp
```

o bien en el web que mantiene el Servei d'Informàtica sobre el software instalado en anubis: `http://anubis.uji.es/soft/clisp/`

# 6. Definición de variables y constantes.

### *1) Definición de constantes:*

Para definir constantes en Common Lisp se utiliza `DEFCONSTANT`. Escribe el siguiente ejemplo en el intérprete:

```
? (DEFCONSTANT CERVANTES '(MIGUEL DE CERVANTES Y SAAVEDRA))
CERVANTES
```

La función `DEFCONSTANT` devuelve siempre el nombre de la constante que se acaba de definir. El valor de una constante no se puede modificar. Según el ejemplo, `CERVANTES` es una constante cuyo valor es una lista con el nombre completo del escritor. Es decir, siempre que se evalúe `CERVANTES`, el resultado será su valor:

```
? CERVANTES
(MIGUEL DE CERVANTES Y SAAVEDRA)
? (REVERSE CERVANTES)
(SAAVEDRA Y CERVANTES DE MIGUEL)
```

Obsérvese que el símbolo `CERVANTES` aparece dos veces en la definición: como símbolo dentro de la lista y como nombre de la constante. En realidad, tanto las constantes como las variables no son sino simples símbolos a los que se ha asociado un valor.

### *2) Definición de variables globales:*

Para definir variables globales se utiliza `DEFVAR`. Existe el convenio de definir las variables globales con dos asteriscos para que las expresiones que las utilizan sean más legibles. Escribe el siguiente ejemplo:

```
? (DEFVAR *QUIJOTE* '(DON QUIJOTE DE LA MANCHA))
*QUIJOTE*
? (DEFVAR *LAZARILLO*)
*LAZARILLO*
```

DEFVAR permite definir variables globales dándoles un valor inicial (opcional). DEFVAR devuelve el nombre de la variable que se acaba de definir. Si se define una misma variable más de una vez, sólo tiene efecto la primera definición. Por ejemplo, si se define de nuevo la variable *QUIJOTE* con otro valor inicial, permanece el valor dado la primera vez:

```
? (DEFVAR *QUIJOTE* '(EL LAZARILLO DE TORMES))
*QUIJOTE*
? *QUIJOTE*
(DON QUIJOTE DE LA MANCHA)
```

Si se define una variable, pero no se le da valor, utilizar esa variable dará error porque no tiene un valor asignado. Por ejemplo, evaluar *LAZARILLO* da error porque es una variable que no tiene asociado ningún valor:

```
? *LAZARILLO*

*** - EVAL: la variable *LAZARILLO* no tiene ningún valor
```

### *3) Asignación:*

Para poder asignar o modificar el valor de una variable, se utiliza SETQ (proviene de "*set quote*"):

```
? (setq *lazarillo* '(EL LAZARILLO DE TORMES))
(EL LAZARILLO DE TORMES)
```

La función SETQ siempre devuelve el valor que se ha asignado a la variable. Modifica el valor de *QUIJOTE* y comprueba los resultados de SETQ.

```
? (setq *QUIJOTE* '(mancha la de quijote don))
(MANCHA LA DE QUIJOTE DON)
? *quijote*
(MANCHA LA DE QUIJOTE DON)
? *lazarillo*
(EL LAZARILLO DE TORMES)
```

También puedes comprobar que el valor de una constante no se puede modificar.

```
? (setq cervantes '(el manco de lepanto))

*** - SETQ: no puede alterarse el valor de la constante CERVANTES
```

### *4) Ejercicio:*

Escribe en el intérprete la sentencia en negrita.

```
? (defvar saludos '(HOLA BUENOS-DIAS BUENAS-TARDES))
SALUDOS
```

A partir de este momento, siempre que nosotros empleemos el símbolo SALUDOS en el Intérprete, entenderá que nos referimos a la lista que le hemos asociado. Por ejemplo:

```
? saludos
(HOLA BUENOS-DIAS BUENAS-TARDES)
```

Para aprender nuevas funciones en Lisp, prueba los siguientes ejemplos y averigua qué devuelven. A la derecha está la explicación de lo que hará el intérprete:

```
(CAR SALUDOS)                 ;Devuelve el primer elemento de la lista
                              ;SALUDOS
(FIRST SALUDOS)               ;FIRST significa lo mismo que CAR

(CDR SALUDOS)                 ;Devuelve la lista que queda de eliminar
                              ;el primer elemento
(REST SALUDOS)                ;REST significa lo mismo que CDR

(CONS 'ADIOS SALUDOS)         ;Añade un elemento a la lista:
                              ;(ADIOS HOLA BUENOS-DIAS ... )
SALUDOS                       ;Está exactamente igual que antes

(CAR (CONS 'ADIOS SALUDOS))   ;El primero de la lista resultante del
                              ;cons: ADIOS
(SETQ SALUDOS (CONS 'ADIOS SALUDOS))  ;Añade a la lista SALUDOS el
                                      ;elemento ADIOS
SALUDOS                         ;SALUDOS ha sido modificada

(CAR SALUDOS)                   ;y ADIOS es el primero de la lista

(SETQ SALUDOS (REST SALUDOS))   ;Quita el primer elemento de SALUDOS
```

## 7. Sistema de ayuda.

En casi todos los compiladores de Lisp existen algunas ayudas, desde los típicos comandos de cortar y pegar hasta otras ayudas más específicas de lisp como es la correspondencia de los paréntesis o ayudas en la identación:

- Generalmente al escribir un cierre de paréntesis, alguna señal nos indica cuál es el paréntesis de apertura que le corresponde, por ejemplo, un parpadeo.
- Suelen haber comandos que permiten obtener las expresiones escritas anteriormente, para facilitar modificarlas y evaluarlas con mayor facilidad.
- A menudo existen comandos que permiten identar de forma coherente las expresiones simbólicas o detectar paréntesis mal colocados.

# 8. Evaluación.

Copia el ejemplo `"/home/inf/lopeza/lisp/ejemplo.lsp"` en tu cuenta, y ábrelo con un editor. Carga el fichero con la función `load` y trata de averiguar qué operación se realiza en cada paso.

```
? (load "/home/inf/lopeza/lisp/ejemplo.lsp")
```

Una vez evaluado el contenido del fichero, comprueba lo siguiente:

- Todas las expresiones han sido evaluadas y memorizadas por el intérprete. Se puede comprobar que todas las variables siguen definidas, consultando su valor en el intérprete directamente. Por ejemplo, las variables `x`, `y` y `z` contienen todavía las medidas del paralelepípedo.

```
? x
3.5
```

- Las variables `*area*` y `*volumen*` adquirirán el área del perímetro y el volumen, respectivamente, del paralelepípedo cuyas medidas se asignan a `x`, y y `z`.

```
? *area*
102
? *volumen*
56
```

- Si, por ejemplo, cambiamos el valor de las coordenadas, para recalcular el valor de todas las variables en el intérprete es necesario volver a evaluar todas las funciones `setq`. Para ello, se puede modificar el fichero, salvarlo y cargarlo nuevamente. Compruébese el nuevo valor de las variables `*area*` y `*volumen*` en el intérprete.

- También se puede utilizar las funciones de cortar y pegar para evaluar solamente algunas expresiones del fichero. Por ejemplo, de esta forma se puede recalcular el valor de cada variable separadamente.

### *Primera aproximación a la regla de evaluación*

Como ya se habrá podido observar en el ejemplo, hay dos reglas básicas a la hora de evaluar una expresión simbólica:

- Si lo que se quiere evaluar es una variable, ésta *no* puede ir precedida de paréntesis.

- Si lo que se quiere evaluar es una función, ésta *debe* ir precedida de paréntesis.

Compruébese observando las expresiones evaluadas hasta el momento.

Por otro lado, ¿qué diferencia hay entre el nombre `X`, `Y` y `Z` de las medidas numéricas y los nombres `X`, `Y` y `Z` del sistema de coordenadas? Haz la siguiente prueba en el intérprete:

```
(LIST X Y Z)     ; El intérprete evalúa X (=3.5), evalúa Y (=8)
                 ; y evalúa Z (=2) antes de construir la lista.
(LIST 'X 'Y 'Z)  ; El intérprete toma X, Y y Z literalmente,
```

```
                    ; son átomos, y después construye la lista.
```

Otro ejemplo:

```
(+ X Y Z)      ; Es posible hacer esta suma ya que X, Y y Z son
               ; variables, y además su contenido es un número.
(+ 'X 'Y 'Z)   ; No es posible hacer la suma ya que X, Y y Z son
               ; átomos, no números.
```

En esta última evaluación, el intérprete no puede realizar la suma de los tres átomos, ya que dichos átomos no son números, y por tanto dará un error.

Práctica
2

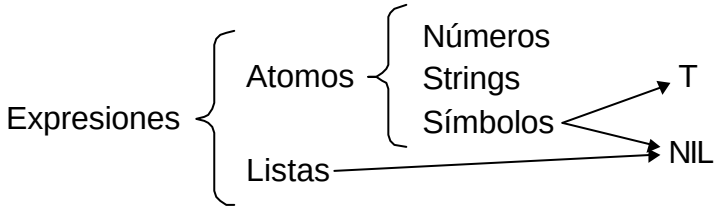
# La regla de evaluación. Definición de funciones.

Se sientan las bases fundamentales de la programación en Lisp. El principal objetivo de esta práctica es estudiar la regla de evaluación, incorporando las funciones básicas relacionadas con la misma. Al mismo tiempo se insiste en el concepto de cons, debido a su importancia, y se estudian las diferencias entre las funciones básicas de creación y manipulación de listas. Por último, se aprende a definir nuevas funciones.

1. **Conses, Atomos y Listas.**
2. **La regla de evaluación.**
3. **Quote y Eval.**
4. **Funciones definidas por el usuario.**
5. **Ejercicios.**

## 1. Conses, Atomos y Listas.

Todos los objetos Lisp son expresiones simbólicas:




Existen dos símbolos con significado especial: T significa “verdadero”, y NIL significa “falso”. Existen ciertas funciones en Lisp denominadas predicados que

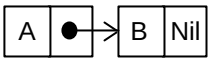
responden a ciertas preguntas con T o NIL. Por otro lado, NIL es el único átomo que al mismo tiempo es una lista, la "lista vacía".

Una celda cons, se puede construir con la sentencia CONS:

```
? (CONS 'A 'NIL)
(A)
```


```
? (CONS 'A 'B)
(A . B)
```


```
? (CONS 'A '(B))
(A B)
```




Dibuja en un papel las siguientes combinaciones de celdas cons:

```
(CONS 'A (CONS 'B (CONS 'C 'NIL)))

(CONS (CONS 'UNO 'NIL)
      (CONS (CONS 'DOS (CONS 'Y (CONS 'TRES 'NIL)))
            (CONS 'CUATRO 'NIL)))

(CONS 'A (CONS 'B (CONS 'C 'D)))

(CONS (CONS 'A 'B) (CONS 'D 'NIL))
```

La sentencia LIST se utiliza para crear *listas propias*. Las *listas impropias* sólo pueden crearse a partir de la función CONS. Veamos los mismos ejemplos anteriores, ¿encuentras alguno cuyo resultado no sea igual? ¿por qué?:

```
(LIST 'A 'B 'C)

(LIST (LIST 'UNO) (LIST 'DOS 'Y 'TRES) 'CUATRO)

(LIST 'A 'B (CONS 'C 'D))

(LIST (CONS 'A 'B) 'D)
```

LIST acepta un número indefinido de listas. Más ejemplos:

```
(LIST 'ALGO 'NIL)

(LIST)
```

APPEND 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:

```
(APPEND '(A B C) '(D E F) '(G) '(H I))

(APPEND '(A B C) '() '(F))

(APPEND '(A B C) 'D)
```

***Ejercicio:***

Estudia las diferencias entre CONS, LIST y APPEND a partir de las expresiones siguientes:

```
? (DEFVAR ab-list '(A B))
? (DEFVAR cd-list '(C D))

? (APPEND ab-list cd-list)
? (LIST ab-list cd-list)
? (CONS ab-list cd-list)

? (APPEND ab-list ab-list)
? (LIST ab-list ab-list)
? (CONS ab-list ab-list)

? (APPEND ab-list 'C)
? (LIST ab-list 'C)
? (CONS ab-list 'C)

? (APPEND 'C ab-list)
? (LIST 'C ab-list)
? (CONS 'C ab-list)
```

## 2. La regla de evaluación.

A menudo nos interesará conocer el valor de una expresión. Dicho valor es calculado automáticamente por el intérprete por medio de un proceso denominado evaluación. A continuación, se detalla una versión simplificada (por ejemplo, no se tiene en cuenta las macros) de la evaluación de una expresión:

1. Si la expresión es un *átomo*, el resultado de su evaluación depende de si es un número o un símbolo.
   - Si es un *número*, el resultado de su evaluación es él mismo.
   - Si es un *símbolo*, el intérprete busca su valor asociado (es decir, lo trata como si fuera una variable y devuelve su contenido). Si el símbolo no tiene ningún valor asociado, su evaluación dará error.
2. Si es una *lista*, el primer elemento de la lista es interpretado como la función a aplicar para obtener su valor, y el resto de elementos son los argumentos que se le pasan a dicha función.
   - Si la función no está definida, su evaluación dará error.
   - Si está definida, primero se evalúan todos los argumentos y después se realiza la llamada a la función con los resultados de evaluar los argumentos.

Dadas las siguientes variables definidas:

```
? (DEFVAR MI-ATOMO 'PELIGRO)
? (DEFVAR MI-LISTA '(1 2 3))
```

el intérprete es capaz de evaluar los siguientes símbolos y predicados, porque tienen un valor asociado:

```
? MI-ATOMO
? MI-LISTA
? (LISTP MI-ATOMO)
? (LISTP MI-LISTA)
```

Sin embargo, los siguientes ejemplos son casos de error. Intenta averiguar por qué:

```
? PELIGRO
? (MI-ATOMO MI-LISTA)
? (MI-LISTA)
```

## 3. Quote y Eval.

QUOTE es una función especial de Lisp que hace que se detenga el proceso de evaluación. Por ejemplo, aplicado sobre una lista, hace que no se considere como una función a evaluar (por ejemplo en el caso de la lista `'(1 2 3)`, donde el 1 no debe ser considerado como una función). Otro ejemplo es el átomo `'PELIGRO`, que de no haberle puesto un QUOTE, el Intérprete hubiera considerado como una variable y hubiera buscado su valor para asignárselo a la variable `MI-ATOMO`. `(QUOTE A)` es equivalente a `'A`. Se suele emplear la forma abreviada para simplificar.

Averigua qué resultados dan las siguientes expresiones y por qué:

```
''A
'(A B 'C D)
```

Como algunos átomos al evaluarlos dan como resultado ellos mismos, en ellos el uso de QUOTE no es necesario, pero tampoco es incorrecto:

```
'1994                    (QUOTE "Buenas tardes")
1994                     "Buenas tardes"
'NIL                     (QUOTE T)
NIL                      T
```

EVAL es el propio evaluador de Lisp, que se puede utilizar directamente. Evalúa según la regla de evaluación normal:

```
? (setq mi-lista '(list 'A 'B))
(LIST 'A 'B)
? (EVAL mi-lista)
(A B)

? (setq mi-lista '(list 'NUMBERP 4))
(LIST 'NUMBERP 4)
? (EVAL (EVAL mi-lista))
T
```

# 4. Funciones definidas por el usuario.

Los usuarios pueden definir sus propias funciones cuyo uso es sintácticamente indistinguible de las funciones predefinidas. Para ello se utiliza la forma DEFUN (de *define function*), que tiene la estructura siguiente:

```
(DEFUN <nombre> <lista de parametros> <documentacion> <cuerpo>)
```

La documentación es opcional, aunque ya veremos más adelante que es sumamente útil.

Un ejemplo de definición de función:

```
? (defun cuadrado (x) (* x x))
CUADRADO
? (cuadrado 5)
25
```

En la lista de argumentos se pueden poner palabras clave que tienen significados especiales. De momento, sólo vamos usar la palabra clave `&OPTIONAL` que significa que todos los argumentos a continuación de ella son opcionales. Por ejemplo:

```
? (defun potencia (x &optional (n 2))
    (if (= n 0)
        1
        (* x (potencia x (- n 1)))))
POTENCIA
? (potencia 5)
25
? (potencia 5 0)
1
```

Se puede especificar el nombre del argumento y su valor inicial en una lista, como está en el ejemplo, o se puede poner solamente el nombre del argumento (sin paréntesis) en cuyo caso el valor por defecto es NIL.

### *Ejemplo.*

Siguiendo con el ejemplo de la práctica anterior, que calcula el volumen y el área de un paralelepípedo, en el fichero "`ejemplo_fun.lsp`" se encuentra una solución realizada con funciones.

```
? (load "/home/inf/lopeza/lisp/ejemplo_fun.lsp")
```

Esta solución tiene la ventaja de que además de solucionar el problema propuesto, define todas las funciones necesarias para ello, de modo que éstas se pueden volver a utilizar con distintos parámetros tantas veces como se desee.

Por ejemplo, prueba a calcular el área y volumen de otro paralelepípedo usando la siguiente función:

```
? (area_y_volumen 18 23.5 16)
```

Lee el fichero "ejemplo_fun.lsp" y busca otras funciones que puedas usar. Por ejemplo, para calcular el volumen de un paralelepípedo:

```
? (volumen 18 23.5 16)
6768.0
```

# 5. Ejercicios.

## *5.1. Quote y Eval.*

1. ¿Qué expresión simbólica hace falta para obtener el siguiente resultado sin utilizar el 3?

```
? (DEFVAR NUM 3)
NUM
? ...
(3 (3 3) (NUM 3) NUM)
? ...
(NUM NIL (3 3))
```

2. ¿Qué expresión simbólica hace falta para obtener el siguiente resultado sin utilizar NUM-MESAS?

```
? (DEFVAR CANTIDAD 'NUM-MESAS)
CANTIDAD
? (DEFVAR NUM-MESAS 3)
NUM-MESAS
? ...
3
```

3. ¿Qué expresión simbólica hace falta para obtener el siguiente resultado a partir de CANTIDAD y NUM-MESAS?

```
? (SETQ CANTIDAD (LIST '+ 5 6))
(+ 5 6)
? (SETQ NUM-MESAS '(ES EL NUMERO DE MESAS))
(ES EL NUMERO DE MESAS)
? ...
(11 ES EL NUMERO DE MESAS)
```

4. ¿Qué expresión simbólica hace falta para obtener el siguiente resultado a partir de RESTA?

```
? (DEFVAR RESTA '(- 4 1))
RESTA
? ...
((- 4 1) ES IGUAL A 3)
```

## *5.2. Definición de funciones*

5. Define una función que dada una lista cualquiera, devuelva el tercer elemento de la lista. Por ejemplo, dadas dos listas definidas como sigue:

```
? (DEFVAR TRIPLETES '((a b c) (1 2 3) (p q r) (T T NIL)))
TRIPLETES
? (DEFVAR LETRAS '(a b c d e f g h))
LETRAS
```

Si llamamos TERCERO a la función que calcula el tercer elemento de la lista, ésta debería funcionar así:

```
? (TERCERO TRIPLETES)
(P Q R)
? (TERCERO LETRAS)
C
```

6. Define una función que dada una lista cualquiera, devuelva el cuarto elemento de la lista, utilizando la función del ejercicio anterior.

```
? (CUARTO TRIPLETES)
  (T T NIL)
? (CUARTO LETRAS)
  D
```

7. Define las siguientes funciones, teniendo en cuenta que si se les pasara la lista TRIPLETES como argumento, los resultados serían los que van a continuación:

   a) Una función que dada una lista de sublistas, devuelva el primer elemento de la segunda sublista.

```
? (ELEM-2-1 TRIPLETES)
1
```

   b) Una función que dada una lista de sublistas, devuelva el tercer elemento de la primera sublista.

```
? (ELEM-1-3 TRIPLETES)
C
```

8. Evalúa las siguientes expresiones simbólicas y averigua la respuesta a las siguientes cuestiones:

```
(append '(a b c) '())
(list '(a b c) '())
(cons '(a b c) '())
(cons (first nil) (rest nil))
```

   – La lista ( ) equivale a ...

   – `Append` de una lista con NIL devuelve ...

   – El `first` y el `rest` de una lista vacía es ...

9. Evalúa las siguientes expresiones simbólicas en este orden para ver las diferencias de PUSH y POP con FIRST y REST:

```
(defvar lista-num '(uno dos tres cuatro))
(push 'cero lista-num)
lista-num
(pop lista-num)
lista-num
```

10. Después de averiguar qué hace NTHCDR evaluando las siguientes expresiones simbólicas, contesta: ¿a qué equivale (nthcdr 1 ...)?

```
(defvar numeros '((1 uno) (2 dos) (3 tres) (4 cuatro)))
(nthcdr 2 numeros)
(nthcdr 3 numeros)
(nthcdr 4 numeros)
```

11. Después de averiguar para qué sirven LAST y BUTLAST evaluando las siguientes expresiones simbólicas, escribe una función que devuelva el último elemento de una lista.

```
(butlast numeros 2)
(butlast numeros 25)
(butlast numeros)
(last numeros)
```

12. Averigua para qué sirven las funciones LENGTH y REVERSE evaluando las siguientes expresiones simbólicas:

```
(length numeros)
(length (first numeros))
(reverse numeros)
```

13. Examina la siguiente función y averigua la sintaxis del condicional, IF, evaluando las siguientes expresiones simbólicas:

```
(defun resta-positiva (n1 n2)
   (if (<= n1 n2)
      0
      (- n1 n2)))
(resta-positiva 2 7)
(resta-positiva 7 2)
```

14. Escribe una función que haga lo mismo que `LAST`, con dos argumentos: una lista (argumento obligatorio) y el número de elementos que deben quedar en la lista resultante (argumento opcional, que por defecto es 1). Téngase en cuenta las funciones de los ejercicios anteriores y `nthcdr`.

15. Define las siguientes funciones, utilizando lo aprendido en el ejercicio 13:

    a) Una función que dándole como parámetros 2 números, devuelva el número más pequeño.

    b) Una función que dados 3 números, devuelva el más grande.

Práctica
3

# Depuración de funciones. Operaciones sobre listas.

Para poder definir funciones más complejas vamos a necesitar, en primer lugar, ser capaces de hacer un seguimiento de la evaluación, mediante funciones de traza que faciliten la depuración de las mismas. En segundo lugar, estudiaremos el manejo de las listas desde distintos puntos de vista, conociendo las funciones más importantes en cada caso. Completaremos el abanico de funciones básicas mediante la revisión de los predicados más utilizados.

1. **Utilidades de Seguimiento y Depuración.**
2. **Listas de asociación.**
3. **Listas como conjuntos.**
4. **Predicados.**
5. **Ejercicios.**



## 1. Utilidades de Seguimiento y Depuración.

El lenguaje Common Lisp incorpora distintas formas de examinar y depurar los programas. Por ejemplo, para averiguar la memoria que queda disponible en un momento dado (y en ocasiones, según el intérprete, otra información adicional), se puede utilizar la función ROOM:

```
? (room)
133,536 bytes free
```

A continuación se muestran otras utilidades que incorpora el lenguaje Common Lisp.

### *Búsqueda y documentación de funciones:*

Sigue los siguientes pasos:

1) Abre un nuevo fichero y escribe en él la siguiente definición de función:

```
(defun factorial (numero)
"Devuelve el factorial de un número."
   (if   (zerop numero)
         1
         (* numero (factorial (- numero 1)))))
```

2) Salva el fichero con un nombre a elegir y evalúa el contenido del fichero.

3) Escribe ahora en el Intérprete las siguientes sentencias y determina para qué sirve cada una:

```
(apropos "factor")

(documentation 'factorial 'function)
```

4) Ahora realiza las mismas operaciones del punto 3 con la siguiente definición de una variable, sabiendo que en `documentation` hay que especificar que es una variable:

```
(defvar centenario '1992
        "Quinto centenario del descubrimiento de América")
```

### *El bucle de ruptura:*

Un bucle de ruptura aparece siempre que ocurre un error, o con una llamada explícita a la función `break`. Se llama así porque “se ha roto” el bucle normal de funcionamiento: el *bucle de lectura-evaluación-escritura*. Un bucle de ruptura es a su vez un bucle de lectura-evaluación-escritura, y puede tener a su vez bucles de ruptura. Por eso aparece una indicación del nivel de anidamiento de los bucles de ruptura.

Un bucle de ruptura permite interactuar con el programa que se está ejecutando, y después continuar la ejecución donde se quedó. La función que permite continuar la ejecución se llama CONTINUE.

Cada nuevo bucle de ruptura añade una nueva área a la pila. Véase el esquema de la figura 2.1. para entender el funcionamiento y las diferencias entre BREAK, ABORT y CONTINUE.

### *Alto nivel de abstracción:*

Como ejemplo del nivel de abstracción que soporta el lenguaje, prueba a calcular el siguiente factorial:

```
? (factorial 300)
```

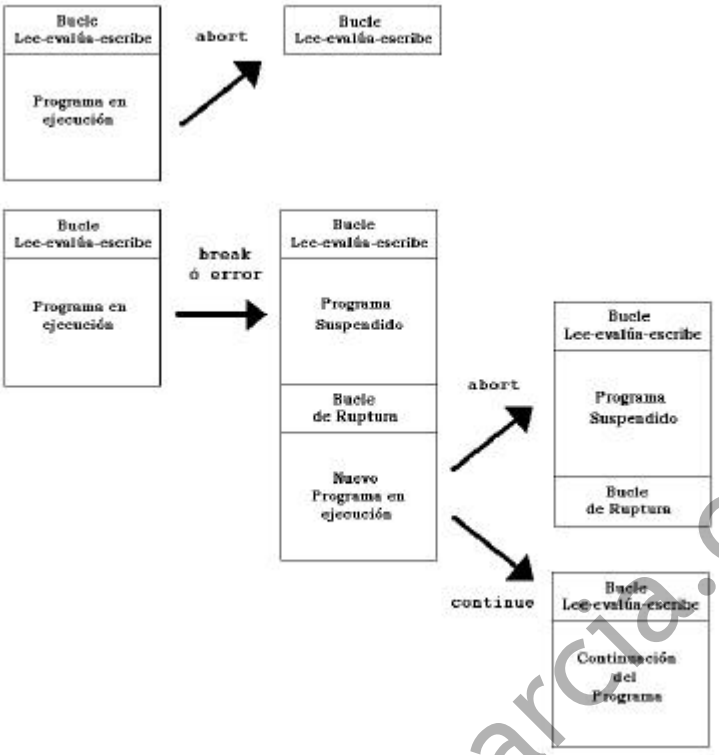



**Figura 2.1.** Bucle de funcionamiento normal y bucles de ruptura

El resultado es un número entero de casi 700 dígitos. En otros lenguajes es necesario utilizar números reales para realizar este cálculo, o crear una librería especial de números extralargos. Sin embargo, el intérprete de Common Lisp maneja números cortos y largos indistintamente, según las necesidades del programa, sin que el programador tenga que preocuparse de indicárselo.

### *Step:*

Para ver la evaluación de cualquier expresión paso a paso, se utiliza STEP:

```
? (step (factorial 3))
```

Se ejecuta una sola evaluación a cada paso y se muestra el resultado. Cada paso se realiza escribiendo `step`.

### *Trace y Untrace:*

Una función `trace` permite averiguar por qué una función no se comporta como se esperaba, quizá debido al valor de sus parámetros.

- `trace` permite imprimir información cuando se inicia y termina una llamada a función. Se imprime el nombre de la función y sus argumentos cuando se llama y el resultado cuando termina.
- `untrace` deshabilita la función `trace` para los argumentos dados.

Para saber qué operaciones tienen habilitado `trace` se usa `(trace)`, y para deshabilitar `trace` para todas las operaciones, `(untrace)`.

Evalúa las siguientes expresiones y observa los resultados:

```
? (defun suma-factorial (num)
     (if (zerop num)
       1
       (+ (factorial num) (suma-factorial (- num 1)))))
? (trace factorial)
? (trace suma-factorial)
? (trace)
? (factorial 5)
? (suma-factorial 3)
? (untrace factorial)
? (suma-factorial 3)
? (untrace)
? (suma-factorial 3)
```

## 2. Listas de asociación.

Las listas pueden utilizarse como asociaciones de claves y datos, recibiendo el nombre de *listas de asociación*:

```
? (defvar ALFA-ROMEO
      '((cilindrada . 3) (potencia . 180) (color . rojo)))
ALFA-ROMEO
? (defvar RENAULT19
      '((cilindrada . 1.8) (color . blanco) (color . rojo)))
RENAULT19
```

En este ejemplo, ambas variables son listas de asociación donde las claves son: cilindrada, potencia y color. ASSOC es una forma de acceder a las sublistas mediante el campo clave.

```
? (assoc 'color ALFA-ROMEO)
? (assoc 'color RENAULT19)
? (assoc 'traccion ALFA-ROMEO)
```

Cuando existe más de un campo con una misma clave, ASSOC siempre encuentra el primer campo que aparece y nunca los siguientes. Por tanto, para "modificar" un dato de una lista de asociación, sería suficiente con añadir en nuevo dato con su clave correspondiente al principio de la lista. Esto inutiliza el dato antiguo.

Para construir listas de asociación, o añadir elementos a una lista ya existente se puede emplear la función PAIRLIS. El tercer argumento es opcional y es el que determina si se trata de crear una lista de asociación nueva o de añadir elementos a una ya existente:

```
? (defvar TWINGO (PAIRLIS '(cilindrada potencia) '(1.1 40)))
? (setq TWINGO (PAIRLIS '(color vel-maxima) '(mostaza 150)
                        TWINGO))
```

# 3. Listas como conjuntos.

Una lista sin elementos repetidos puede considerarse como un conjunto. Una de las operaciones para manejar conjuntos es MEMBER, que verifica que su primer argumento sea un elemento de su segundo argumento, y devuelve el resto de la lista que comienza en el símbolo coincidente, o NIL si no lo encuentra.

```
? (defvar frase '(esto es otra cosa))
? (MEMBER 'pepino frase)
? (MEMBER 'otra frase)
? (MEMBER 'otra (list frase frase))
```

Existen otras operaciones que tratan la lista como un conjunto de elementos, como son las funciones `INTERSECTION`, `UNION`, etc.

# 4. Predicados.

NULL y ENDP devuelven T si el argumento es una lista vacía, con la diferencia de que ENDP presupone que el argumento es una lista, y NULL admite cualquier átomo o lista.

```
? (null 'simbolo)
? (endp 'simbolo)
```

### *Predicados de igualdad*

Predicados de comparación entre átomos y listas: Veamos la diferencia entre los distintos predicados para comprobar la igualdad entre argumentos.

| *Nombre* | *Propósito* |
|---|---|
| EQ | ¿Los dos argumentos son el mismo objeto físicamente?<br>(¿Se refieren a la misma dirección de memoria?) |
| EQL | ¿Los dos argumentos son el mismo objeto conceptualmente? |
| EQUAL | ¿Los dos argumentos tienen el mismo “aspecto”?<br>(¿su representación simbólica es la misma?) |
| = | ¿El valor de los dos argumentos es el mismo número? |

Supongamos que se evalúan las siguientes expresiones:

```
? (setq R (/ 1.0 3.0))
? (setq lis '(a b))
? (setq X 'a)
```

Si se trata de números, = tiene distinto significado que los demás:

```
? (= 3.0 3)    ? (EQL 3.0 3)     ? (EQUAL 3.0 3)
? (= 3 3)      ? (EQL 3 3)       ? (EQUAL 3 3)
```

Si se trata de átomos la función = daría error, y el resto son equivalentes:

```
? (EQ 'a X)          ? (EQL 'a X)          ? (EQUAL 'a X)
? (EQ X (first lis)) ? (EQL X (first lis)) ? (EQUAL X (first lis))
```

Si se trata de listas, = daría error, y EQUAL tiene distinto significado que EQ y EQL, que son equivalentes:

```
? (EQ lis lis)    ? (EQL lis lis)    ? (EQUAL lis lis)
? (EQ lis '(a b)) ? (EQL lis '(a b)) ? (EQUAL lis '(a b))
```

Pero ¿qué diferencia hay entre EQ y EQL? En el caso de los números, EQUAL es idéntico a EQL, pero ambos son distintos de EQ. EQ no debería utilizarse con números porque es dependiente de la implementación. Veamos un ejemplo:

```
? (EQ 3.2 3.2)           ? (EQL 3.2 3.2)
? (EQ R (/ 1.0 3.0))     ? (EQL R (/ 1.0 3.0))
```

Resumiendo, las funciones de igualdad de objetos Lisp se pueden usar para:

| | **Atomos** | | **Listas** |
|---|---|---|---|
| | **Números** | **Símbolos** | |
| **=** | ¿Mismo número? | No | No |
| **EQ** | ¿Mismo objeto? | ¿Es el mismo símbolo? | ¿Es el mismo objeto físicamente? |
| **EQL** | ¿Mismo tipo y mismo número? | | |
| **EQUAL** | | | ¿Tiene el mismo aspecto? |

### *Otros predicados*

Aunque algunos ya se han visto anteriormente, he aquí una lista de los predicados más comunes.

| *Nombre* | *Propósito* |
|---|---|
| ATOM | ¿Es el argumento un átomo? |
| CONSP | ¿Es el argumento un cons? |
| LISTP | ¿Es el argumento una lista? |
| NUMBERP | ¿Es el argumento un número? |
| ZEROP | ¿Es el argumento el número 0? |
| SYMBOLP | ¿Es el argumento un símbolo? |
| NULL | ¿Es el argumento NIL? |
| ENDP | ¿Es el argumento NIL? |
| NOT | Negación lógica |

# 5. Ejercicios.

Escribe las siguientes funciones en un fichero de forma que puedas aprovecharlo en las siguientes prácticas.

1. Define una función que determine si un número es divisible por 3, utilizando la función REM, que dados dos parámetros enteros, devuelve el resto de la división entera.

2. Define una función con 3 argumentos, donde el primer argumento es una lista y los siguientes son dos índices de la lista, que borre los elementos situados entre los dos exclusive. Se pueden usar las funciones ya conocidas e incluso se recomienda hacer *trace* de `butlast` y `nthcdr`.

```
? (borrar '(a b c d e f) 2 5)   ;borra entre tercero y cuarto
(a b e f)
? (borrar '(a b c d e f) 4 2)   ;no borra nada
(a b c d e f)
? (borrar '(a b c d e f) 0 25)  ;borra toda la lista
NIL
```

3. Elabora una función que dados dos meses, diga si el primero aparece antes que el segundo en el calendario.

```
? (mesanterior 'ENERO 'DICIEMBRE)
T
? (mesanterior 'ENERO 'ENERO)
NIL
```

4. Elabora una función que acepte dos parámetros que son dos fechas y devuelva la fecha que sea anterior. Un ejemplo:

```
? (defvar descubrimiento '(12 OCTUBRE 1492))
? (defvar bisiesto '(29 FEBRERO 1992))
? (fecha-anterior descubrimiento bisiesto)
(12 OCTUBRE 1492)
? (fecha-anterior bisiesto descubrimiento)
(12 OCTUBRE 1492)
```

5. Define las siguientes variables y funciones:

   a) Una variable global `*ordenador-ideal*` que contenga una lista de asociación con las caracteristicas de vuestro ordenador ideal. En aquellas características que se definan en base a una medida, hay que especificar el tipo de medida, por ejemplo: 8 Mb, 640 Kb, 100 Mhz, etc.

   b) Una variable global `*mi-ordenador*` que contenga una lista de asociación similar sobre vuestro ordenador (o de un amigo).

   c) Una función que dado un ordenador (en forma de lista de asociación como los anteriores) devuelva el tipo de procesador que tiene.

   d) Una función que dado un ordenador devuelva la cantidad de memoria RAM que tiene.

   e) Otras tantas funciones como las anteriores que devuelvan el resto de características del ordenador (disco duro, tarjeta gráfica, tarjeta de sonido, etc).

f) Una función que devuelva si un ordenador es igual que el ordenador ideal. (Téngase en cuenta que el orden de las claves en ambas listas de asociación puede ser diferente.)

g) Una función (utilizando las dos anteriores) que dados dos ordenadores devuelva el mejor, siguiendo el siguiente criterio: el mejor ordenador es el que tiene mejor procesador, y en caso de que tengan procesador idéntico, el que tenga más memoria RAM.

6. Determina el número máximo de llamadas recursivas que se pueden hacer con la función factorial definida en esta práctica haciendo pruebas hasta producir un error.

Práctica
4

# Estructuras de control. Funciones destructivas.

Para concluir con el aprendizaje de las funciones básicas, se estudian las estructuras de control más importantes, como son las formas condicionales, la definición de variables locales, estructuras de iteración, etc. En segundo lugar se introduce el concepto de función *destructiva / no destructiva*, cuál es la diferencia y sus ventajas e inconvenientes. Por último, se estudian las funciones básicas de copia de listas.





## 1. Estructuras de control condicionales.

La estructura **IF** tiene la forma

```
(IF <condicion> <forma-then> <forma-else>)
```

Las formas **WHEN** y **UNLESS** son variaciones de la forma IF. WHEN equivale a un IF sin parte ELSE y UNLESS equivale a un IF sin parte THEN. Evalúa el siguiente ejemplo:

```
? (SETQ X ...)
? (WHEN (EQL 3 X) 'ENTONCES)
? (UNLESS (EQL 3 X) 'SINO)
```

Si X tiene valor 3 el resultado del WHEN será `ENTONCES`, y el del UNLESS `NIL`. Si no vale 3, el resultado del WHEN será `NIL` y el del UNLESS será `SINO`.

La estructura **COND** evalúa las condiciones hasta que encuentra una que se cumpla, y entonces evalúa la forma asociada a ésta. Si ninguna es verdadera, devuelve NIL. Normalmente se utiliza como última condición T para realizar una acción en el caso de que ninguna otra condición se cumpla. Si alguna condición no tiene ninguna forma asociada se devuelve el valor de la condición.

La estructura **CASE** evalúa una expresión y compara el resultado con los posibles valores sin evaluar; y si encuentra alguno coincidente evalúa la forma asociada. Si el último valor es T u OTHERWISE y ninguno de los anteriores ha coincidido, se evalúa la última cláusula. En el caso de que los valores tengan la forma de listas, CASE usa como criterio la función MEMBER.

En la siguiente tabla se puede ver la equivalencia aproximada de todas las formas anteriores con la forma IF:

| *Estructura de control* | *Estructura IF equivalente* |
|---|---|
| `(WHEN condición`<br>`  forma-then)` | `(IF condición`<br>`  forma-then`<br>`  NIL)` |
| `(UNLESS condición`<br>`  forma-else)` | `(IF condición`<br>`  NIL`<br>`  forma-else)` |
| `(COND (condición1 forma1)`<br>`  (condición2 forma2)`<br>`  ...`<br>`  (condiciónN formaN))` | `(IF condicion1`<br>`  forma1`<br>`  (IF condición2`<br>`  forma2`<br>`  ...`<br>`  (IF condiciónN`<br>`  formaN`<br>`  NIL)...))` |
| `(CASE forma`<br>`  (valor1 forma1)`<br>`  (valor2 forma2)`<br>`  ...`<br>`  (valorN formaN))` | `(IF (EQL forma valor1)`<br>`  forma1`<br>`  (IF (EQL forma valor2)`<br>`  forma2`<br>`  ...`<br>`  (IF (EQL forma valorN)`<br>`  formaN`<br>`  NIL)...))` |

## 2. AND y OR.

Para utilizar las formas de control condicionales es necesario escribir expresiones lógicas complejas. **AND** y **OR** se utilizan, respectivamente, para la

conjunción y disyunción lógica de condiciones. Ambas admiten un número indeterminado de argumentos.

AND evalúa una a una las formas que se pasan como argumento y si alguna de ellas devuelve NIL, se detiene y devuelve NIL. Si todas devuelven valores distintos de NIL, devuelve el valor de la última forma.

OR evalúa una a una las formas y si alguna devuelve un valor distinto de NIL, se detiene y devuelve ese valor. En caso contrario continúa con las siguientes formas o devuelve NIL si no queda ninguna.

## 3. Estructuras de control LET y LET*.

Las estructuras **LET** y **LET*** sirven para la definición de variables locales e incluyen un cuerpo en donde tienen validez las mismas. La sintaxis es idéntica para ambas:

```
(LET ((var1 valor1)
      (var2 valor2)
      ...
      (varN valorN) )
  cuerpo)
```

En un LET todos los valores iniciales de las variables se asignan en paralelo, lo que significa que una variable no puede depender de otra definida en el mismo LET, mientras que en un LET* el valor de las variables se asigna secuencialmente, y por tanto puede depender de las variables definidas anteriormente. Por ejemplo, son equivalentes:

```
(defun LIO (l)
  (let ((resto (rest l))
        (prim (first l))
        (segu (first (rest l)))
      (...))
```

```
(defun LIO* (l)
  (let* ((resto (rest l))
         (prim (first l))
         (segu (first resto))
      (...))
```

Evalúa la siguiente función y determina dónde empieza y dónde termina el ámbito de LET:

```
(defun mi-last (lista &optional (n 1))
  (let ((quitar (- (length lista) n)))
    (if (<= quitar 0)
       lista
      (nthcdr quitar lista))))
```

## 4. Estructura de control PROGN.

La forma especial **PROGN** define varias expresiones simbólicas que se evalúan secuencialmente. En el siguiente ejemplo, se podría haber evaluado ambos `setq` por separado y se hubiera obtenido en las variables globales A y B el mismo resultado:

```
? (progn (setq A (PAIRLIS '(UNO DOS TRES) '(1 2 3)))
         (setq B (PAIRLIS '(CUATRO CINCO) '(4 5) A)))
? A
? B
```

PROGN devuelve el resultado de la última forma evaluada. Sólo es útil por los efectos secundarios de las expresiones simbólicas. Por ejemplo, el efecto realizado por `setq`, que actualiza el valor de la variable A. Muchas formas especiales llevan en su cuerpo un PROGN implícito: DEFUN, LET, LET*, COND, DO, etc.

```
(defun factorial-largo (n)
   "Ejemplo de progn implícito"
   (print "Espere un momento, por favor...")
   (factorial n))
```

# 5. Estructuras de iteración.

Las funciones especiales que permiten definir iteraciones son `DO` y `DO*`:

```
(DO ( (var1 valorinic1 funcionpaso1)
      (var2 valorinic2 funcionpaso2)
      ...
      (varN valorinicN funcionpasoN) )
    ( condicion-final
      cuerpo-final )
    cuerpo)
```

Algunas de las partes de `DO` pueden no aparecer o estar vacías. El funcionamiento de `DO` se puede explicar de la siguiente manera:

1) Se inicializan las variables.
2) MIENTRAS no sea cierta la condicion-final hacer:
   - Evaluar cuerpo (puede ser una o varias sentencias = PROGN implícito)
   - Modificar las variables según la función paso.
3) Se evalúa cuerpo-final (PROGN implícito). El resultado de éste es el resultado del DO.

`DO*` es similar a `DO`. `DO` hace las asignaciones iniciales y las actualizaciones de los valores en paralelo como `LET`, mientras que `DO*` las hace secuencialmente como `LET*`.

# 6. Funciones destructivas.

Las siguientes funciones son versiones destructivas de otras ya conocidas:

- NCONC, versión destructiva de APPEND
- NREVERSE, versión destructiva de REVERSE

Las funciones no destructivas realizan una copia de los argumentos cuando es necesario para devolver el resultado esperado sin modificar ningún argumento. Las funciones destructivas, sin embargo, no realizan una copia sino que operan sobre las celdas cons de los parámetros que se les pasa. El cómo pueden modificar los argumentos es dependiente de la implementación.

Para comprender mejor su funcionamiento, realizar los siguientes ejemplos de uso de funciones no destructivas y destructivas, después de evaluar las siguientes definiciones de variables globales:

```
? (defvar *LETRAS* '(A B C))
? (defvar *NUMEROS* '(1 2 3))
? (defvar *ATOMOS* '(AA BB CC))
? (defvar X)
? (defvar Y)
```

a) Uso de APPEND y REVERSE. Evalúa las siguientes expresiones y comprueba después que todas las variables contienen los valores esperados.

```
? (setq X (APPEND *LETRAS* *NUMEROS*))
? (setq Y (REVERSE *ATOMOS*))
```

b) Uso de NCONC y NREVERSE.

```
? (setq X (NCONC *LETRAS* *NUMEROS*))
? (setq Y (NREVERSE *ATOMOS*))
```

Comprueba ahora qué variables contienen los valores esperados y cuáles han cambiado de valor. ¿Cuál es la razón de que hayan cambiado de valor?

Existen dos funciones destructivas básicas que reemplazan la parte CAR (RPLACA) y CDR (RPLACD) de la celda cons que se le pasa como primer parámetro por el elemento que se le pasa como segundo parámetro:

```
? (rplaca X 'NUEVO)
? X
? (rplaca Y '(OTRA COSA))
? Y
? (rplacd X '(TODO NUEVO))
? X
? (rplacd Y '(QUE CAMBIA))
? Y
```

Averigua a través de la documentación la utilidad de las siguientes funciones (no destructivas y destructivas).

- UNION y NUNION
- INTERSECTION y NINTERSECTION
- SUBST y NSUBST
- SUBLIS y NSUBLIS

# 7. Copia de listas.

Las siguientes funciones sirven para copiar listas:

- COPY-LIST Copia listas de un sólo nivel de anidamiento.
- COPY-ALIST Sirve para copiar **listas de asociación**.
- COPY-TREE Copia recursivamente **todos los conses** del árbol.

En el caso de COPY-LIST y COPY-ALIST, si hay listas anidadas no se realiza una copia de las mismas, pero sí quedan enlazadas con la lista principal. La diferencia estriba en que si se realiza una operación destructiva sobre alguna parte de la lista no copiada (sólo enlazada), esa modificación se realiza tanto en el original como en la copia. Veamos un ejemplo con las siguientes variables:

```
? (defvar original nil "Lista original")
? (defvar copia nil "Copia de Original")
```

Evalúa las siguientes expresiones que muestran el uso de COPY-LIST:

```
? (setq original '(ES UNA (LISTA DE LISTAS)))
? (setq copia (COPY-LIST original))
? (nsubst 'COSAS 'LISTAS original)
? (nsubst 'ALGUNA 'UNA original)
? copia
```

Dibuja los conses de ORIGINAL y COPIA después de realizar la copia. Después evalúa este ejemplo con COPY-ALIST:

```
? (setq original '((VERBO . ES) (ARTICULO . UNA)
                   (OBJETO LISTA DE LISTAS)))
? (setq copia (COPY-ALIST original)
? (nsubst 'COSAS 'LISTAS original)
? (nsubst 'ALGUNA 'UNA original)
? copia
```

De nuevo, dibuja los conses de ORIGINAL y COPIA. ¿Qué partes son comunes?

# 8. Ejercicios.

1. Reescribe un ejercicio de la práctica anterior en el que se pedía una función que debe aceptar dos fechas del tipo `'(24 OCTUBRE 1994)` y devolver la fecha más antigua, esta vez usando una estructura COND en lugar de IF anidados.

2. ¿Qué resultados produce la evaluación de las siguientes expresiones?

```
? (and t nil 'three)
? (and nil nil nil)
? (and 'talking 'heads)
```

```
? (or t nil 'three)
? (or nil nil nil)
? (or 'talking 'heads)

? (defvar *letras* '(A B C))
? (and (member 'B *letras*) 'SI)
? (first (or (member 'Z *letras*) (member 'B *letras*)))

? (defvar *numeros* 'NIL)
? (or *letras* *numeros* (intersection *letras* *numeros*))
? (setq *numeros* '(1 2 3))
? (and *letras* *numeros* (intersection *letras* *numeros*))
```

3. ¿Qué resultados produce la evaluación de las siguientes expresiones, dadas las siguientes variables?

```
(setq *letras* '(a b c d))
(setq *numeros* '(1 2 3))

? (progn (list 'a 'b) (list 'c 'd))
? (progn (list 'c 'd) (list 'a 'b))

? (progn (append letras numeros) (cons 'hola numeros))
? letras
? numeros

? (progn (setq letras numeros) (cons 'hola numeros))
? letras
? numeros

? (progn (cons 'hola numeros) (setq letras '(a b c d)))
? letras
? numeros

? (progn (cons 'hola numeros) (append letras numeros))
? letras
? numeros
```

4. Escribe una función que devuelva si un número se encuentra en un rango dado, siendo el número el primer argumento, y los valores inicial y final el segundo y el tercero.

5. Escribe un predicado que dados dos argumentos:
   - si el segundo es un átomo, devuelva si es igual al primero conceptualmente (EQL).
   - si el segundo es una lista, devuelva si el primero es miembro de la misma (MEMBER).

6. Dadas las siguientes definiciones,

```
(defun cambia (lista elemviejo elemnuevo)
   (rplaca (member elemviejo lista) elemnuevo) lista)

(defvar panel1 '(inicio fin introducir insertar suprimir))
(defvar panel2 panel1)
(defvar panel3 (copy-list panel1))
(defvar panel4 '(inicio fin introducir insertar suprimir))
```

¿qué valores habrá en la variables definidas después de la siguiente evaluación?:

```
? (cambia panel1 'introducir 'return)
(inicio fin return insertar suprimir)
```

7. Repite el ejercicio 4, cambiando la función `cambia`, y reinicializando las variables con `setq`:

```
(defun cambia (lista elemviejo elemnuevo)
  (cond ((null lista) nil)
        ((eq (first lista) elemviejo)
             (cons elemnuevo (rest lista)))
        (T   (cons (first lista)
              (cambia (rest lista) elemviejo elemnuevo)))))

(setq panel1 '(inicio fin introducir insertar suprimir))
(setq panel2 panel1)

? (cambia panel1 'introducir 'return)
(inicio fin return insertar suprimir)
```

8. Escribe una función **iterativa** que sume los números de una lista.

9. Escribe una función **iterativa** que haga lo mismo que `member`.

10. Escribe una función que calcule los cuadrados de los números de una lista.

```
? (cuadrados '(1 4 6 10))
(1 16 36 100)
```

11. Escribe una función que dada una lista de números, devuelva una lista de sublistas donde cada sublista corresponde a un número de la lista argumento junto con su cuadrado. Por ejemplo:

```
? (listacuadrados '(1 4 6 10))
((1 1) (4 16) (6 36) (10 100))
```

Práctica
5

# Recursividad: conceptos y bases.

Una característica de Lisp es su gran facilidad para programar recursivamente. En esta práctica se introduce la recursividad desde los ejemplos más sencillos, para aprender de forma sólida los aspectos básicos.

1. **Recursividad.**
2. **Ejercicios sencillos sobre resursividad.**
3. **Ejercicios sobre recursividad infinita.**
4. **Ejercicios sobre combinación del resultado de la llamada recursiva.**

## 1. Recursividad.

La recursividad consiste en resolver un problema dividiéndolo en un paso sencillo y un problema más pequeño. Cuando el problema llega a ser suficientemente pequeño tiene una solución trivial.

Veamos, por ejemplo, la definición recursiva del factorial de un número natural.

$$factorial\ (n) = \begin{cases} 1 & n=0 \\ n * factorial\ (n-1) & n>0 \end{cases}$$

El caso *n=0* se llama caso trivial porque tiene solución directa, y el caso *n>0* se llama caso general porque se soluciona recursivamente. Por ejemplo, ¿cómo calcularíamos recursivamente el factorial de 3?

```
factorial (3)= 3* factorial (2)     ; caso general
factorial (2)= 2* factorial (1)     ; caso general
factorial (1)= 1* factorial (0)     ; caso general
factorial (0)= 1                    ; caso trivial
```

Un planteamiento recursivo puede tener uno o más casos triviales y uno o más casos generales. Para que la recursión funcione bien debe cumplir:

- que el problema se haga más pequeño en cada llamada recursiva y la disminución lleve necesariamente a un caso trivial. Por ejemplo: en el factorial, *n* se disminuye en uno en cada llamada recursiva, con lo que necesariamente en *n* llamadas se alcanzará *n=0*.
- que en cada caso se devuelva el resultado adecuado, tanto en los casos triviales como en los generales. Por ejemplo: si para *n=0* devolviéramos cero, el factorial de cualquier número saldría, erróneamente, cero.
- que se hayan considerado todos los casos posibles, tanto triviales como generales.

## 2. Ejercicios sencillos sobre recursividad.

1. Escribe una función `QUE-RISA` que, dado un parámetro numérico natural, devuelva una lista de átomos JA. Esta función debe ser recursiva.

```
? (que-risa 3)
(JA JA JA)
```

2. Haz un `trace` de la función anterior.

3. Escribe una función `SUMA-LISTA` que, dada una lista de números cualesquiera, devuelva la suma de todos los números de esa lista.

```
? (suma-lista '(3,6 70 0,2))
73,8
```

4. Haz un `trace` de la función anterior. ¿Cómo se realiza la suma, de delante hacia atrás o de atrás hacia delante?

5. Escribe una función `TODOS-PARES` que, dada una lista de números enteros, devuelva `T` si todos los números son pares y `NIL` si alguno es impar. (NOTA: Se considera que si la lista está vacía, entonces todos los números de la lista son pares).

```
? (todos-pares '(3 2 3))
nil
? (todos-pares '(6 2 80))
t
```

6. Escribe una función recursiva que haga lo mismo que `ASSOC`. El primer parámetro ha de ser una palabra clave y el segundo una lista de asociación.

```
? (mi-assoc 'c1 '((c1 . d1) (c2 . d2)))
(c1 . d1)
```

7. Escribe una función recursiva que haga lo mismo que `MEMBER`. El primer parámetro ha de ser un átomo y el segundo una lista.

```
? (mi-member 'poco '(muchisimo mucho algo poco poquisimo))
(poco poquisimo)
```

8. Haz un `trace` de la función anterior en el caso de que el átomo se encuentre en la lista y en el caso en que no se encuentre en la lista. ¿Cuál es la condición de parada en cada caso?

9. Define una función que determine si una lista es palíndromo o no.

```
? (palindromo '(a b c b a))
T
? (palindromo '(a b c c b a))
T
? (palindromo '(a b c c c a))
NIL
```

10. Las primitivas de Lisp 1+ y 1- sirven para incrementar y decrementar en uno cualquier número. Escribe una función recursiva que sume dos números naturales utilizando únicamente estas primitivas y las estructuras de control necesarias.

11. Escribe una función `TODOS-IGUALES`, que dada una lista de átomos cualesquiera, devuelva T si todos ellos son iguales y NIL si alguno difiere.

```
? (todos-iguales '(A A A A A A A A))
T
? (todos-iguales '(A A A B A))
NIL
```

12. Haz un `trace` de la función anterior. ¿Cuál es la condición de parada en cada caso?

13. Escribe una función que devuelva el primer elemento de la lista argumento que sea átomo y no sublista.

```
? (primer-elem-atomo '(((a b c) (d) e) (f g h) i))
I
```

14. Escribe una función que encuentre el primer átomo que aparezca en un árbol.

```
? (primeratomo '(((a b c) (d) e) (f g h) f))
A
```

## 3. Ejercicios sobre recursividad infinita.

15. Di por qué es incorrecta la siguiente función:

```
(defun hayalgun7 (lista)
   (cond ((equal (first lista) 7)   T)
         (T                         (hayalgun7 (rest lista)))))
```

16. ¿Y ésta?

```
(defun mi-member (elem lista)
   (let ((encontrado (mi-member elem lista))
        (if (null lista)
            NIL
            encontrado)))
```

17. Escribe la función recursiva infinita más corta que se te ocurra.

18. Supón la siguiente definición de una variable y la modificación a continuación:

```
? (defvar atlantida '((desaparecida)))
? (rplacd atlantida atlantida)
```

a) ¿Cuál es el resultado? Dibújalo.

b) Dada la siguiente función:

```
? (defun hay-atomos (lista)
     (cond ((null lista)         NIL)
           ((atom (first lista)) T)
           (T                    (hay-atomos (rest lista)))))
```

¿Qué efecto tendrá la siguiente llamada?

```
? (hay-atomos atlantida)
```

19. Escribe una función que encuentre el primer número impar de una lista de números enteros o NIL si no lo hay (NOTA: Suponemos una lista de 1 sólo nivel). Una vez terminado, contesta: ¿qué ocurriría si le quitásemos el test de lista vacía? ¿en qué casos no funcionaría bien?

20. Di por qué es incorrecta la siguiente función, que comprueba si hay alguna sublista en la lista argumento:

```
(defun haysublistas? (lista)
   (cond ((null lista)          NIL)
         (listp (first lista))  T)
         (T                     (haysublistas? lista))))
```

## 4. Ejercicios sobre combinación del resultado de la llamada recursiva.

21. Escribe una función `SUMATORIO`, que sume todos los números entre dos dados, inclusive.

```
? (sumatorio 2 7)
27
```

Este problema requiere sumar el resultado de la llamada recursiva con el número actual.

22. Escribe una función que haga la cuenta atrás desde un número que se le pase como argumento, devolviendo el resultado en forma de lista.

```
? (cuenta-atras 5)
(5 4 3 2 1)
```

23. Modifica la función anterior para que devuelva una lista hasta el 0.

24. Escribe una función que dada una lista de números devuelva una lista con todos los números incrementados en una cantidad que se le da como argumento. El argumento debe ser opcional y por defecto será 1.

```
? (inc-lista '(1 3 5) 7)
(8 10 12)
? (inc-lista '(1 3 5))
(2 4 6)
```

25. Escribe una función recursiva que haga lo mismo que REVERSE.

26. Escribe una función recursiva que haga lo mismo que APPEND, pero con sólo dos listas como argumento.

Práctica
6

# Recursividad: una forma natural de programar.

Esta práctica trata de profundizar en la programación recursiva para ser capaces de solucionar problemas cada vez más complejos. Además, se estudian las operaciones de entrada/salida, que permiten al usuario interactuar con el programa.

1. **Operaciones de Entrada/Salida.**
2. **Ejercicios sobre entrada/salida de datos.**
3. **Ejercicios sobre combinación condicional del resultado de la llamada recursiva.**
4. **Ejercicios sobre recursividad simultánea con varios parámetros.**
5. **Ejercicios "típicos" sobre recursión múltiple: fibonacci y hanoi.**
6. **Ejercicios sobre recursión múltiple.**

## 1. Operaciones de Entrada/Salida.

Las siguientes funciones se emplean en la entrada/salida de datos al programa Lisp:

```
(READ)                    ;lee expresión simbólica
(READ-LINE)               ;lee cadena
(READ-CHAR)               ;lee carácter

(PRINT expresión)
(FORMAT destino string-de-control
        parametro1 parametro2 ... parametroN)
```

PRINT escribe el resultado de la evaluación de *expresión*, de la misma forma que Lisp escribe los resultados. READ lee en el mismo formato. READ-LINE y READ-CHAR leen cadenas y caracteres. FORMAT permite realizar

impresiones más elegantes: destino T singnifica la salida de datos estándar; en el *string-de-control* se pueden intercalar directivas; y deberá haber un *parámetro* por cada directiva.

Las directivas van precedidas de ~ y las más corrientes son:

~% Nueva línea

~S Imprimir según las reglas de evaluación

~A Igual que ~S, salvo que elimina las comillas de los strings

~nT Añade blancos hasta llegar a la columna n (si se ha alcanzado, sólo inserta 1).

## 2. Ejercicios sobre entrada/salida de datos.

1. ¿Has jugado alguna vez a "Las Tres en Raya"? Define una función DIBUJA-3-EN-RAYA que dibuje el tablero del juego, a partir de una lista de 9 elementos que pueden ser: una X, una O, o NIL. En el tablero deberán aparecer las X y las O, y NIL significa casilla vacía. Por ejemplo, la siguiente expresión debería dibujar:

```
? (DIBUJA-3-EN-RAYA '(X O O NIL X NIL O NIL X))
X | O | O
-----------
  | X |
-----------
O |   | X
```

2. Escribe una función DIBUJA-LINEA que dibuje una línea de una longitud especificada haciendo (FORMAT T "*") el número de veces que se le diga por parámetros.

3. Escribe una función recursiva DIBUJA-CAJA que llame a DIBUJA-LINEA para dibujar una caja de dimensiones especificadas por parámetros.

```
? (DIBUJA-CAJA 10 4)
**********
**********
**********
**********
```

4. Escribe una función que haga una cuenta atrás, escribiendo en cada paso el número.

5. Escribe una función que tome una lista de números y calcule sus cuadrados, escribiendo una línea por número donde se muestre, por ejemplo: 4 x 4 = 16

## 3. Ejercicios sobre combinación condicional del resultado de la llamada recursiva.

6. Escribe una función recursiva que, dada una lista de átomos cualesquiera obtenga otra lista con sólo los números de la lista original (elimina los símbolos).

7. Escribe una función que dada una lista de números devuelva una lista con sólo los números que sean pares (habiendo eliminado los impares).

8. Escribe una función recursiva que devuelva una lista de elementos que se encuentren en ambas listas argumento (versión de `INTERSECTION`). Se supone que los elementos no están repetidos.

9. Escribe una función que haga lo mismo que `UNION`. Se supone que los elementos no están repetidos.

10. Escribe una función que, dada una lista de átomos cualesquiera, y dado un átomo de esa lista, devuelva una lista igual donde se han borrado todas las ocurrencias de ese átomo.

11. Escribe una función que dadas dos listas ordenadas de números, devuelva todos los números ordenados en una sola lista.

```
? (mezcla '(0 4 6 20) '(1 3 5 11 13 15 17))
(0 1 3 4 5 6 11 13 15 17 20)
```

## 4. Ejercicios sobre recursividad simultánea con varios parámetros.

12. La siguiente función, que hace lo mismo que `nth`, realiza siempre n llamadas recursivas:

```
(defun mi-nth (n lista)
   (cond ((zerop n) (first lista))
         (t (mi-nth (- n 1) (rest lista)))))
```

Modifícala para que, en una llamada del tipo `(mi-nth 1000 '(A B C))` no realice las 1000 llamadas recursivas, sino que pare cuando encuentre el final de la lista. Comprueba que funciona correctamente utilizando la función `TRACE`.

13. Escribe una función que determine si la primera lista argumento es más larga que la segunda, sin utilizar la función `LENGTH`. La función deberá acabar su ejecución al llegar al final de la lista más corta.

```
? (primera-más-larga '(1 2 3) '(a b))
T
```

14. Escribe una función recursiva que haga lo mismo que `NTHCDR`. Tiene dos argumentos: una lista, y el número de veces que hay que realizar el `CDR` de la lista.

15. Escribe una función que dadas dos listas de números, devuelva una lista de las diferencias entre elementos en la misma posición. Si una lista es más corta que otra, se supondrá que los números a restar son ceros.

```
? (resta '(0 4 6 20) '(1 3 5 11 13 15 17))
(-1 1 1 9 -13 -15 -17)
```

16. Escribe una función que dadas dos listas de números, devuelva la lista de los productos de los elementos en la misma posición. Si una lista es más corta que otra, se terminará de multiplicar, despreciando el resto de números de la otra lista.

```
? (multiplica '(0 4 6 20) '(1 3 5 11 13 15 17))
(0 12 30 220)
```

## 5. Ejercicios “típicos” sobre recursión múltiple: fibonacci y hanoi.

17. **Números de Fibonacci**: Elabora una función recursiva que calcule el enésimo número de la serie de Fibonacci, sabiendo que:

*Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2)*

*Fibonacci (0) = Fibonacci (1) = 1*

18. **Las torres de Hanoi**: Un mito ancestral dice que en algún templo en Hanoi, el tiempo es registrado por monjes que se ocupan en transferir 64 discos de un poste a otro, de un total de 3 postes que llamaremos A, B y C. Para hacerlo, siguen 3 reglas:

- Inicialmente todos los discos están en A y cada disco descansa sobre uno más grande.
- Sólo un disco puede moverse a la vez.
- Todos tienen diferente diámetro y ninguno puede ponerse sobre otro más pequeño.

La secuencia de movimientos para transferir *n* discos del poste A al B sería:

- transferir los primeros *n*-1 discos de A a C (usando B como espacio de trabajo).
- mover el disco más grande a B.
- transferir los *n*-1 discos de C a B (usando A como espacio de trabajo).

a) Escribe una función recursiva que cuente los movimientos requedidos para mover un número de discos dado de un poste a otro.

```
? (torres-de-hanoi-1 1)
1
? (torres-de-hanoi-1 10)
```

```
1023
```

b) Escribe una función recursiva que escriba una serie de instrucciones para mover los discos, en lugar de contar la cantidad de movimientos

```
? (torres-de-hanoi-2 '(1 2 3) 'a 'b 'c)
Transfiere 1 de A a B.
Transfiere 2 de A a C.
Transfiere 1 de B a C.
...
NIL
```

c) ¿Se puede reducir el coste temporal de la función a)? ¿cómo?

d) ¿Se puede reducir el coste temporal de la función b)? ¿por qué?

## 6. Ejercicios sobre recursión múltiple.

En este apartado y en las prácticas siguientes nos referiremos a las "listas de listas de cualquier nivel de anidamiento" como *árboles*. Esta denominación proviene del hecho de que cada uno de los conses de la lista podría identificarse como un nodo de un árbol binario, donde el nodo raíz es el primer cons de la lista y las hojas son los átomos de la lista. El CAR de cada CONS es el hijo izquierdo del nodo y el CDR el hijo derecho. Por ejemplo, dibujando la lista `((A . B) C)` en forma de árbol se puede ilustrar este aspecto: las hojas de este árbol son `A`, `B`, `C` y `NIL`.

19. Haz una función que busque en un árbol de cualquier tipo y devuelva `T` si encuentra algún número, y `NIL` si no.

20. Haz una función que cuente el número de átomos de un árbol, incluidos los nil.

21. Haz una función que cuente el número de celdas `CONS` en un árbol.

22. Haz una función que sume todos los valores que aparezcan en un árbol, ignorando todos los elementos que no sean números.

23. Elabora una función similar a MEMBER, que busque recursivamente un átomo en cualquier lista anidada dentro de una lista. Es recomendable hacer trazas para comprobar bien su funcionamiento.

```
? (mi-member 'x '(sqrt (/ (+ (expt x 2) (expt y 2)) 2)))
T
```

24. Haz una función que devuelva todos los elementos de un árbol en una lista de un sólo nivel.

```
? (aplana '((N O) ((T E) (R I A S)) (Q U E) ((E S) (P E O R))))
(N O T E R I A S Q U E E S P E O R)
```

25. Escribe una función recursiva que sustituya todos los valores numéricos de un árbol por el símbolo `NUMERO` y todos los símbolos por el símbolo `SIMBOLO`.

```
? (sustituye-tipos '((3 AMIGOS) CENAN (1 PIZZA) EN
                     (LA (AVENIDA 7))))
((NUMERO SIMBOLO) SIMBOLO (NUMERO SIMBOLO) SIMBOLO (SIMBOLO
(SIMBOLO NUMERO)))
```

26. Escribe una función recursiva que localice una palabra clave dentro del árbol y devuelva su contenido, sabiendo que puede encontrarse en cualquier nivel de anidamiento:

```
? (busca 'O-DIRECTO '((SUJETO 3 AMIGOS) (PREDICADO (VERBO CENAN)
                 (O-DIRECTO 1 PIZZA) (O-LUGAR EN LA TRAVIATTA)))
(1 PIZZA)
? (busca 'O-DIRECTO '((O-DIRECTO 1 PIZZA) (VERBO SE COMIERON)
                 (SUJETO 3 AMIGOS) (O-LUGAR EN LA TRAVIATTA)))
(1 PIZZA)
```

27. Escribe una función que compare dos árboles devolviendo T si todos los tipos de sus átomos son iguales, teniendo en cuenta sólo los tipos `NUMBER` y `SYMBOL`. (OBSERVACION: En este ejemplo, la función `sustituye-tipos` nos ayuda a identificar la estructura de las frases. ¿Hasta donde podríamos llegar si logramos identificar: verbo, sujeto, nombre, adjetivo, adverbio, etc.?)

```
? (compara-tipos-1 '((3 AMIGOS)CENAN(1 PIZZA) EN (LA (AVENIDA 7)))
          '((4.0 ANCIANOS) JUEGAN (10 PARTIDAS)Y(CANTAN (LAS 40)))
T
```

28. Generaliza la función del ejercicio anterior para cualquier tipo de átomo (es decir, que distinga entre números enteros y reales). Úsese la función TYPE-OF.

```
? (compara-tipos-2 '((3 AMIGOS)CENAN(1 PIZZA) EN (LA (AVENIDA 7)))
           '((4.0 ABUELOS) JUEGAN (10 PARTIDAS)Y(CANTAN (LAS 40)))
NIL
```

29. Haz una función que devuelva la profundidad máxima de un arbol binario.

```
? (profundidad-arbol '((A B C D)))
5
? (profundidad-arbol '((A . B) . (C . D)))
2
? (profundidad-arbol '(A B C))
3
```

30. Haz una función que devuelva la profundidad máxima de paréntesis anidados en una lista. Por ejemplo:

```
? (max-parentesis '((A B C D)))
2
? (max-parentesis '((A . B) (C . D)))
2
? (max-parentesis '(A B C))
1
```

Práctica
7

# Transformaciones de listas. Filtros.

En esta práctica se estudian funciones avanzadas de manejo de listas como son los filtros, las transformaciones de listas, funciones de conteo y localización... Estas operaciones permiten al programador aplicar casi cualquier transformación a una o varias listas de forma sencilla y concisa.

1. **Transformaciones de listas.**
2. **Ejercicios sencillos de transformaciones de listas.**
3. **Ejercicios.**



## 1. Transformaciones de listas.

Hay operaciones específicas que simplifican las operaciones con listas. Se les llama **transformaciones de listas**. MAPCAR se aplica a todos los CAR de la lista y MAPLIST se aplica a los CDR de la lista, y ambos acumulan todos los resultados en una sola lista (usando LIST).

```
(MAPCAR #'funcion lista1 lista2 ... listaN)
(MAPLIST #'funcion lista1 lista2 ... listaN)
```

Algunos ejemplos sencillos:

```
? (MAPCAR #'+ '(1 2 3) '(4 5 6))
(5 7 9)
? (MAPCAR #'ODDP '(1 2 3 4 5 6))
(T NIL T NIL T NIL)

? (MAPLIST #'APPEND '(1 2 3) '(4 5 6))
((1 2 3 4 5 6) (2 3 5 6) (3 6))
? (MAPLIST #'UNION '(a b c) '(a c b))
((A B C) (B C) (B C))
```

MAPC y MAPL son similares a los anteriores, pero no acumulan en una lista los resultados, sólo sirven por sus efectos secundarios. MAPCAN y MAPCON también son similares a MAPCAR y MAPLIST, pero concatenan los

resultados en lugar de listarlos (usando NCONC), por lo que se suelen usar como filtro.

Cuando lo que se quiere es eliminar elementos de una lista que cumplan o no una condición se les llama **filtros**. MAPCAN y MAPCON son ejemplos. Existen otros filtros:

```
(REMOVE-IF #'condicion lista)
(REMOVE-IF-NOT #'condicion lista)

? (REMOVE-IF #'ODDP '(1 2 3 4 5 6))
(2 4 6)
? (REMOVE-IF-NOT #'NUMBERP
          '(EL EQUIPO 57 DE LA SECCION 13 OBTUVO 89 PUNTOS))
(57 13 89)
```

Existen otras de **conteo** y **localización**: Busca su documentación e intenta utilizarlas en algún caso sencillo.

```
(COUNT-IF #'condicion lista)
(FIND-IF #'condicion lista)
```

En todas las funciones estudiadas se pueden usar funciones definidas previamente o bien definirlas en el propio lugar de uso en forma de lo que se llama una **expresión lambda**. Veamos un ejemplo:

```
? (MAPCAR #'(lambda (num1 num2) (* (+ num1 num2) 2))
          '(1 2 3)
          '(4 5 6))
(10 14 18)
? (REMOVE-IF #'(lambda (x) (> x 3))
             '(7 1 2 9 0))
(1 2 0)
```

## 2. Ejercicios sencillos de transformaciones de listas.

1. Dada una lista de números haz las funciones que devuelvan las siguientes listas:
   a) Una lista que determine si cada número es cero o no.
   b) Una lista que determine si cada número es > que 5 o no.
   c) Una lista de los cuadrados de los números.

2. Dada una lista de palabras en inglés y francés, define funciones que realicen las siguientes operaciones sobre todos los elementos de la lista:

```
? (defvar *words* '((one un) (two deux) (three trois)
                    (four quatre) (five cinq)))
```

   a) Devolver una lista con sólo las palabras en inglés.
   b) Devolver una lista con sólo las palabras en francés.
   c) Del diccionario inglés-francés crear un diccionario francés-inglés.

d) Dada una función TRADUCIR, traducir una lista de palabras en inglés al francés.

```
(defun traducir (x)
   (second (assoc x *words*)))
```

3. Dada una lista de elementos SI y NO, haz una función que invierta dichos elementos. Usa para ello una expresión lambda. Por ejemplo:

```
? (cambia '(SI NO SI SI NO))
(NO SI NO NO SI)
```

4. Dada una lista de números, haz las funciones que realicen las siguientes acciones:
   a) Devolver una lista con los números mayores que uno y menores que 5.
   b) Contar el número de ceros en la lista.
   c) Contar la cantidad de números negativos de la lista.

5. Dada una lista de números, elabora una función que extraiga sólo aquéllos números que cumplan que su cuadrado sea mayor que un valor que se pasa por parámetros. Por ejemplo:

```
? (borra-grandes '(1 2 3 4 5) '24)
(5)
? (borra-grandes '(1 2 3 4) '25)
NIL
? (borra-grandes '(1 2 3 4) '0)
(1 2 3 4)
```

6. Haced una función que dada una lista de números, borre los que sean mayores que un valor dado, y devuelva una lista de los cuadrados de dichos números.

```
? (borra-y-cuadrado '(1 2 3 4 5) '3)
(1 4 9)
? (borra-y-cuadrado '(4 5 1 2 3) '6)
(16 25 1 4 9)
? (borra-y-cuadrado '(4 5 1 2 3) '0)
NIL
```

7. Dada una lista de números reales, L, y un número real X, haz una función que busque en la lista el primer elemento que sea "casi-igual" que X, es decir, que cumpla que $X-1 \leq r \leq X+1$, devolviendo el número encontrado en la lista, o NIL si ninguno cumple la condición. Se debe usar FIND-IF.

```
? (busca 5 '(3.7 7.2 5.3 6.9 5.0 13.9))
5.3
```

8. Dada una lista de átomos, escribe una función que cuente cuántas veces aparece un determinado átomo.

```
? (cuenta-atomos 'Y '(PEDRO Y VILMA CENAN CON PABLO Y BETTY))
2
```

# 3. Ejercicios.

9. Escribe una función que elimine los elementos repetidos de una lista usando, entre otras funciones, MAPLIST.

10. Se dispone de una lista con información sobre un zoo, donde aparece los nombres de los animales del zoo, junto con su especie y la familia a la que pertenecen en este orden. Por ejemplo:

```
(defvar *ZOO* '( (esperanza oso-pardo mamifero)
                 (mimosin chimpance mamifero)
                 (eulalia urraca ave)
                 ...)
```

Se pide:

- una función que, dada una especie, devuelva los nombres de todos los animales que pertenecen a ella.

```
? (especie 'oso-pardo *ZOO*)
(ESPERANZA ...)
```

- una función que devuelva la lista de familias que estan representadas en el zoo (sin repetir los nombres). Usa MAPCAN, pero ten cuidado con la acción destructiva de NCONC.

```
? (familias *ZOO*)
(MAMIFERO AVE ...)
```

11. Se quiere escribir un programa para los refugios de montaña de una asociación de manera que los encargados puedan registrar en el ordenador las reservas de los distintos refugios que se pidan. Dichas reservas se guardan en una lista y cada reserva es a su vez una lista de esta forma:

```
(numero-de-reserva (TIPO . tipo) (NOMBRE . nombre-refugio)
                   (INICIO dia mes año)
                   (FIN dia mes año)
                   (RESPONSABLE nombre-de-la-persona-responsable))
```

Excepto el número de reserva, el resto de datos pueden estar desordenados. Por ejemplo:

```
((R8 (TIPO . refugio) (NOMBRE . D205) (INICIO 7 Febrero 1996)
     (FIN 17 Marzo 1996) (RESPONSABLE “Eduardo Manostijeras”))
 (R9 (TIPO . albergue) (NOMBRE . D403) (FIN 2 Marzo 1996)
     (RESPONSABLE “Julio Cesar”) (INICIO 1 Marzo 1996))
 (R10 (TIPO . refugio) (NOMBRE . D501) (RESPONSABLE “Juan del Pi”)
     (FIN 12 Marzo 1996) (INICIO 1 Marzo 1996))
...)
```

Escribe las siguientes funciones usando **transformaciones de listas** y/o **filtros**:

- una función que dado un nombre de refugio, devuelva si tiene o no alguna reserva.

- una función que dado un tipo de refugio, devuelva una lista de todos los refugios de este tipo que aparecen en las reservas (sólo el nombre).
- una función que dado un día, mes y año, devuelva una lista con las reservas que comiencen en ese día.
- una función que dado un día, mes y año, devuelva una lista con las reservas que comiencen en ese dia, pero poniendo solo el nombre del refugio y la fecha de fin de la reserva: `(nombre-refugio dia mes año).`

12. Dado un árbol, escribe una función que devuelva sólo los números en el mismo nivel de anidamiento original (elimina los símbolos).

```
? (solo-numeros '(((1 RELOJ) MARCA (LAS 9)) Y ((3 RELOJES)
   MARCAN (LAS (9 Y 20))) ))
(((1) (9)) ((3) (9 20)))
? (solo-numeros '((PEDRO Y VILMA) CENAN CON (PABLO Y BETTY)))
Nil
```

13. Escribe una función que, dada una lista de elementos cualesquiera, devuelva una lista de asociación donde las claves son los tipos de los elementos y asociado a cada tipo están los elementos de ese tipo. Recuerda que puedes utilizar la función TYPE-OF.

```
? (asocia-tipos '(SON LAS 8 Y 20 (JA JA JA) Y ES (JUEVES)))
((FIXNUM 8 20) (SYMBOL SON LAS Y Y ES) (CONS (JA JA JA)
(JUEVES)))
```

14. Se cuenta con una base de datos de la red de Universidades de un país determinado, que incluye el grafo de conexiones entre ellas. Entre los datos que se precisan de cada universidad están: su nombre, la lista de universidades a las que está conectada directamente, y el tiempo de respuesta de dicha universidad a la hora de poner en ruta un mensaje que está de paso.

```
(defvar *UNIVERSIDADES* '(
   (uji (NOMBRE . "Universitat Jaume I") (TIEMPO . 15)
        (CONEX uv upc))
   (uv  (NOMBRE . "Universitat de Valencia") (TIEMPO . 20)
        (CONEX upv uji upm ua))
   (upc (NOMBRE . "Universitat Politecnica de Cataluña")
        (CONEX uji upm) (TIEMPO . 18))
   (upv (NOMBRE . "Universitat Politecnica de Valencia")
        (CONEX uv ua) (TIEMPO . 17))
   (upm (NOMBRE . "Universidad Politecnica de Madrid")
        (CONEX um uah uac upc) (TIEMPO . 19))
   (ua  (NOMBRE . "Universitat de Alicante") (TIEMPO . 22)
        (CONEX uv upv))
   ...))
```

Escribe las siguientes funciones, utilizando **transformaciones de listas y/o filtros.**

a) Una función que, dada una universidad, devuelva una lista de las universidades a las que esté conectada y que tengan un tiempo de respuesta menor que uno dado. En el resultado debe incluirse la clave y el tiempo de respuesta de cada universidad, igual que en el siguiente ejemplo:

```
? (conex-rapida 'uv '18)
((UPV . 17) (UJI . 15))
```

b) Una función que devuelva todas las conexiones existentes en forma de pares de claves (`uni1 . uni2`). Por cada conexión existente deben aparecer dos pares en la lista, uno para cada sentido de la conexión.

```
? (pares)
((UJI . UV) (UJI . UPC) (UV . UPV) (UV . UJI) (UV . UPM)
(UV . UA) (UPC . UJI) (UPC . UPM) ...)
```

15. El jefe de producción de una fábrica guarda los datos de los materiales que emplea en una lista del siguiente tipo:

```
(defvar *materiales* '(
  (CINTA-FINA (TIPO CINTA-AISLANTE) (GROSOR 2 cm))
  (TORNILLO5 (DIAMETRO 5 mm) (TIPO TORNILLO) (LARGO 5 cm))
  (TORNILLO10 (TIPO TORNILLO) (DIAMETRO 10 mm) (LARGO 5 cm))
  (TORNILLO5LARGO (DIAMETRO 5 mm) (TIPO TORNILLO) (LARGO 7 cm))
  (CINTA-GRUESA (TIPO CINTA-AISLANTE) (GROSOR 5 cm))
  (TUERCA5 (TIPO TUERCA) (DIAMETRO 5 mm))
  (TUERCA10 (DIAMETRO 10 mm) (TIPO TUERCA))
  ...)
"Datos de materiales para cadena de producción")
```

Esta lista se almacena en una variable global llamada **`*materiales*`** y contiene todos los materiales utilizados en la cadena de ensamblaje de la fábrica. Cada uno se describe mediante un tipo de material y algunos datos más, diferentes para cada tipo. Haz las siguientes funciones, utilizando **transformaciones de listas y/o filtros.**

a) Una función que devuelva las claves de aquellos materiales de la lista `*materiales*` que sean de un tipo dado:

```
? (material-tipo 'TORNILLO)
(TORNILLO5 TORNILLO10 TORNILLO10CORTO)
? (material-tipo 'MESA)      ;no hay mesas en la lista
NIL
```

b) Una función que devuelva los diámetros de una lista de claves de materiales cualesquiera (en el mismo orden). Por ejemplo:

```
? (diametros-de '(TORNILLO5 TORNILLO10CORTO))
((5 mm)(10 mm))
? (diametros-de '(CINTA-FINA))
(NIL)
```

Práctica
8

# Abstracción de datos y funciones.

La abstracción de datos y funciones es un aspecto fundamental que no hay que descuidar a la hora de abordar un problema complejo. En esta práctica se ejercita el desglosamiento de los problemas en partes significativas.

1. **Abstracción de datos y de funciones.**
2. **Ejercicio: Lista de libros.**
3. **Ejercicio: Juego de cartas.**
4. **Ejercicio: Mundo de bloques.**
5. **Otros ejercicios.**



## 1. Abstracción de datos y de funciones.

La abstracción en la definición de los procedimientos y en el manejo de los datos es muy importante para construir programas grandes y complicados. Dada la gran potencia de Lisp en el procesamiento de listas (LISP = LIST Processor), se recomienda disponer los datos en **listas** de las que se extraerán en cada momento los datos necesarios por medio de funciones específicas para ello.

Así pues, habrán funciones con distintos **niveles de abstracción**: desde los que toman los datos directamente y por tanto conocen los detalles de cómo están dispuestos (funciones de acceso) hasta los que realizan tareas de nivel superior, obviando los detalles.

Esto permite:

- Definir funciones *breves*, fáciles de leer y comprender.
- Utilizar la *recursión* de una forma más clara.
- Facilitar la *depuración* de los programas.
- Modificar con mayor facilidad la estructura de los datos, sin necesidad de modificar todo el programa sino sólo las *funciones de acceso*.

## 2. Ejercicio: Lista de libros.

1. Dada una lista de asociación que contiene información sobre un libro determinado, haz las funciones necesarias para extraer el autor, título y la clasificación del libro, sabiendo que esos datos se marcan en la lista de asociación con las palabras clave AUTOR, TITULO, CLASIFICACION.

```
? (DEFVAR LIBRO1 '((AUTOR (MIGUEL DE CERVANTES))
                   (TITULO (DON QUIJOTE DE LA MANCHA))
                   (CLASIFICACION (NARRATIVA HISPANICA CLASICA))))
LIBRO1
? (AUTOR LIBRO1)
(MIGUEL DE CERVANTES)
```

2. Dada una lista de libros del tipo anterior, desarrolla una función **recursiva** que extraiga una lista sólo de autores y otra que extraiga una lista sólo de libros. Utiliza las funciones de acceso definidas en el ejercicio anterior.

3. Dada una lista de libros, desarrolla una función **recursiva** que extraiga una lista (incluyendo toda la información de cada libro) de aquellos libros que contengan una determinada palabra en su clasificación.

4. Utiliza la función anterior para hacer otra función que acepte como argumento una palabra o una lista de palabras. En el caso de una lista, debe buscar los libros que contengan en su clasificación todas las palabras de la lista (recursivamente).

5. Supongamos que se decide cambiar la estructura de un libro por la siguiente:

```
? (setq LIBRO1 '((AUTOR MIGUEL DE CERVANTES)
                 (TITULO DON QUIJOTE DE LA MANCHA)
                 (CLASIFICACION NARRATIVA HISPANICA CLASICA)))
```

   a) Modifica las funciones de acceso para que se adecúen al nuevo formato.

   b) Comprueba que no necesitas modificar ninguna otra de las funciones definidas en los ejercicios anteriores.

6. Modifica las funciones del ejercicio 2 para que hagan lo mismo, pero utilizando las **transformaciones de lista y/o filtros** adecuados.

7. Modifica la función del ejercicio 3 para que haga lo mismo, utilizando las **transformaciones de lista y/o filtros** adecuados.

8. ¿Será necesario modificar la función del ejercicio 4 para que funcione correctamente con las nuevas definiciones de los ejercicios 6 y 7?

9. Haz una función para obtener una lista de todas las palabras diferentes utilizadas como clave en la clasificación de los libros.

10. Realiza una función que obtenga una lista de libros (con toda su información) que contenga cualquiera de las clasificaciones que se le pasan en una lista.

## 3. Ejercicio: Juego de cartas.

Supongamos que queremos jugar a las cartas. Cada carta se representa con una lista de dos átomos: el número y el palo. Una mano se representa con una lista de cartas.

11. Escribe las funciones de acceso NUMERO y PALO que devuelvan el número y palo de una carta, respectivamente.

12. Supongamos que definimos la siguiente constante:

```
? (defconstant *colores* '((treboles . negro) (corazones . rojo)
                           (diamantes . rojo) (picas . negro)))
```

Haz una función que, dada una carta, devuelva su color.

13. Dada una mano de cartas, como por ejemplo:

```
? (defvar miscartas '((3 . corazones) (5 . tréboles)
          (2 . diamantes) (4 . diamantes) (as . picas)))
```

Haz una función que devuelva la primera carta de un color determinado, o NIL si no hay.

14. Dada una mano de cartas, como la anterior, haz una función que devuelva una lista de todas las cartas de un color determinado.

15. Dada una mano de cartas y un palo, haz una función que devuelva una lista de los números de las cartas de ese palo. Por ejemplo:

```
? (listapalo miscartas 'diamantes)
(2 4)
```

16. Dada la siguiente variable global:

```
(defvar *TODOS-LOS-NUMEROS*
        '(2 3 4 5 6 7 8 9 10 sota caballo rey as))
```

Haz un predicado que, dadas dos cartas devuelva T si la primera tiene número mayor que la segunda, según el orden en `*TODOS-LOS-NUMEROS*`.

17. Escribe una función que devuelva la carta de número más grande en una mano.

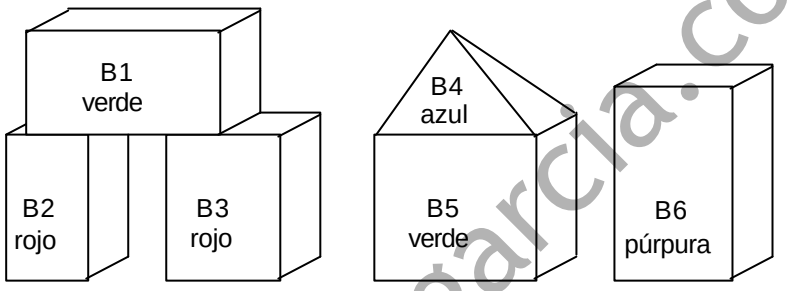
## 4. Ejercicio: Mundo de bloques.

Vamos a desarrollar un pequeño sistema para representar conocimiento sobre una escena de un "mundo de bloques". Las aserciones sobre los objetos

de la escena se representan como tripletes de la forma (*bloque atributo valor*). He aquí algunas aserciones sobre los atributos de un bloque B2:

```
(b2 forma ortoedro)
(b2 color rojo)
(b2 tamaño pequeño)
(b2 soporta-a b1)
(b2 a-la-izquierda-de b3)
```

Una colección de aserciones como estas se llama una **base de datos**. Dada una base de datos que describe los bloques de la figura, podemos escribir funciones para responder preguntas como: "¿De qué color es B2?" o "¿Qué bloques soportan a B1?". Para ello debemos usar una función que se llame **busca-patron**. Para buscar el color de B2 usaremos el patrón (b2 color ?), y para buscar los bloques que soportan a B1 usaremos el patrón (? soporta-a b1).




18. Define una variable global que albergue todas las aserciones sobre el mundo de bloques. Llámala *DATABASE*.

19. Escribe una función COMPRUEBA-ELEMENTOS que toma dos símbolos como argumentos y, si son iguales o si el segundo es un interrogante, devuelve T.

20. Escribe una función COMPRUEBA-TRIPLETE que tome una aserción y un patrón como argumentos y devuelva T si comprobando elemento a elemento la aserción coincide con el patrón.

```
? (comprueba-triplete '(b2 color rojo) '(b2 color ?))
T
```

21. Escribe la función BUSCA-PATRON que tome un patrón como argumento y devuelva una lista con todas las aserciones de la base de datos que se correspondan con el patrón. La base de datos se encuentra en la variable global *DATABASE*.

```
? (busca-patron '(b2 ? ?))
((b2 forma ortoedro) (b2 color rojo) (b2 tamaño pequeño)
 (b2 soporta-a b1) (b2 a-la-izquierda-de b3))
```

22. Usa BUSCA-PATRON con patrones apropiados para responder a las siguientes cuestiones:

- ¿De qué forma es el bloque B4?
- ¿Qué bloques son ortoedros?

- ¿Qué relación hay entre B2 y B3?
- ¿Cuáles son los colores de todos los bloques?
- ¿Qué se conoce sobre B4?
- ¿Qué bloques soportan a B1?
- ¿Qué bloques son soportados por B5?

23. Escribe una función que tome un nombre de un bloque como parámetro y devuelva un patrón preguntando el color del bloque.

24. Escribe una función que tome como entrada un bloque y devuelva una lista de los bloques que lo soportan. La función debe construir un patrón adecuado, hacer una búsqueda en la base de datos y eliminar la información que sobre quedándose sólo con los nombres de los bloques.

25. Escribe un predicado que tome un bloque como entrada y devuelva T si el bloque es soportado por un cubo o NIL si no.

26. Para escribir una función que haga una descripción de un bloque en una única lista de un nivel, haz primero una función ASERCIONES-DE-BLOQUE que tome un bloque como entrada y devuelva todas las aserciones que tengan que ver con él.

27. Escribe una función DESCRIPCION-DE-BLOQUE que llame a ASERCIONES-DE-BLOQUE y borre el nombre del bloque en cada elemento del resultado.

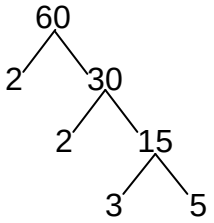
28. Escribe la función DESCRIPCION. Debe tener un argumento, llamar a DESCRIPCION-DE-BLOQUE, y combinar el restultado en una sola lista.

29. ¿Cuál es la descripción de B1? ¿Y la de B4?

30. El bloque B1 está hecho de madera, pero el bloque B2 está hecho de plástico. ¿Cómo podrías añadir esta información a la base de datos?

## 5. Otros Ejercicios.

31. Números primos son los divisibles sólo por sí mismos y por 1. Un número no primo, es un número compuesto, que puede factorizarse en primos. He aquí un árbol de factorización de un número no primo, obtenida con divisiones sucesivas por primos.



a) Escribe una definición recursiva para los números enteros positivos mayores que 1, en términos de los números primos.

```
? (numero '(2 2 3 5))
60
```

b) Realiza una función recursiva que, dado un número, devuelva una lista de los números primos de que consta su árbol de factorización.

```
? (factoriza 60)
(2 2 3 5)
```

32. Escribe una función **recursiva** que sume por un lado los números positivos y por otro lado los números negativos de una lista y devuelva ambas sumas en una lista de dos elementos.

```
? (suma-pos-y-neg '(1 -1 20 5 -16))
(26 -17)
```

Práctica
9

# Funciones como argumentos.

En esta práctica se abordan dos aspectos avanzados de la definición de funciones: cómo manejar argumentos que son funciones y cómo definir un número indeterminado de argumentos.

1. **Funciones como argumentos.**
2. **Funciones con número indeterminado de argumentos.**
3. **Ejercicios.**

## 1. Funciones como argumentos.

Las funciones de transformación de listas (p.e. mapcar) tienen una función como argumento (en inglés abreviado, *funargs*). Existen dos funciones en Common Lisp muy importantes, que permiten aplicar funciones a argumentos:

```
(FUNCALL #'funarg arg1 arg2 ... argN)
(APPLY #'funarg arglist)
```

FUNCALL aplica la función que se da como primer argumento al valor de los otros argumentos.

APPLY aplica la función que se da como primer argumento a los valores de la *lista de argumentos* proporcionada como último argumento. También puede tener argumentos intermedios, que son añadidos a la lista final antes de aplicar la función. Por tanto, también puede tener la forma:

```
(APPLY #'funarg arg1 arg2 ... argN arglist)
```

La principal diferencia es que en FUNCALL hay que proporcionar tantos argumentos adicionales como argumentos necesita la función, y en APPLY todos los argumentos se pueden sintetizar en una lista.

Las siguientes expresiones simbólicas son **equivalentes**:

```
(funcall #'append '(1 2) '(3 4))
(apply #'append '((1 2) (3 4)))
(apply #'append '(1 2) '(3 4) NIL)
(append '(1 2) '(3 4))
```

La ventaja de `apply` con respecto a `funcall` es que se puede aplicar una función de número de argumentos indeterminado a todos los elementos de una lista (de cualquier longitud). Por ejemplo:

```
? (APPLY #'+ '(1 1 0 2 2 2))
8
? (APPLY #'= '(1 1 1 1 1))
T
```

*Restricción*: No se puede aplicar funciones especiales ni macros con `apply` y `funcall`. Por ejemplo: `and, or, cond, if, do, let`... Se puede saber si un símbolo es una función especial, macro o función normal mirando la documentación.

Sin embargo, siempre existe la posibilidad de aplicar una función definida por el programador, bien como función aparte, bien como **expresión lambda**.

```
? (APPLY #'(lambda (x y) (and x y)) '(T NIL))
NIL
```

## 2. Funciones con número indeterminado de argumentos.

La expresión lambda anterior, imita el funcionamiento de un AND con sólo dos parámetros. Pero, ¿podemos definir funciones con número indeterminado de parámetros? Sí. Existe una palabra clave, `&REST`, que se utiliza en la definición de funciones para determinar que el resto de parámetros a dicha función se encuentra en una lista que puede contener un número de elementos indeterminado. `&REST` sólo puede aparecer al final de la lista de parámetros. No es que el último parámetro sea una lista, sino que todos los parámetros del final se engloban en una lista.

Veamos un ejemplo. Supongamos que se quiere definir una función similar a AND, que acepte cualquier número de parámetros, como mínimo 1, de forma que devuelva el "y" lógico de todos:

```
? (Y 'T 'T 'T 'T)
T
? (Y 'T 'NIL 'T)
NIL
? (Y 'NIL)
NIL
```

Si queremos obligar a que la función Y tenga al menos un argumento, hay que definirla con un argumento normal y el resto de argumentos se engloban en una lista:

```
(defun Y (param1 &rest restoparam)
  (cond ((null param1)     nil)
        ((null restoparam) param1)
        (T                 (apply #'Y restoparam))))
```

Nótese el uso de `apply` en la llamada recursiva. Averígüese la necesidad del uso de `apply` haciendo trazas de la función.

La función anterior no es una macro como `and`, de modo que sí se podría emplear en un `apply` o `funcall`.

```
? (APPLY #'Y '(T T T NIL T))
NIL
```

## 3. Ejercicios.

1. Dada una lista de listas (anidamiento en 2 niveles), escribe una función APLANA que devuelva la lista en un sólo nivel.

```
? (aplana '((2 3) (A B C) (permiso denegado)))
(2 3 A B C permiso denegado)
```

2. Dadas las definiciones, ¿son equivalentes las siguientes expresiones simbólicas? Si no lo son, razona por qué.

```
(defvar acciones '((comida comer) (bebida beber) (sonido oir)
                   (aroma oler) (imagen ver)))
(defvar arg1 '(list '+ 2 3))
(defvar arg2 (list '+ 2 3))
(defvar suma #'+)

a)(funcall #'+ 2 3)
  (eval (list '+ 2 3))

b)(funcall #'append '(perrito caliente) '(hamburquesa con queso))
  (apply #'append '(perrito caliente) '(hamburguesa con queso))

c)(funcall #'reverse '(reves al esta esto))
  (apply #'reverse '(reves al esta esto) NIL)

d)(funcall #'assoc 'comida acciones)
  (assoc 'comida acciones)

e)(funcall #'nth 1 acciones)
  (apply #'nth 1 acciones)

f)(eval arg1)
  (funcall arg1)

g)(eval arg2)
  (apply (first arg2) (rest arg2))

h)(funcall suma 2 3)
```

```
  (funcall #'+ 2 3)

i)(apply (first arg2) (rest arg2))
  (funcall #'+ 2 3)
```

3. Dada la siguiente definición:

```
? (defun saludos (s1 &REST s2)
   "Saludos desde Castellón"
     (list s1 s2))
SALUDOS
```

Rellena los huecos con las preguntas o respuestas adecuadas:

```
?
Saludos desde Castellón

? (saludos 'HOLA)


? (saludos 'HOLA 'COMO 'ESTAS)


?
((HOLA) (AMIGOS))
```

4. Haz una función que, dada una lista de nombres de funciones y dada una lista donde cada elemento es una lista de argumentos para cada una de esas funciones devuelva una lista de los resultados de aplicar cada función a sus argumentos. Se asume que el orden de cada función corresponde con el de sus argumentos y que no hay más funciones que listas de argumentos o viceversa. Por ejemplo:

```
? (ejecuta-lista '(reverse append cons first)
                 '(((A B C)) ((A) (B)) (A NIL) ((A B C))))
((C B A) (A B) (A) A)
```

5. Escribe el resultado de las siguientes expresiones simbólicas, teniendo en cuenta la siguiente definición:

```
(defun mi-funcion (elem &optional (tipo 'es-variable))
"Hace algo"
   (cond ((eq tipo 'es-funcion)   (documentation elem 'function))
         ((eq tipo 'es-constante) elem)
         (T                       (cons 'variable elem))))

? (mi-funcion 'mi-funcion 'es-funcion)


? (mi-funcion 'mi-funcion)


? (apply #'mi-funcion 'mi-funcion 'es-constante nil)


? (mi-funcion 'mi-funcion nil)
```

6. Teniendo en cuenta las siguientes definiciones:

```
(defun suma (&REST x)
```

```
"suma los elementos de una lista"
   (apply #'+ x))

(defvar nums '(1 2 3 4 5))
```

a) ¿Cuál es el resultado de las siguientes expresiones simbólicas?

```
? (documentation 'suma 'function)
? (documentation 'nums 'variable)
```

b) ¿Cuales de estas expresiones simbólicas es incorrecta? Pon `error` en las que creas que son incorrectas y `15` en las que la suma se realice correctamente.

```
? (apply #'suma 1 2 3 4 5)     ?     (apply #'suma nums)

? (suma 1 2 3 4 5)  ?     (suma nums)
```

7. La siguiente función tiene como parámetros un átomo y un número indeterminado de listas. La función intenta averiguar si el átomo se encuentra en alguna de las listas. Si está, devuelve la primera sublista donde se encuentre y si no, devuelve NIL.

```
(defun continente (arg1 &REST arglist)
   (cond ((null arglist) nil)
         ((member arg1 (first arglist)) (first arglist))
         (T (continente arg1 (rest arglist)))))
```

a) ¿Qué hay incorrecto en la función?.

b) ¿Qué línea cambiarías en la función para resolver el problema y cómo?

8. Haz una función recursiva que haga lo mismo que `remove-if-not`.

9. Escribe una función recursiva que haga lo mismo que `mapcar`, considerando que tiene como argumento una función de un sólo argumento y una sola lista.

10. Haz una función **recursiva** con los siguientes argumentos:

- una lista de elementos, *Lista*.
- una función de dos argumentos, *Funcion*, que devuelve T si el primer argumento ***es mejor que*** el segundo y NIL si no. *Funcion* se puede aplicar a 2 elementos cualesquiera de *Lista*.

La función debe devolver una lista con los mismos elementos que *Lista*, en la que ***el mejor*** de todos se haya colocado el primero. El resto de elementos es *indiferente* en qué posición queden.

Por ejemplo, si la lista es de números y el mejor es el más pequeño:

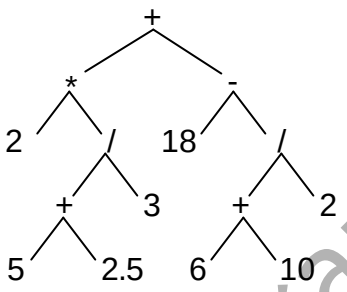
```
? (pon-el-mejor-el-primero '(3 4 5 6 7 8 1 9) #'<)
```

```
(1 3 4 5 6 7 8 9)
```

Otro ejemplo, si la lista contuviera sublistas y el mejor elemento es el de longitud más larga, podríamos definir una función que determine qué elemento es de longitud más larga y utilizarla para hallar el mejor:

```
? (defun mayor-longitud (lista1 lista2)
    (< (length lista1) (length lista2)))
? (pon-el-mejor-el-primero '((6 7) (3 4 5) (8 9 10 2) (1))
                           #'mayor-longitud)
((8 9 10 2) (6 7) (3 4 5) (1))
```

11. Elabora una función, que dado un árbol que representa una expresión aritmética, evalúe el resultado de dicha expresión. Supongamos, por ejemplo, la siguiente expresión aritmética:




Asumiremos que todos los operadores requieren dos argumentos. Si llamamos a la función EVALUA-EXP, quedaría así:

```
? (evalua-exp '(+ (* 2 (/ (+ 5 2.5) 3)) (- 18 (/ (+ 6 10) 2))))
15.0
```

Aunque la solución más trivial sería la siguiente:

```
(eval '(+ (* 2 (/ (+ 5 2.5) 3)) (- 18 (/ (+ 6 10) 2))))
```

resuelve el problema recursivamente, usando `funcall` o `apply`.

12. Se desea implementar un evaluador de expresiones matemáticas, capaz de manejar cualquier número de variables. Para ello, se dispone de una lista de variables con sus valores, como las del siguiente ejemplo:

```
((A . 5) (B . 3) (C . 10))
```

de forma que para evaluar una expresión matemática se toman los valores de las variables de esa lista y se calcula el resultado. Por ejemplo:

```
? (evalua-ecuacion  '(* (+ A 3) B)  '((A . 5) (B . 3) (C . 10)))
24
? (evalua-ecuacion  'B  '((A . 5) (B . 3) (C . 10)))
3
```

Escribe esta función haciendo las siguientes asumciones:

- Todas las variables que aparezcan en la expresión, están en la lista de variables.
- Todos los operadores que aparezcan en la expresión tienen dos operandos.

NOTA: Se recomienda utilizar recursividad.

13. Se desea elaborar una función que, dada una lista de ecuaciones, devuelva una lista de variables con los resultados de cada una. Por ejemplo, la siguiente secuencia de ecuaciones:

```
A ← 4          que vendria representada por: ((A . 4)
B ← 6                                         (B . 6)
C ← A + B                                     (C + A B)
A ← C                                         (A . C)
C ← (B * 2) + 3                               (C + (* B 2) 3))
```

Produciría la siguiente lista de resultados en las variables:

```
A = 10         que se representará por:    ((A . 10)
B = 6                                      (B . 6)
C = 15                                     (C . 15))
```

La lista de ecuaciones se proporciona como entrada a la función, de manera que el orden de las ecuaciones indica en qué orden han de calcularse. La lista de variables que debe devolver la función debe ser una lista de asociación, donde a cada variable se le asocia el valor resultante al final de realizar todas las ecuaciones. En este caso el orden es indiferente.

```
? (evalua-secuencia '((A . 4) (B . 6) (C + A B) (A . C)
                      (C + (* B 2) 3)))
((A . 10) (B . 6) (C . 15))
? (evalua-secuencia  '((P . 54))
((P . 54))
? (evalua-secuencia  '((P . 3) (P * P P))
((P . 9))
```

La función `evalua-expresion` del ejercicio anterior servirá para obtener el resultado de cada expresión que aparece en la lista de ecuaciones.

Práctica
10

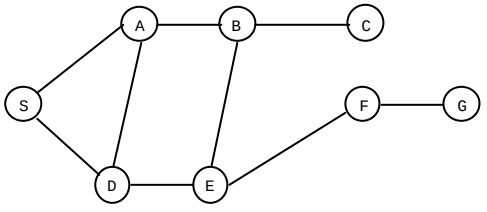
# El problema de la búsqueda en un grafo: métodos ciegos.

Esta práctica tiene como objetivo aplicar lo aprendido hasta el momento a un problema de Inteligencia Artificial que consiste en hallar un camino entre dos nodos de un grafo.

1. **Planteamiento del problema.**
2. **Representación de los datos.**
3. **Análisis de los métodos no informados.**
4. **Desglose en subproblemas.**

## 1. Planteamiento del problema.

Dado un grafo de nodos, p.e. A B C ... entre los que puede haber un camino o no, y dados dos nodos de ese grafo, p.e. S y G, se desea encontrar el camino más corto entre ambos nodos.




Hay dos problemas muy distintos según los datos de que dispongamos:

a) Disponiendo sólo del grafo.

b) Disponiendo del grafo y de información adicional sobre la distancia entre los nodos.

Si sólo disponemos del grafo, el problema se reduce a encontrar un camino, pues no podemos saber si un camino es mejor que otro. Estos métodos se llaman ***ciegos*** o ***no informados***.

Si disponemos de más información, como la distancia entre los nodos, la búsqueda de la solución se puede mejorar al poder ordenar los caminos según un criterio heurístico. Estos métodos se llaman ***informados*** o ***heurísticos***.

En esta práctica vamos a ver solamente los métodos no informados.

## 2. Representación de los datos.

¿Qué datos tenemos sobre el grafo? Los nodos que lo componen y las conexiones entre ellos. Para la próxima práctica, también necesitaremos saber otro tipo de información, como puede ser las coordenadas del nodo, etc.

¿Cómo podemos representar esto en forma de listas? Por ejemplo, una lista de nodos, para cada uno de los cuales hay una lista de asociación que contiene los vecinos de ese nodo y cualquier otra información que nos pudiese ser útil. El grafo del ejemplo anterior se podría representar así:

```
;lista de nodos con sus características
((S  (coordenadas 0 3)  (vecinos a d))
 (A  (coordenadas 4 6)  (vecinos s b d))
 (B  (coordenadas 7 6)  (vecinos a c e))
 (C  (coordenadas 11 6) (vecinos b))
 (D  (coordenadas 3 0)  (vecinos s a e))
 (E  (coordenadas 6 0)  (vecinos b d f))
 (F  (coordenadas 11 3) (vecinos e g))
 (G  (coordenadas 15 3) (vecinos f)) )
```

Otra cuestión importante es si vamos a poner este grafo en una variable global a la que hagan referencia todas las funciones del programa o lo pasaremos por parámetros todas las veces que haga falta.

Para realizar esta práctica, asumiremos que el grafo se encuentra en una variable global llamada `*grafo*`.

## 3. Análisis de los métodos no informados.

La búsqueda de un camino desde un nodo raíz (en el ejemplo, S) a un nodo meta (en el ejemplo F), se basa en explorar todos los caminos posibles desde el nodo raíz para ver si alguno de ellos nos lleva al nodo meta.

Todos los métodos se basan en mantener una **lista de caminos posibles** que parten del nodo raíz. Cada vez se expande un paso más en uno de los caminos hasta que ocurre que ese camino lleva a algun sitio por donde ya se ha pasado (bucle) y se descarta, o bien lleva a la meta, con lo que ya se ha encontrado el camino final. Si al ir descartando caminos posibles, nos quedamos sin ninguno, quiere decir que no hay camino posible del nodo raiz al nodo meta.

Una lista de caminos posibles sería por ejemplo:

```
((E B A S) (D A S) (D S))
```

que significa que hasta el momento, hemos abierto los caminos S-D, S-A-D y S-A-B-E y que puedo seguir explorando todos los caminos que sean continuación de ellos.

***Búsqueda en Profundidad***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - Expandir el primer camino de la lista, descartando todos los caminos que generen bucles.
   - Añadir los nuevos caminos, si hay, **al principio** de la lista, y quitar el camino que ya ha sido expandido.
3) Si se encontró el nodo meta, éxito, si no, fracaso.

***Búsqueda en Anchura***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - Expandir el primer camino de la lista, descartando todos los caminos que generen bucles.
   - Añadir los nuevos caminos, si hay, **al final** de la lista, y quitar el camino que ya ha sido expandido.
3) Si se encontró el nodo meta, éxito, si no, fracaso.

***Búsqueda No Determinista***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - Expandir el primer camino de la lista, descartando todos los caminos que generen bucles.
   - Añadir los nuevos caminos, si hay, **en posiciones aleatorias** de la lista, y quitar el camino que ya ha sido expandido.
3) Si se encontró el nodo meta, éxito, si no, fracaso.

## 4. Desglose en subproblemas.

En primer lugar hay que analizar qué datos vamos a necesitar del grafo, y realizar las *funciones de acceso* necesarias para obtenerlos. Así pues, necesitaremos diversas funciones:

- `coordenadas`, para extraer las coordenadas de un nodo del grafo.
- `vecinos`, para extraer los vecinos de un nodo del grafo.

Lo más importante del análisis de los 3 métodos es que los tres son casi idénticos excepto en la forma de añadir los nuevos caminos a la lista de caminos por explorar (en negrita). Por tanto, podemos realizar funciones idénticas aplicables a los tres métodos y la única función diferente es la que añade los nuevos caminos.

Funciones idénticas para los tres métodos:

- `inicia-lista`, construye una lista formada por un sólo camino abierto que incluya únicamente el nodo raíz (nodo de partida).

  *Datos necesarios:* el nodo raiz.

  *Resultado*: una lista que contiene una lista con sólo el nodo raíz.

- `expande-camino`, dado el primer camino de la lista, lo explande, y obtiene todos los nuevos caminos a partir de él, después descarta todos los caminos que generen bucles, y devuelve los que queden.

  *Datos necesarios*: un camino.

  *Resultado*: lista de nuevos caminos que no generen bucles.

  *Realización*: Esta función se puede desglosar en las dos siguientes: `halla-nuevos-caminos` y `descarta-bucles.`

- `halla-nuevos-caminos`, dado el primer camino de la lista, lo explande, y obtiene todos los nuevos caminos a partir de él.

  *Datos necesarios*: un camino.

  *Resultado*: una lista de nuevos caminos formados a partir del camino inicial.

  *Realización*: Cada nuevo camino se forma añadiendo un vecino del último nodo del camino.

- `descarta-bucles`, descarta todos aquellos caminos que generen bucles.

  *Datos necesarios*: lista de nuevos caminos.

  *Resultado*: lista de nuevos caminos excluyendo los caminos con bucles.

  *Realización*: Como se descartan bucles cada vez que se expande un camino, sólo hay que mirar que el último nodo añadido no se encuentre ya en el camino.

- `camino-completo-p`, determina si hemos encontrado el camino al nodo meta.

  *Datos necesarios*: un camino y el nodo meta.

  *Resultado*: T o NIL según si ese camino conduce al nodo meta o no.

  *Realización*: Ver si el último nodo del camino es el nodo meta.

Funciones distintas para cada método:

- `añadir-al-principio`, concatena el segundo argumento con el primero.
- `añadir-al-final`, concatena el primer argumento con el segundo.
- `añadir-aleatoriamente`, inserta cada uno de los elementos de la segunda lista en posiciones aleatorias de la primera.

En las tres funciones:

*Datos necesarios*: lista de caminos por expandir y lista de nuevos caminos.

*Resultado*: lista de caminos unión de ambas según el caso.

*Realización*: las dos primeras son triviales, y la tercera puede necesitar usar funciones auxiliares para insertar un camino nuevo en una posición aleatoria de la primera lista (usar `random`).

Una vez realizadas todas estas funciones de ayuda, debe resultar sencillísimo realizar las funciones `busqueda-en-profundidad`, `busqueda-en-anchura` y `busqueda-no-determinista`. Las tres funciones tienen como datos el nodo raíz y el nodo meta. Como resultado deben devolver NIL si no hay camino entre ambos y en caso de que lo haya, el camino encontrado.

Práctica
11

# El problema de la búsqueda en un grafo: métodos informados.

En esta práctica se aplica lo estudiado con el fin de implementar los métodos heurísticos o informados para la resolución del problema de la búsqueda en un grafo.

1. **Análisis de los métodos informados o heurísticos.**
2. **Desglose en subproblemas.**
3. **Otro ejemplo**

## 1. Análisis de los métodos informados o heurísticos.

Estos métodos necesitan de una medida heurística de la distancia que falta para llegar a la meta dado un camino parcial. Esta medida se puede extraer por ejemplo de la distancia en línea recta entre un nodo cualquiera y la meta. Teniendo en cuenta la misma representación de los datos (grafo y lista de caminos abiertos) vista en la sesión anterior proponemos el análisis de los siguientes métodos:

***Búsqueda mediante el método de la Escalada***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - Expandir el primer camino de la lista, descartando todos los caminos que generen bucles.
   - Añadir los nuevos caminos, si los hay, mediante el siguiente método, y quitar el camino que ya ha sido expandido:
     - ***Ordenar los nuevos caminos*** *de menor a mayor, según la distancia estimada entre sus nodos terminales y la meta.*

- *Añadir los nuevos caminos ordenados **al principio** de la lista.*

3) Si se encontró el nodo meta, éxito, si no, fracaso.

***Búsqueda en Haz (n)***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - *Expandir **todos los caminos** de la lista a la vez, descartando todos los caminos que generen bucles.*
   - *Reunir los nuevos caminos, si los hay, en una nueva lista de caminos.*
   - ***Ordenar la lista de caminos** de menor a mayor, según la distancia estimada entre sus nodos terminales y la meta.*
   - ***Tomar los** **n** **primeros** de la lista, si los hay, y eliminar los restantes.*
3) Si se encontró el nodo meta, éxito, si no, fracaso.

***Búsqueda mediante el método de Primero El Mejor***

1) Construye una lista formada sólo por el camino (nodo-raiz).
2) Repetir hasta que el primer camino de la lista esté completo o la lista esté vacía:
   - Expandir el primer camino de la lista, descartando todos los caminos que generen bucles.
   - Añadir los nuevos caminos, si los hay, mediante el siguiente método, y quitar el camino que ya ha sido expandido:
     - *Añadir los nuevos caminos a la lista (no importa el método).*
     - ***Ordenar la lista completa de caminos** según la distancia estimada entre sus nodos terminales y la meta, de menor a mayor.*
3) Si se encontró el nodo meta, éxito, si no, fracaso.

# 2. Desglose en subproblemas.

De los tres métodos, el **método de la escalada** y el **método de primero el mejor** son similares a los vistos en la sesión anterior. La diferencia está en la forma de añadir los nuevos caminos a la lista de caminos abiertos (en cursiva), ya que se realiza algún tipo de ordenación que mejore la búsqueda del camino.

Por tanto, para añadir los nuevos caminos a la lista de caminos abiertos, hay que tener en cuenta la distancia de estos caminos al nodo meta:

- Para búsqueda en escalada: `añadir-ordenado-al-principio`

  *Datos necesarios*: lista de caminos abiertos, lista de nuevos caminos, y nodo meta.

*Resultado*: lista de caminos resultante de añadir al principio de la lista de caminos abiertos la lista de nuevos caminos ordenada según su distancia al nodo meta.

- Para primero el mejor: `añadir-primero-y-ordenar-despues`

  *Datos necesarios*: lista de caminos abiertos, lista de nuevos caminos, y nodo meta.

  *Resultado*: lista de caminos resultante de concatenar ambas listas y ordenar a continuación según su distancia al nodo meta.

Para hacer las funciones anteriores, hay que definir las siguientes funciones que se emplearán en ambas y servirán para simplificarlas:

- `ordena-lista-según-distancia`, dada una lista de caminos abiertos, la devuelve ordenada según la distancia estimada entre cada camino y el nodo meta.

  *Datos necesarios:* una lista de caminos a ordenar, el nodo meta.

  *Resultado*: una lista con los mismos caminos, ordenada.

  *Realización*: Calcular la distancia entre un camino abierto y la meta, que es la distancia entre el último nodo de ese camino y la meta. Esta función debería basarse en una función `distancia-entre-nodos`, que se explica a continuación.

- `distancia-entre-nodos`, dados dos nodos, calcula la distancia entre ellos como la distancia en línea recta entre sus coordenadas.

  *Datos necesarios*: dos nodos.

  *Resultado*: distancia entre ellos.

  *Realización*: Calcular la distancia entre sus coordenadas.

Las funciones `busqueda-en-escalada`, y `busqueda-1o-el-mejor` se construyen de forma similar a las de la sesión anterior. Al igual que ellas tienen como datos el nodo raíz y el nodo meta, y como resultado deben devolver NIL si no hay camino entre ambos y en caso de que lo haya, el camino encontrado.

Sin embargo, el **método de búsqueda en haz** requiere cambiar en algo el algoritmo por dos razones:

1) No se expande uno a uno los nodos y se añaden los nuevos caminos cada vez, sino que se expanden todos los nodos a la vez. Esto equivale a que cada vez se expanden todos los nodos correspondientes a un mismo nivel del árbol de búsqueda.
2) De la lista de caminos abiertos en cada nivel nos quedamos con los *n* mejores (los *n* más cercanos al nodo meta), luego *n* ha de ser un parámetro más de la búsqueda.

Por tanto, para hacer esta función, además de las funciones de cálculo de distancia y ordenación y las funciones de la práctica anterior, necesitaremos también otras:

- `selecciona-n-mejores`, dada una lista de caminos, se queda con los *n* primeros caminos y elimina todos los demás.

*Datos necesarios*: una lista de caminos, el nodo meta, el parámetro *n*.

*Resultado*: una lista de como máximo *n* elementos (los *n* primeros) con los *n* caminos más cercanos al nodo meta ordenados.

- `expande-nivel`, dada una lista de caminos, expande todos ellos y devuelve una lista con todos los nuevos caminos, excepto aquellos que generen bucles.

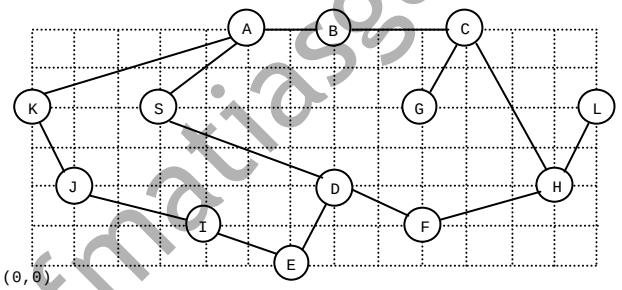
  *Datos necesarios*: una lista de caminos.

  *Resultado*: la concatenación de los resultados de expandir todos los caminos de la lista.

La función `busqueda-en-haz` se construye teniendo en cuenta que se han de expandir a la vez todos los caminos abiertos hasta el momento, para obtener los caminos correspondientes al siguiente nivel. Después han de ordenarse y quedarse sólo con los n mejores.

## 3. Otro ejemplo.

Se propone el siguiente ejemplo a resolver mediante los seis métodos.



Práctica
12

# Repaso. Ejercicios generales.



## 1. Ejercicios generales.

1. Dibuja las celdas cons de las siguientes listas y su contenido:

```
(A B (C . D))

((A B . C))

((((FIN))))

((UNO . 1) (DOS . 2) (TRES . 3) . FIN)

(A (B) NIL)

((PI 3.1416) (G 9.2) . CTES)
```

2. Haz una función **recursiva** que, dado un número entero y una lista de números enteros ordenada de menor a mayor, devuelva una lista similar a la anterior en la que se ha insertado el número de forma que la lista permanezca ordenada.

3. ¿Cuál será el resultado de las siguientes expresiones simbólicas?

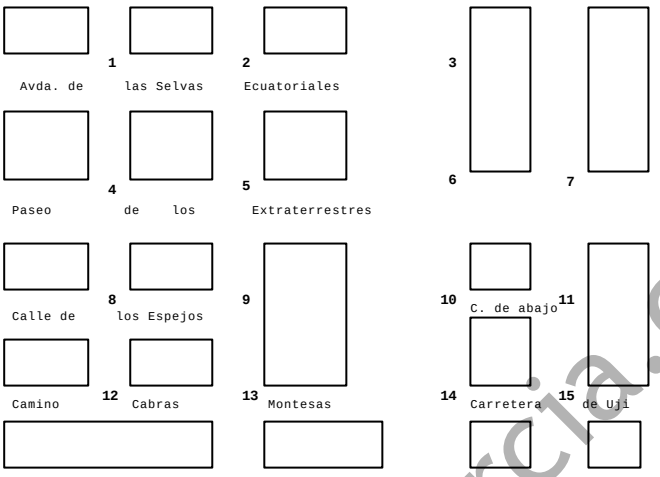
```
? (eval ''(pairlis '(CLAVOS TUERCAS TORNILLOS) '(43 12 35)))

? (progn (setq A '(a b c d))
         (append A '(e f g))
         A)

? (let ((a '(1 2 3))
        (b '(a b c)))
       (list (nconc a b) (nconc b a)))
```

```
? (do ((n     8         (- n 1))
       (equis (list 'X) (cons 'X equis))
      ((zerop n)
       equis)
      (format T "~A~A " (first equis) n))
```

4. Se dispone de una base de datos de las calles de una ciudad como la que se ve en el ejemplo:




En dicha base de datos aparecen todas las calles, con su nombre, el tipo de calle y una lista de cruces que componen dicha calle (los identificadores de cruce elegidos en este caso son números). En el ejemplo:

```
((SELVAS-ECUAT (nombre "de las Selvas Ecuatoriales") (tipo AVENIDA)
               (cruces 1 2 3))
 (EXTRATERRESTRES (nombre "de los Extraterrestres")(cruces 4 5 6 7)
               (tipo PASEO))
 (ESPEJOS (nombre "de los Espejos") (tipo CALLE) (cruces 8 9))
 (CABRAS-MONTESAS (nombre "Cabras Montesas") (tipo CAMINO)
               (cruces 12 13 14))
 (UJI (nombre "de Uji") (tipo CARRETERA) (cruces 14 15))
 (ABAJO (nombre "de Abajo") (tipo CALLE) (cruces 10 11))
 (PINTOR-TRAPEZOIDE (nombre "Pintor Trapezoide") (tipo CALLE)
               (cruces 1 4 8 12))
 (BLANCO (nombre "Blanco") (cruces 2 5) (tipo PASEO))
 (VERDE (nombre "Verde") (cruces 8 12) (tipo PASEO))
 (SUBMARINO-AMAR (tipo GRAN-VIA) (nombre "del Submarino Amarillo")
               (cruces 3 6 10 14))
 (COCOTEROS (nombre "de los Cocoteros") (cruces 7 11 15) (tipo
AVENIDA))
 ...
 )
```

Sabiendo que la lista de calles está almacenada en una variable global llamada `*callejero*`, haz una función que, dadas dos calles cualesquiera de la ciudad, *calle1* y *calle2*, devuelva:

- NIL, si no hay ningún cruce entre ambas.
- ERROR, si no encuentra alguna de las calles en la base de datos.
- El identificador del cruce, si lo hay (sólo puede haber uno).

5. Usando la variable global `*callejero*` del ejercicio anterior, haz una función que dado un tipo de calle (avenida, gran vía, calle, camino, etc.) extraiga una lista de todos los nombres de calles de ese tipo mediante **transformaciones** de listas y/o **filtros**.

6. Dada la siguiente definición:

```
? (defun monstruo_galletas (&OPTIONAL (galletas NIL))
  "El monstruo de las galletas"
    (if (null galletas)
        'TENGO_HAMBRE
        (progn (format T "COME ~A~%" (first galletas))
               (if (eq 'GALLETA (first galletas))
                   (cons 'RICA (monstruo_galletas (rest galletas)))
                   (cons 'PUAG (monstruo_galletas (rest galletas)))
                                                              ))))
MONSTRUO_GALLETAS
```

   Rellena los huecos con las preguntas o respuestas adecuadas:

```
?
El monstruo de las galletas

? (monstruo_galletas)


? (monstruo_galletas '(galleta))


? (monstruo_galletas '(galleta chorizo))
```

7. En un periódico se guarda la cartelera de cines de la ciudad en una estructura de datos donde para cada cine se sabe:
   - nombre (que sirve al mismo tiempo de identificativo),
   - dirección (en un sólo string literal, p.e: "C/. Enmedio, 18"),
   - cuántas salas tiene, mediante una lista con los nombres de las salas (números o no).
   - qué película se proyecta en cada sala, mediante una lista con las películas en el mismo orden que la lista de las salas.
   - horario de proyecciones, si lo hay.
   - precio, si lo hay.
   - día del espectador, si lo hay.

   Véase el siguiente ejemplo de una cartelera:

```
((CLASICO
  (DIRECCION . "C/Las Prodigas, 32")
  (SALAS ATENEA ARTEMISA)
  (PELICULAS (LO QUE EL VIENTO SE LLEVO) (ESPARTACUS))
  (PROYECCIONES (11 0) (6 0))
  (PRECIO . 350)
```

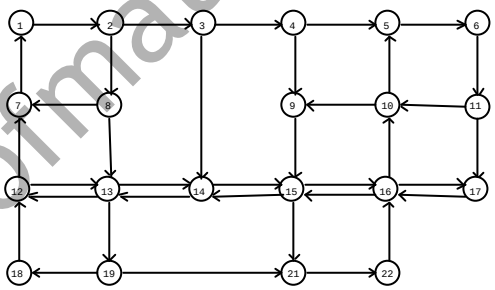
```
  (DIA_ESPECTADOR . MIERCOLES))
(ALEJANDRIA
 (PELICULAS (LA MASCARA) (FORREST GUMP) (DOS BOBOS MUY BOBOS)
        (PULP FICTION) (WYATT EARP) (EL PIANO) (EL REY LEON))
 (SALAS 1 2 3 4 5 6 7)
 (PROYECCIONES (16 0) (19 30) (23 0))
 (DIRECCION . "C/Alejandría, 24")
 (PRECIO . 600))
...
)
```

Utilizando transformaciones de listas, haz una función que dada la cartelera extraiga una lista de asociación con sólo los nombres y direcciones de aquellos cines que tengan día del espectador. En el ejemplo anterior, el resultado sería:

```
((CLASICO . "C/Las Prodigas, 32")
   ...)
```

8. Haz una función que, basándose en la cartelera del ejercicio anterior, sustituya las listas de salas y películas por una lista de asociación en la que se asocie cada sala con su película, encabezada con la clave SALAS-PELICULAS, y devuelva la nueva cartelera completa. Usa al menos una transformación de listas.

9. Se desea controlar el tráfico de una ciudad y para ello se dispone de un *grafo* cuyos nodos son los **cruces** de calles y cuyos arcos son **tramos de calle**. El sentido de los arcos determina el sentido de **circulación** de la calle y por tanto entre dos nodos puede haber hasta dos arcos, uno de ida y otro de vuelta.




La información sobre cada cruce incluye: los cruces hacia los que se puede dirigir un coche desde cada cruce, la media de vehículos que pasan al día en vehículos/minuto y la velocidad media de los vehículos al pasar por dicho cruce en kilómetros/hora. La representación en listas del ejemplo de la figura será como sigue:

```
(defvar *TRAFICO*
   '(( 1 (siguientes 2)    (media 15) (velocidad 30))
     ( 7 (siguientes 1)    (media 20) (velocidad 27))
     (12 (siguientes 7 13) (media 23) (velocidad 45))
     (18 (siguientes 12)   (media  2) (velocidad 35))
     ... )
   "Lista de cruces con la información de siguientes cruces, número
   medio de vehículos y velocidad media de circulación.")
```

Haz una función que, dados dos cruces, determine si están conectados directamente mediante un sólo tramo de calle. Si lo están, además debe decir si ese tramo de calle es unidireccional o bidireccional. El resultado debe darse en el siguiente formato:

```
? (tipo-tramo '1 '2)
(TRAMO UNIDIRECCIONAL)
? (tipo-tramo '2 '1)
(TRAMO UNIDIRECCIONAL EN SENTIDO INVERSO)
? (tipo-tramo '13 '12)
(TRAMO BIDIRECCIONAL)
? (tipo-tramo '12 '13)
(TRAMO BIDIRECCIONAL)
? (tipo-tramo '1 '12)
(NO HAY TRAMO)
```

10. Dada la estructura `*TRAFICO*` del ejercicio anterior, realiza una función que devuelva la velocidad media ponderada de todos los cruces mediante los siguientes pasos:
    - Extraer en una lista las cantidades medias de vehículos de cada cruce.
    - Extraer en otra lista las velocidades medias de cada cruce.
    - Calcular en una lista el producto `nº-vehículos * velocidad-media` para cada cruce.
    - Sumar el total de resultados parciales de la lista y dividir por la cantidad total de vehículos.

11. Haz una función recursiva que dada una lista y dado un dato de la lista de cualquier tipo, devuelva la posición del dato en la lista. Las posiciones se numeran desde 1 hasta la longitud de la lista, y si devuelve 0 significa que el dato no se encuentra en la lista.

```
? (posicion '(A) '(12 (A) B))
2
? (posicion 'A '(12 (A) B))
0
```

12. ¿Cuál será el resultado de las siguientes expresiones simbólicas?

```
? (eval (list 'second ''(43 12 35)))


? (rest '(ALGO . MAS))


? (let ((numeros  '(1 2 3))
        (al-reves (nreverse numeros)))
       numeros)


? (progn (setq numeros '(1 2 3 4))
         (cons '0 numeros)
         numeros)


? (mapcan #'(lambda (x) (if (atom x) nil x)) '((a) b (c d)))
```

```
? (maplist #'reverse '(1 2 3))
```

13. Se quiere escribir un programa para las conserjerías de la universidad de manera que los bedeles pueden registrar en el ordenador las reservas de aulas, despachos o seminarios que se pidan. Dichas reservas se guardan en una lista de reservas, cada una de las cuales se guarda como una lista de:

```
(numero-de-reserva (TIPO tipo-de-servicio numero-despacho)
                   (DIA dia mes año)
                   (HORARIO horas-de-reserva)
                   (RESPONSABLE nombre-persona-responsable))
```

Por ejemplo:

```
((1298 (TIPO aula D205) (DIA 7 Febrero 1996) (HORARIO 16 20)
       (RESPONSABLE "Eduardo Manostijeras"))
 (1299 (TIPO seminario D403) (RESPONSABLE "Julio Cesar")
       (DIA 1 Marzo 1996) (HORARIO 12 14))
 (1343 (TIPO despacho D501) (RESPONSABLE "Juan de Juan")
       (HORARIO 12 14) (DIA 1 Marzo 1996))
...)
```

Sabiendo que el orden de los datos puede ser cualquiera, realiza usando **transformaciones de listas** y/o **filtros**:

a) una función que devuelva si un despacho tiene o no alguna reserva hecha.

b) una función que dado un día, mes y año, devuelva una lista con una sublista por cada reserva que haya para ese día del tipo `(numero-despacho horas-de-reserva)`.

14. Para una aplicación gráfica, se dispone de una lista de puntos en el plano (*X Y*). Se quiere extraer parejas de puntos que tengan la misma *Y*. Haz una función recursiva en Lisp teniendo en cuenta que si un valor determinado de *Y* aparece:

- En dos puntos, éstos formarán una pareja.
- Solo en un punto, éste no tiene pareja y por tanto no aparecerá en el resultado.
- En más de dos puntos, se emparejan de dos en dos, de manera que si el número de puntos es impar, el último no tiene pareja.

Por ejemplo:

```
? (empareja '((23 7) (8 3) (10 3) (22 7) (8 1) (21 7)))
(((23 7) (22 7)) ((8 3) (10 3)))
```

Nótese que el punto `(22 7)` no se vuelve a emparejar con `(21 7)`, pues sólo puede aparecer una vez cada punto en el resultado.

15. Para hacer un programa que juegue a "Hundir la flota" se necesita una función que devuelva la lista de todas las casillas del juego, dependiendo del tamaño del tablero. Por ejemplo, en un tablero de 4x3 y numerando las casillas con letras y números:

```
? (todas-casillas '(A B C D) '(1 2 3))
((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3) (D 1) (D 2)
(D 3))
```

Escribe dicha función, sabiendo que no importa el orden en que se devuelvan las casillas.

16. Escribe una función recursiva que haga exactamente lo mismo que MAPCAR, en el caso de que sólo se aplique a una lista.

17. Escribe una función recursiva que haga exactamente lo mismo que MAPCAR, en el caso de que se aplique a dos listas. Recuerda que la función MAPCAR realiza tantas operaciones como elementos tenga la lista más corta.

18. Escribe una función que haga exactamente lo mismo que MAPCAR, en el caso de que se aplique a cualquier número de listas. Recuerda que la función MAPCAR realiza tantas operaciones como elementos tenga la lista más corta. Asume que cuando no se le pase ninguna lista la función devuelva NIL. Se puede usar la función definida en el ejercicio 4.

19. Escribe una función que, dado un número indeterminado de parámetros, devuelva una lista con todas las parejas que se puedan hacer con ellos en el mismo orden en que aparecen. Si el número de parámetros es impar, el último elemento NO deberá aparecer en la lista, quedándose sin "emparejar".

```
? (emparejar 'sanson 'dalila 'pedro 'vilma 'pablo 'betty
  'freddy)
((SANSON DALILA) (PEDRO VILMA) (PABLO BETTY))

? (emparejar 'ajo 'cebolla 'limon 'pomelo)
((AJO CEBOLLA) (LIMON POMELO))

? (emparejar 'naranja)
NIL

? (emparejar)
NIL
```

20. Escribe otra función que empareje de la siguiente manera: todos los elementos están emparejados dos veces, con el anterior y con el posterior, excepto el primero y el último que sólo se emparejan con una.

```
? (emparejar-2 'castello 'xilxes 'sagunt 'valencia)
((CASTELLO XILXES) (XILXES SAGUNT) (SAGUNT VALENCIA))
? (emparejar-2 'castello 'vinaroz)
((CASTELLO VINAROZ))
```

Pero esta vez obligaremos a que al menos deben aparecer 2 parámetros.

21. Haz una función recursiva que, dada una lista de átomos y un predicado de tres argumentos, aplique la función a los elementos consecutivos de la lista tomados de tres en tres de manera que si algún grupo de tres no satisface el predicado devuelva NIL y si todos lo satisfacen, devuelva T. Si la lista tiene menos de tres argumentos, ha de devolver NIL.

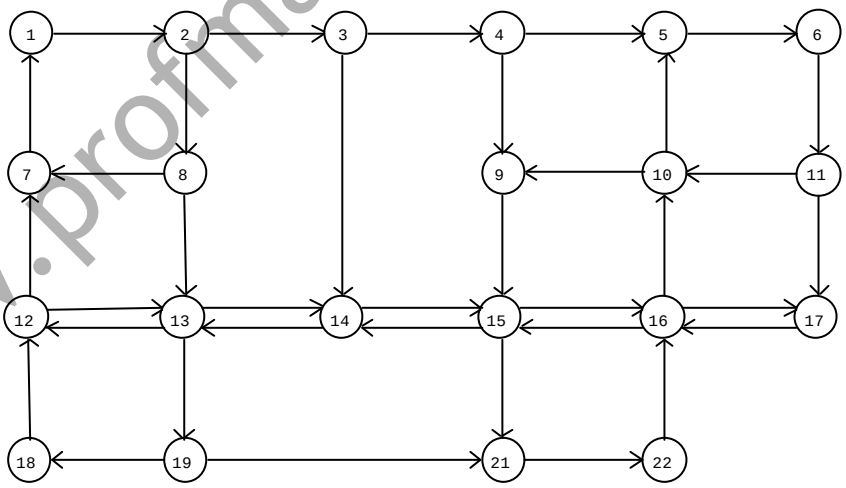
```
? (defun funcioncilla (a b c) (and (= b (+ a 1)) (= c (+ b 1))))
FUNCIONCILLA
? (chequea-lista #'funcioncilla '(1 2 3 4 5 6))
T
? (chequea-lista #'funcioncilla '(14 15 14 17))
NIL
? (chequea-lista #'(lambda (x y z) (eq z (+ x y)))
                 '(1 2 3 5 8 13 21))
T
```

22. ¿Por qué expresiones más del estilo de programación Lisp sustituirías las siguientes?

```
? (progn (setq A 'HOLA)
         (setq B 'AMIGOS)
         (list A B))

? (do ((a 10))
      ((zerop a)
       'TERMINADO)
      (format T "Iteracion ~A~%" a)
      (setq a (- a 1)))
```

23. Se desea controlar el tráfico de una ciudad y para ello se ha creado un grafo cuyos nodos son los cruces de calles y cuyos arcos son tramos de calle. El sentido de los arcos determina el sentido de circulación de la calle y por tanto entre dos nodos puede haber hasta dos arcos, uno de ida y otro de vuelta. Ved el ejemplo de la figura:




Los nodos están numerados y la representación del grafo en listas será como sigue:

```
((1 (siguientes 2))      (2 (siguientes 3 8))       ...
 (7 (siguientes 1))      (8 (siguientes 7 13))      ...
 (12 (siguientes 7 13))  (13 (siguientes 12 14 19)) ... ))
```

Haz una función que dada una lista de cruces (un camino) determine si todos los cruces son consecutivos y si la dirección es correcta. Si todo es

correcto, debe devolver `T` y si algo falla, debe devolver cuál es el fallo: `NO-CONSECUTIVO`, `DIRECCION-CONTRARIA`.