APÉNDICE: EJERCICIOS DE PROGRAMACIÓN

JESÚS GONZÁLEZ BOTICARIO

EJERCICIOS DE LISP

1. Ejercicios Básicos de Lisp

1.1 Definir una función que calcule el valor de:

$$F = \frac{1}{\sqrt{ax^2 + bx^2 + c}}$$

1.2 Definir una función que devuelva la longitud de una circunferencia, dando como parámetro el radio R de la misma siendo

$$L = 2\rho R$$

1.3 Definir una función que pase de grados centígrados a grados Fahrenheit, sabiendo que:

$$F = (C + 40) \times 1.8 - 40$$

1.4 Definir una función que, dados tres argumentos numéricos, devuelva cuál es el mediano, utilizando MAX y MIN.

1.5 Definir un predicado que dados A, B y C como argumentos devuelva T si $\mathbf{B^2 - 4AC}$ es menor que cero.

1.6 Definir un predicado que devuelva T si alguno de sus dos primeros argumentos es menor que el tercero y mayor que el cuarto.

1.7 Definir una función que calcule la entropía de un suceso aleatorio que representa k modos de realización de probabilidades $P_1, P_2, P_3, \ldots, P_k$, cuyo valor viene dado por la expresión:

$$H = \sum_{i=1}^{K} p_i \log p_i$$

1.8 Asignar a la variable X1 la lista (COCHE MOTO TREN) y a la variable X2 la lista (EDUARDO PEDRO ANTONIO).

a) Concatenar las dos listas y calcular la longitud de la lista resultante.

b) Construir una lista cuyos elementos sean los elementos finales de X1 y X2.

c) A partir de X1 y X2, construir las listas:

(TREN ANTONIO)

((TREN) (ANTONIO))

((TREN) ANTONIO)

d) Concatenar X1 con el inverso de X2 y asignar el resultado a X3.

1.9 Definir una función que tenga por argumento una lista y devuelva el tercer elemento de dicha lista.

1.10 Definir una función que tenga por argumento una lista y devuelva otra lista con el primer y último elemento de la lista.

1.11 Definir un predicado con tres argumentos: un átomo y dos listas. El predicado debe devolver T si el átomo pertenece a las dos listas.

1.12 Definir ROTAIZQ, un procedimiento que recibe una lista como argumento y devuelve otra en la que el primer elemento pasa a ser el último y todos los demás ocupan una posición más a la izquierda.

Ejemplo:

```
> (ROTAIZQ  '( A B C ))

 (B  C  A)
```

Definir también ROTADCHA que realiza la operación inversa.

1.13 Un palíndromo es una lista que tiene la misma secuencia de elementos cuando se lee de izquierda a derecha que cuando se hace de derecha a izquierda. Definir un función PALINDROMO que tome una lista como argumento y devuelva su palíndromo. Definir también PALINDROMOP, un predicado que comprueba si la lista que se pasa como argumento es un palíndromo.

1.14 Definir una función que dados tres números X, Y y Z, devuelva una lista con los números ordenados por orden creciente.

1.15 Definir una función que tomando como argumentos una lista y un elemento, devuelva T si el elemento aparece más de una vez en la lista.

1.16 Definir una función que devuelva el número de átomos que hay en una lista situados a la izquierda de un átomo determinado de dicha lista.

1.17 Definir una función que añada un elemento a una lista en caso de que aquel no se encuentre en ésta.

1.18 Definir la función CLIMA que, recibiendo como parámetro la temperatura en un recinto, devuelva:

a) HELADO si la temperatura es menor de 0 grados.

b) FRIO si está entre 0 y 10.

c) CALIDO si está entre 10 y 20.

d) SOFOCANTE si está entre 20 y 30.

e) ABRASIVO si es mayor de 30 grados.

1.19 Definir un predicado que tome tres argumentos: día, mes y año, y devuelva T si es una fecha válida.

```
(FECHAP 12 12 1986) => T

(FECHAP 12 30 1987) => NIL

(FECHAP 31 2 1986) => NIL

(FECHAP 31 11 1876) => T
```

1.20 Definir una función que devuelva cierto (T) si alguno de sus tres argumentos no es divisible por 2.

1.21 Definir la función ABSOLUTO que calcula el valor absoluto de su argumento si éste es un número utilizando una estructura condicional.

1.22 Definir la función MINIMO, que devuelve el mínimo de sus tres argumentos numéricos, utilizando una estructura condicional.

1.23 Definir la función MAXIMO, que devuelve el máximo de sus tres argumentos numéricos, utilizando una estructura condicional.

1.24 Definir, utilizando una estructura condicional, las funciones lógicas OR y AND para tres argumentos.

1.25 Definir de forma iterativa la función SUMA de dos números.

1.26 Definir de forma iterativa la EXPONENCIACION de dos números

1.27 Definir la función FACTORIAL de forma recursiva e iterativa sabiendo que el factorial de N es 1 si N es 0 y en otro caso será N veces el factorial de N-1

1.28 Redefinir de forma recursiva e iterativa las funciones primitivas MEMBER, REVERSE y LENGTH.

1.29 Definir una función que determine la profundidad de una lista (número máximo de paréntesis que hay que extraer para obtener un determinado elemento de la misma).

1.30 Definir las funciones SUST1 y SUST2 que reciban como argumentos una lista de asociación y una expresión simbólica L. SUST1 debe sustituir en L los primeros elementos de las parejas de la lista de asociación por sus correspondientes segundos elementos; esta sustitución la hará de forma secuencial (una sustitución puede influir en el resultado de otra sustitución anterior). SUST2 opera de forma análoga pero todas las sustituciones en la lista L deben realizarse simultáneamente (no teniendo ninguna influencia el resultado de una sustitución en las sustituciones que se realicen posteriormente).

Ejemplo:

```
> (SUST1 '( (A B) (C D) (E F) (B K) (D L) )  '(A C E B D M))
(K L F K L M)

> (SUST2 '( (A B) (C D) (E F) (B K) (D L) )  '(A C E B D M))
(B D F K L M)
```

1.31 Definir una función recursiva AGRUPAR que reciba dos argumentos, compruebe cuál de ellos es un átomo y cuál una lista, y a continuación introduzca el átomo junto a los átomos iguales que hubiera en la lista o al final de la misma, en el caso de no encontrar semejantes.

Por ejemplo:

```
> (AGRUPAR '(A A A B B B C C C) 'B)

(A A A B B B B C C C )
```

1.32 Construir de forma recursiva e iterativa la función de Fibonacci, sabiendo que:

F(0) = 1

F(1) = 1

F(n) = F(n-1) + F(n-2) si n>1

1.33 Definir una función APLANAR que reciba como argumento una expresión simbólica y elimine todos los paréntesis que aparezcan en esa expresión, devolviendo como resultado una lista con todos los átomos que aparezcan en el argumento.

Ejemplo:

```
> (APLANAR '( (1  2  3) (9  (2  3  4) ) ( ( ( ( 3  4  ( 7 ) ) ) ) ) ) )

    ( 1  2  3  9  2  3  4  3  4  7 )
```

1.34 Usando la función MAPCAR, definir PRESENTP, un procedimiento que determine si un átomo particular existe en una expresión, teniendo en cuenta los elementos de las listas anidadas.

1.35 Definir una función que reciba como argumento una lista de números y devuelva otra lista cuyos elementos sean los cuadrados de los anteriores.

1.36 Definir una función que devuelva verdadero (cualquier valor distinto de NIL) o falso (NIL) dependiendo de si el menor de sus argumentos sea par o impar. Resolverlo para dos argumentos y para un número variable de éstos.

1.37 Definir una función que compruebe si un número variable de listas tienen el mismo número de elementos.

1.38 Definir una función que al aplicarla a un número variable de números cree una lista con todos aquellos números que sean múltiplos de 3.

1.39 Definir una función que concatene un número variable de listas, comprobando que los argumentos que recibe son listas y excluyendo todos las listas que estén vacías.

1.40 Definir la función SUMA que, tomando dos argumentos numéricos devuelva su suma.

Definir la función SUMA-VARIOS con un número variable de argumentos, utilizando la función SUMA definida anteriormente. Deberá devolverse NIL si alguno de los argumentos no es numérico.

1.41 Definir una función que añada un átomo a una lista que se le pasan como argumentos a la función.

Modificar la función anterior de forma que el átomo que se le pasa como argumento sea opcional. En caso de que no se especifique en la llamada el átomo, se añadirá a la lista un átomo cualquiera.

Modificar la función anterior de forma que pueda añadirse un número variable de átomos a la lista que se le pasa como argumento.

1.42 Definir una función que tome como argumentos una lista y un número variable de átomos y devuelva una lista con los resultados de aplicar la función MEMBER con cada uno de los átomos y la lista.

Modificar la función anterior de forma que devuelva NIL si alguno de los átomos no es miembro de la lista. En caso contrario, el resultado de aplicar MEMBER al último átomo.

1.43 Definir una macro MI-IF que reciba tres argumentos, siendo el tercero opcional; si el primero es cierto devuelve el segundo, si no devuelve el tercero o NIL si éste no existiera.

1.44 Además de IF, Common LISP incorpora las primitivas WHEN y UNLESS definidas como sigue:

(WHEN <cond> <cuerpo>) = (COND (<cond> <cuerpo>)

(UNLESS <cond> <cuerpo>) =

(COND ((NOT <cond>) <cuerpo>))

Definir las macros MI-WHEN y MI-UNLESS, de forma que realicen las misma funciones que las versiones que incorpora COMMON LISP.

1.45 Definir una macro MI-DO que tenga exactamente la misma funcionalidad que la macro DO, pero que además de devolver el valor correspondiente cuando se cumpla la condición de finalización, devuelva un segundo valor que indique el número de iteraciones que se han realizado. No se deben utilizar las primitivas DO, DO*, DOLIST, DOTIMES.

1.46 Definir mediante una macro una función iterativa que realice algo (el cuerpo) un número de veces. Suponer que la llamada a la función se hace de la siguiente forma:

(DOTIMES (<var> <cont> <result>) <cuerpo>)

donde *var* es una variable que va a contar el número de iteraciones que se han realizado, su valor inicial será cero y se irá incrementando hasta que valga el valor dado en *cont*. Al final,

después de haber realizado *cont* veces el cuerpo se devolverá el valor almacenado en *result*.

1.47 Definir DOWHILE, una macro de dos argumentos que evalúa ambos hasta que el primero es NIL.

(DOWHILE <cond> <cuerpo>)

1.48 No todos los sistemas de LISP implementan el mecanismo Backquote. Definir BACKQUOTE de forma que tenga el mismo efecto que Backquote y permita manejar de forma apropiada expresiones como COMA y COMA-AT de la forma que se muestra en el siguiente ejemplo:

```
> (BACKQUOTE (A  B  (LIST ‘C  ‘D) (COMA (LIST ‘E  ‘F)
                                    (COMA-AT  (LIST  ‘G  ‘H)))
(A  B  (LIST  ‘C  ‘D)  (E  F)  G  H)
```

1.49 Supongamos que LET no existe. definir MI-LET como una macro utilizando LAMBDA.

```
(LET  ( ( <parámetro 1>  <valor inicial 1>)
        ( <parámetro 2>  <valor inicial 2>)
          … … … … …)
<cuerpo> )
```

1.50 Una pila puede ser considerada como una lista a la que puede accederse mediante las operaciones INTRODUCIR que añade un nuevo elemento al principio de la lista que representa la pila y SACAR que obtiene el primer elemento de dicha lista. Definir ambas operaciones mediante el uso de macros.

1.51 Definir una macro que comprueba si un símbolo es o no un litatom (es un átomo y no es un número).

1.52 Definir una macro SETQ-SI-NO-VALOR que reciba como argumentos una variable y un valor, de forma que si dicha variable no poseía ningún valor anterior se le asigne el nuevo valor.

1.53 Definir una macro que añada un átomo al final de una lista

1.54 Definir una macro CARATOM que devuelva el CAR del argumento de llamada siempre que dicho argumento no sea un átomo, en caso contrario devolverá dicho átomo.

1.55 Definir una macro POSICION-EN-LISTA que reciba como primer argumento un símbolo y como segundo argumento una lista, devolviendo en qué posición de la lista está el símbolo.

1.56 Definir una macro MI-FUNCALL que reciba como argumento una función y una serie de elementos y devuelva el resultado de aplicar la función a ese conjunto de argumentos.

1.57 Definir una macro MI-REMPROP que realice la misma función que REMPROP (borrar una propiedad y su correspondiente valor de una lista de propiedades)

1.58 Definir una macro denominada MI-PUTPROP que asigne un valor nuevo a una propiedad de un símbolo.

1.59 Definir una macro de nombre INTRODUCIR que permita añadir un nuevo valor a los valores que ya poseyera una determinada propiedad de un símbolo.

1.60 Definir una matriz de dimensiones 3 x 3 con los valores numéricos iniciales que se desee. Acceder a varios elementos, modificar sus valores y comprobar que la modificación se ha realizado según se deseaba. Esta matriz va a utilizarse para probar las funciones que se definirán a continuación.

Definir una función que calcule la suma de los elementos de la diagonal de la matriz que se pasa como argumento

Definir una función que suma dos matrices.

Definir una función que comprueba si algún elemento de la matriz no es numérico. Para probarlo, asignar a alguna de las componentes de la matriz un valor no numérico.

1.61 Rellenar una matriz de 20 x 20 con los valores de la distancia euclídea de cada elemento a la esquina inferior izquierda e imprimirla.

1.62 Definir una matriz de 10 filas y 20 columnas, rellenarla, asignando a cada elemento el valor de la suma de su fila más su columna y obtener una lista de salida con los elementos de la diagonal.

1.63 Un conjunto puede representarse como una lista, siendo cada elemento del conjunto un átomo. Cada elemento debe aparecer una sola vez en la lista, y el orden en que aparezcan no debe tener ningún significado especial. Utilizando esta representación de los conjuntos, definir las siguientes funciones (algunas de ellas ya están implementadas en COMMON LISP, definir una versión propia).

a) Definir MI-UNION. La unión de dos conjuntos es un conjunto que contiene todos los elementos que aparecen en alguno de los dos conjuntos.

b) Definir MI-INTERSECCION. La intersección de dos conjuntos contendrá aquellos elementos que aparecen en los dos conjuntos.

c) Definir MI-DIFERENCIA-CONJUNTOS. La diferencia de dos conjuntos A y B estará constituida por todos aquellos elementos de A que no aparezcan en B.

1.64 Definir IGUALESP, un predicado que comprueba si dos conjuntos tienen los mismos elementos. Observe que las dos listas que representan los conjuntos pueden tener los elementos en orden diferente, por lo tanto no es válido utilizar EQUAL.

1.65 Escribir un procedimiento TENDENCIA que reciba como argumentos una clave y dos listas de asociación en las que las claves son nombres de empresas que cotizan en bolsa y los valores de las claves son las correspondientes cotizaciones para un determinado día. La función debe devolver ALTA, BAJA, ESTABLE dependiendo de la relación entre los valores asociados a esas claves en las listas de asociación pasadas como argumentos. Considérese también el caso de que la clave no aparezca en una o en ninguna de las listas de asociación, devolviendo en estos casos los mensajes que considere apropiados.

1.66 Definir una función LETRAS que tienen un número variable de argumentos; todos los argumentos serán listas o átomos. Si aparece un único argumento debe ser un átomo, y la función debe devolver una lista con las letras de ese átomo. si aparecen más argumentos serán o bien listas de caracteres o bien átomos. En esos casos, la función debe devolver una lista que contenga todos los caracteres.

1.67 Definir una función LETRAS2 que acepte dos argumentos siendo uno de ellos opcional. Si la función es llamada con un único argumento, el resultado será el mismo que el descrito en el ejercicio anterior. Si aparecen los dos argumentos, uno de ellos será una lista y el otro un átomo, que podrán aparecer un cualquier orden. En este último caso la función debe devolver una lista donde se habrán incluido los caracteres que forman el átomo y los que aparecían en la lista. No debe aparecer ningún carácter repetido en la lista resultado.

1.68 Definir RQ, un procedimiento que imprime su argumento como una lista no anidada, lee una forma dada por el usuario e

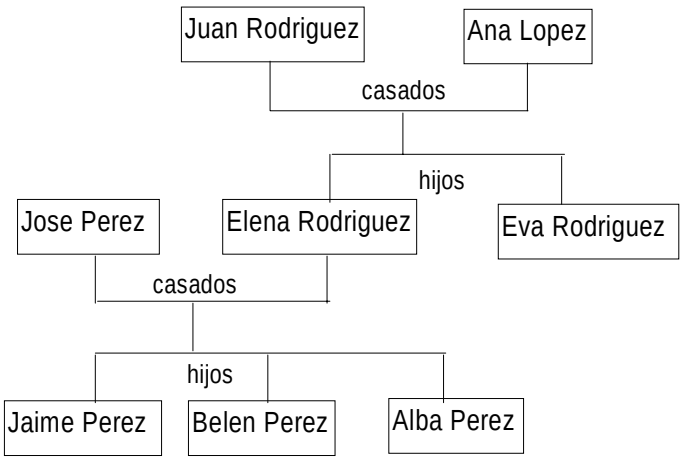
imprime su valor. El siguiente ejemplo muestra su funcionamiento:

```
> (RQ  '(POR FAVOR SUMINISTRE UN VALOR PARA E))

(POR FAVOR SUMINISTRE UN VALOR PARA E) (+  3  4)

7
```

1.69 Definir una función que reciba como argumento una lista de tabuladores (enteros que indican las columnas en que debería comenzarse la escritura). El procedimiento debe leer un átomo y escribir cada una de sus letras en la columna especificada en la lista de tabuladores. Considérese que en la lista aparece un número suficiente de tabuladores como para asignar uno a cada letra, y que los enteros que aparecen en la lista de tabuladores aparecen ordenados en orden creciente.

1.70 Definir una estructura PERSONA en la que se incluyan los campos NOMBRE, APELLIDO1, APELLIDO2 y DNI con valores por defecto sus datos personales. Definir un procedimiento que lea los datos correspondientes a varias personas y los introduzca en las sucesivas posiciones de un vector EMPLEADOS. Cada posición del vector contendrá una estructura de las definidas anteriormente. Una vez leídos los datos correspondientes a una persona la función debe verificar estos datos, pidiendo conformidad y modificarlos si se ha producido un error. El procedimiento finalizará la lectura de datos cuando el nombre introducido por el usuario sea N. Defínase el vector EMPLEADOS con un tamaño suficiente para incluir los datos de 10 empleados.

1.71 Definir una estructura LISP PERSONA que contenga la siguiente información: nombre, apellido, cónyuge, padres, hijos y si está viva o no. Definir instancias de la estructura para representar la siguiente familia:



a) Definir funciones que obtengan: padres, abuelos y tíos de una determinada persona.

b) Definir una función que determine cuáles son los herederos de una persona. Para ello habrá que tener en cuenta las siguientes reglas:

- Si una persona tiene un cónyuge que está vivo, entre sus herederos estará el cónyuge.

- Si una persona tiene hijos que están vivos, estos estarán entre sus herederos.

- Si una persona tiene un descendiente (hijo) que tiene hijos, los hijos del descendiente serán sus herederos siempre que el cónyuge y sus hijos hayan fallecido.

- Si una persona no tiene hijos ni cónyuge ni padres vivos, sus herederos serán los herederos de sus padres.

2. Ejercicios avanzados de Lisp

2.1 Las expresiones aritméticas pueden representarse utilizando un árbol binario, como por ejemplo:

```
(* (+ A B) (- C (/ D E)))
```

Parte del trabajo de un compilador es traducir estas expresiones aritméticas del lenguaje máquina de algún ordenador. Supóngase, por ejemplo, que la máquina tiene un conjunto de registros numerados secuencialmente que pueden manejar resultados intermedios. Supóngase también que la máquina tiene una instrucción MOVE para introducir valores en estos registros, y las instrucciones ADD, SUB, MUL y DIV para realizar operaciones aritméticas entre dos registros. El ejemplo anterior podría traducirse a la siguiente secuencia de instrucciones:

```
( (MOVE 1 A)
  (MOVE 2 B)
  (ADD 1 2)
  (MOVE 2 C)
  (MOVE 3 D)
  (MOVE 4 E)
  (DIV 3 4)
  (SUB 2 3)
  (MUL 1 2) )
```

El resultado debe dejarse en el registro 1. Definir COMPILADOR-ARITMETICO, un procedimiento que ejecuta esta traducción.

2.2 Definir un procedimiento CIFRADO que debe leer un mensaje escrito por el usuario y de acuerdo a un método previamente fijado debe devolver el texto cifrado. Puede utilizarse como ejemplo de función de cifrado la sustitución de cada carácter del texto por el carácter que se obtiene al sumar a su número ASCII una cantidad fija. Considérese la posibilidad de modificar el procedimiento para que puede leer también el método de cifrado.

Definir un procedimiento que realice la operación inversa a la descrita en el ejercicio anterior, leyendo un mensaje cifrado y devolviendo el mensaje original

2.3 Definir un procedimiento TORRES-HANOI que resuelva el problema de las torres de Hanoi para un número arbitrario de discos. La resolución de este problema implica detallar los pasos a realizar para trasladar un número determinado de discos de una aguja A hasta otra aguja B utilizando una aguja auxiliar C. el movimiento de los discos debe seguir las siguientes reglas.

- Sólo se puede mover un disco cada vez

- Los discos tienen diferentes diámetros, ningún disco puede situarse sobre otro de menor diámetro.

Representar las agujas mediante listas que incluyan en orden los discos que contiene la aguja correspondiente. Utilice las variables A, B, C como nombre de las agujas. la situación para el caso, por ejemplo, de 3 discos se especificaría de la forma:

```
(SETQ   A '(1  2  3)     B  NIL     C  NIL)
```

Para la situación inicial especificado el resultado obtenido debería ser:

```
>  (TORRES-HANOI)

(MUEVE  1  de  A  a  B)
(MUEVE  2  de  A  a  C)
(MUEVE  1  de  B  a  C)
(MUEVE  3  de  A  a  B)
(MUEVE  1  de  C  a  A)
(MUEVE  2  de  C  a  B)
(MUEVE  1  de  A  a  B)
```

2.4 Definir un procedimiento SALTO-CABALLO que acepte como único argumento el tamaño del tablero de ajedrez para el que se desea resolver el problema. La solución del problema debe especificar los sucesivos movimientos de un caballo en el juego del ajedrez, de forma que comenzando en la casilla superior izquierda recorra todas las posiciones del tablero sin pasar dos veces por ninguna. El procedimiento debe devolver un tablero del tamaño especificado en el que en cada casilla aparezca el número

correspondiente a la secuencia de movimientos del caballo en el tablero. El movimiento del caballo debe seguir las reglas del juego del ajedrez. Si no es posible encontrar una solución para el tamaño del tablero especificado debe devolver el correspondiente mensaje.

2.5 Escribir REINAS, una función que devuelva la solución al problema de las ocho reinas. Se trata de situar a ocho reinas en un tablero de ajedrez, de manera que ninguna de ellas sea atacada por otra, es decir, no puede haber dos reinas en la misma fila, columna o diagonal.

2.6 Un móvil es un tipo de escultura abstracta construida por elementos que pueden tener un movimiento relativo unos respecto a otros. Puede definirse un tipo particularmente simple de móvil de forma recursiva como, o bien un objeto suspendido en el aire, o bien una barra con un submóvil colgando de cada extremo. Si se asume que cada barra está suspendida de su punto medio, puede representarse un móvil como un árbol binario. Los objetos suspendidos ser representarán como números que corresponderán al peso de cada objeto. Los móviles más complejos se representarán como listas de tres elementos; el primero de ellos será un número igual al peso de la barra, y los otros dos representan submóviles unidos a los extremos de la barra.

Un móvil debe ser balanceado, es decir, los dos submóviles de los extremos de la barra deben tener el mismo peso. definir MOVILP, una función que determina si un móvil está o no balanceado. Devolverá NIL si no lo está y el peso total del móvil en caso contrario.

```
> (MOVILP  '(6  (4  (2  1  1)  4)  (2  5  (1  2  2))))

30
```

2.7 El juego de la vida es un juego de simulación que se desarrolla en una cuadrícula, de modo que en cada casilla pueda

haber un organismo. Cada casilla se puede encontrar ocupada o vacía.

Dos organismos se consideran vecinos si sus casillas son contiguas en sentido horizontal, vertical o diagonal. Por tanto, cada casilla tiene ocho casillas vecinas.

Las reglas del juego son:

- En cada casilla vacía nace un nuevo organismo si dicha casilla tiene exactamente tres vecinos.

- Una casilla ocupada que tenga cero o un vecino muere por aislamiento.

- Una casilla ocupada con cuatro o más vecinos muere por superpoblación.

- Una casilla ocupada con dos o tres vecinos sobrevive.

Todos los nacimientos y muertes ocurren simultáneamente y la aplicación de las leyes anteriores produce una nueva generación.

El juego continúa hasta que suceda uno de los siguientes hechos:

- La generación actual es igual que la generación inicial.

- La generación actual es igual que dos generaciones anteriores.

- Se ha alcanzado el número máximo de generaciones prefijado.

2.8 Una empresa desea organizar todas las facturas que posee de cada uno de sus clientes:

a) Crear una estructura CLIENTE que contenga los campos NOMBRE, DIRECCION, RAZON-SOCIAL, NUMERO-DE-

FACTURAS y FACTURAS (lista en la que se encuentran todas las facturas que posea un cliente).

b) Asignar a la variable CLIENTE1 una estructura en la que los campos tengan inicialmente los siguientes valores:

DIRECCION: “Paseo de la Castellana, 125”

RAZON-SOCIAL: “Desarrollo de programas inteligentes”

NUMERO-DE-FACTURAS: 10

Comprobar que cada uno de los campos ha sido adecuadamente inicializado, utilizando las diferentes funciones de acceso creadas para cada uno de los mismos por el COMMON LISP.

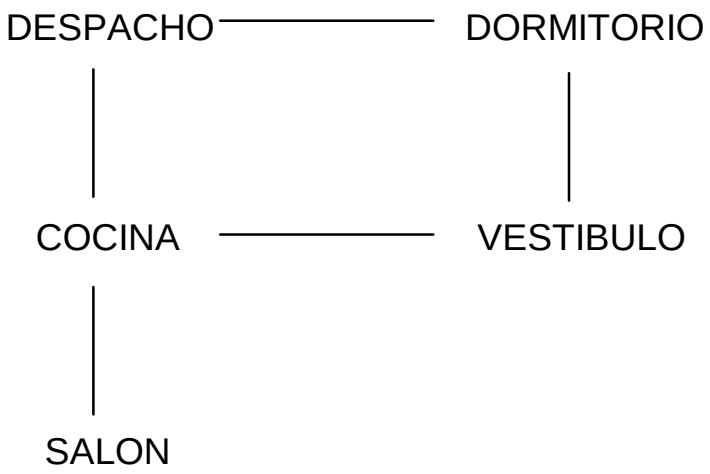
c) Definir otra estructura denominada FACTURA cuyos componentes sean: NUMERO-FACT, NIF (número de identificación fiscal), FECHA e IMPORTE.

d) Crear dos facturas FACTURA1 y FACTURA2 inicializando el valor de sus campos con valores arbitrarios. Introducir ambas facturas en la lista de facturas del CLIENTE1.

e) Acceder al campo FECHA de la FACTURA2 del cliente CLIENTE1 (suponiendo que éste ocupa la segunda posición en la lista de facturas asignada al campo FACTURAS de dicho cliente).

f) Como ha podido apreciarse en el apartado anterior el acceso a un determinado campo de una determinada factura es bastante engorroso. Definir para tal efecto una macro que reciba como argumentos el nombre del cliente, el nombre de la factura y el campo del que se desee obtener el valor de dicha factura. Utilizar dicha función para modificar alguno de dichos valores.

2.9 Supongamos que tenemos un robot que puede moverse en una de las cuatro direcciones en el interior de una casa. El plano de la casa es el siguiente:



El valor de la propiedad HABITACIONES del átomo CASA será una lista con el nombre de todas las habitaciones que aparecen en el plano. Para cada una de las habitaciones, sobre el nombre de las propiedades NORTE, SUR, ESTE y OESTE incluir el nombre de la habitación que se encuentra en la dirección correspondiente.

Definir una función POSIBILIDADES que reciba como argumento el nombre de una habitación y devuelva una lista en la que cada elemento es una de las posibles direcciones en el movimiento del robot junto a la habitación que se encuentra de esa dirección.

Por ejemplo:

```
> (POSIBILIDADES ‘VESTIBULO)

( (NORTE DORMITORIO) (OESTE COCINA) )
```

Definir una función TERMINO que dados dos argumentos, una dirección y una habitación, devuelva la habitación donde se encontraría el robot si avanzara en esa dirección.

Por ejemplo:

```
> (TERMINO 'NORTE 'COCINA)

DESPACHO
```

Utilizar la propiedad LUGAR del símbolo ROBOT para almacenar la posición del robot. Situarlo inicialmente en el vestíbulo.

Definir una función NUMERO-POSIBILIDADES que devuelva el número de posibles direcciones de movimiento para el robot.

Definir una función MUEVETE que recibe como argumento una dirección y mueve el robot en esa dirección. Si el movimiento en esa dirección no está permitido debe devolver un mensaje de aviso.

Definir una función que reciba como argumentos los nombres de dos habitaciones y devuelva los nombres de aquellas habitaciones por las que debería pasar el robot en su movimiento de una a otra. Utilizar para ello las funciones definidas anteriormente.

2.10 Definir una función LISP que carge un fichero que contiene una base de reglas (las reglas de los llamados sistemas basados en el conocimiento tienen el formato: "Condición $\rightarrow$ Acción". Suponer que la *condición* de las reglas está formada por un conjunto de cláusulas unidas por las conectivas AND y OR.

Condición $\equiv$ C_1 and C_2 and C_3

C_1 or (C_2 and C_3)

(C_1 or C_2) and C_3

Por otro lado, la acción puede estar formada por más de una cláusula unidas mediante la conectiva AND.

Acción ≡ A_1 and A_2 and A_3

A_1 and A_3

A_1

En el fichero las reglas aparecen una tras otra en forma de listas, siendo el primer elemento de la lista su nombre y el segundo una lista compuesta de dos listas: su parte de *acción* seguida de la *condición*. Por ejemplo:

(R1 ((and (equal ?x persona) (equal ?x estudiante)) (presentar-nota ?x)))

La función definida debe leer cada una de las reglas y transformarla en una notación interna que sea una estructura con dos campos: uno para la parte condición y otro para la de acción.

2.11 Definir una estructura denominada REGLA que contenga dos campos; PC y PA, siendo dichos campos los que identifican las partes de condición y de acción de cada una de las reglas que se vayan a crear. PC y PA serán dos listas de la forma:

PC: ((A < 5) (B > 7) (CONSP C))

PA: AÑADIR-SUMA

Siendo AÑADIR-SUMA una función que sume A y B añadiendo el resultado de la suma como nuevo último elemento de la lista C.

a) Definir la función AÑADIR-SUMA.

b) Construir un procedimiento que compruebe si se cumple la parte de condición de la regla. Es decir, comprueba que cada uno de los predicados incluidos en PC devuelve un valor distinto de NIL

c) Definir una función MOTOR que utilizando las funciones anteriores compruebe si una regla es aplicable (se cumplan las condiciones de PC), en cuyo caso realizaría lo indicado en su parte de acción PA.

2.12 Escribir un programa que aprenda a jugar a "¿Sabes de qué animal se trata?. Este juego consiste en que una persona piensa en un animal y el programa ha de descubrir cual es el animal en que se piensa mediante preguntas a las que se ha de contestar SI o NO.

Algunas veces el programa será capaz de determinar de que animal se trata. En otras ocasiones dirá otro animal que no es el deseado, en cuyo caso el programa preguntará de qué animal se trata y qué pregunta habrá de hacer para diferenciarlo del que él ha sugerido.

Los animales pueden clasificarse atendiendo a los valores de una serie de propiedades que los identifican (pares de la forma: atributo<->valor)

Por ejemplo, un perro puede tener la propiedad:

ATRIBUTO	VALOR
especie	mamífero

Esta propiedad, por ejemplo, permitiría diferenciarlo de una tortuga.

Igualmente un gato podría tener la propiedad:

ATRIBUTO	VALOR
sonido	maullido

Mediante la cual puede distinguirse a un perro de un gato.

El programa LISP que se propone desarrollar debe aprender a identificar a los animales atendiendo al valor de las propiedades que estos posean. Para ello se pretende construir un árbol binario

de discriminación de especies de animales, definido en base a las contestaciones que un usuario del juego haga a las preguntas que el propio juego debe plantear.

Para poder realizar este ejercicio es necesario saber lo que es un árbol binario. Un árbol binario puede ser definido recursivamente como, o bien una hoja, o un nodo con dos árboles binarios unidos a él. Esta estructura puede representarse utilizando átomos para las hojas y listas de tres elementos para los nodos. El primer elemento de cada lista será un átomo que representa un nodo, y los otros dos elementos serán los subárboles unidos a él.

El siguiente ejemplo muestra la representación de un árbol con seis nodos N1 a N6 y siete hojas H-A a H-G:

```
(N1 (N2 H-A H-B) (N3 (N4 H-C H-D) (N5 H-E (N6 H-F H-G))))
```

Volviendo al ejercicio en si. Una de las preguntas que el programa podría realizar sería:

"¿ Es maullido el valor del atributo sonido de tu animal?"

El desarrollo completo del juego podría corresponderse con el siguiente:

```
> (comenzar)

¿ Quieres que averigüe en qué animal estás pensando ? s

¿ Es MAMIFERO el valor del atributo ESPECIE de tu animal ? s

¿ Es tu animal un(a) PERRO ? n

¿ Qué nuevo atributo y valor tiene tu animal que PERRO no posee?

ATRIBUTO: sonido

VALOR: maullido
```

¿ De qué animal se trata? gato

¿ Quieres que averigüe en qué animal estás pensando ? s

¿ Es MAMIFERO el valor del atributo ESPECIE de tu animal ? s

¿ Es MAULLIDO el valor del atributo SONIDO de tu animal? n

¿ Es tu animal un(a) PERRO ? n

¿ Qué nuevo atributo y valor tiene tu animal que PERRO no posee?

ATRIBUTO: movimiento

VALOR: galopar

¿ De qué animal se trata? caballo

¿ Quieres que averigüe en qué animal estás pensando ? s

¿ Es MAMIFERO el valor del atributo ESPECIE de tu animal ? s

¿ Es MAULLIDO el valor del atributo SONIDO de tu animal? s

¿ Es tu animal un(a) GATO ? s

¿ Quieres que averigüe en qué animal estás pensando ? n

La lista que representa el árbol binario de discriminación creado es:

((GATO (SONIDO MAULLIDO) (CABALLO (MOVIMIENTO GALOPAR) PERRO)) (ESPECIE MAMIFERO) TORTUGA)

Para crear este juego pueden utilizarse las siguientes funciones:

- Una función RECORRER que recorra el árbol binario haciendo preguntas según los pares atributo<->valor de cada nodo, Cuando el programa realiza una suposición errónea, RECORRER se preocupa de que un nuevo nodo de discriminación sea introducido convenientemente en el árbol.

- Funciones que realicen las preguntas necesarias al usuario (guarda todas estas funciones en un apartado distinto del fichero que contenga el programa).

- Las funciones básicas que permitan que el programa funciones correctamente.

Utilice un estilo de programación modular, de forma que haya muchas funciones pequeñas particionadas a su vez en otras según la misión que éstas deban realizar, documentando con la mayor claridad posible todas y cada una de ellas. Su principal objetivo no debe ser la eficiencia sino la abstracción.