Manejador de Listas

PRÁCTICA DE FUNDAMENTOS DE PROGRAMACIÓN

Se pide realizar un programa en C que, mediante una interfaz de línea de comandos, también llamada **CLI** (*Command Line Interface*), permita realizar operaciones con listas de valores numéricos (enteros y reales). Durante la ejecución del programa se podrán ir creando variables y asignarles un valor, dicho valor será siempre una lista de valores numéricos (la lista podría estar vacía). Adicionalmente, se podrán hacer operaciones con estás variables para calcular nuevos valores (listas) y/o crear nuevas variables o modificar el valor de alguna de las variables ya creadas. En cada momento, el conjunto de variables y valores que se hayan creado constituirán el espacio de trabajo (o workspace) que se deberá poder guardar en disco para poder recuperarlo más adelante.

Sobre el funcionamiento general del programa, al arrancar imprimirá en pantalla el nombre y apellidos del alumno y su email, y a continuación, para que el usuario pueda introducir comandos desde teclado, mostrará el siguiente prompt:

<*>:

La ejecución general del programa deberá ser como se indica en el siguiente pseudocódigo:

1. Imprimir en pantalla datos del alumno:
   nombre, apellidos y correo electrónico
2. Mostrar el **prompt en color blanco**
3. Leer comando (o expresión) introducido por teclado
4. Analizar el comando
   ⇒ Si el comando es incorrecto se indica con un **mensaje de error en rojo** y se vuelve al paso 2
5. Determinar si es posible ejecutar el comando
   ⇒ Si no es posible ejecutar el comando se indica con **un mensaje de error en rojo** y se vuelve al paso 2
6. Si el comando es '**exit**' (sin parámetros) se termina la ejecución del programa, liberando previamente toda la memoria y mostrando por pantalla el **mensaje "*Bye, bye*" en verde** y volver al **color por defecto blanco**
7. Si el comando no es '**exit**' se ejecuta la acción correspondiente y se vuelve al paso 2. Cuando la acción implique mostrar alguna salida por pantalla, se mostrará dicha **información en color verde**

En el paso 7, donde dice “***se ejecuta la acción correspondiente***” habrá que realizar alguna de las acciones que se describirán a lo largo de este documento. Nótese que a veces esas acciones deberán mostrar un resultado por pantalla, en cuyo caso, dicho resultado se mostrará en color verde, seguido inmediatamente del prompt en color blanco. Otras veces, no será necesario proporcionar ninguna salida, por lo que inmediatamente se volverá a mostrar el prompt (siempre en color blanco).

Como ya se ha introducido, el programa a realizar deberá hacer operaciones con listas. Una lista de valores será una serie de números, enteros o reales, colocados entre corchetes y separados por espacios o tabuladores, por ejemplo, la siguiente descripción sería una lista válida:

```
[12 +33   3.141592      25 18.33 -77.223344  0]
```

Una lista vacía se representa con una pareja de corchetes sin ningún contenido en su interior, como se muestra a continuación:

```
[]
```

A la hora de declarar una lista se puede hacer de dos formas, de forma explícita o implícita. La forma explícita es indicando uno tras otro, todos los números que debe contener la lista, ejemplo de ello son las dos listas que se han puesto más arriba (una con 7 elementos y otra, vacía, con 0 elementos). La forma implícita consiste en indicar un valor inicial, uno final y, opcionalmente, un incremento, según el siguiente patrón:

```
[valor_inicial : valor_final]
[valor_inicial $ incremento : valor_final]
```

véanse los siguientes ejemplos (lista implícita ⇒ lista explícita equivalente):

```
[1 : 5]        ⇒ equivale a: [1 2 3 4 5]
[1.5 : 5]      ⇒ equivale a: [1.5  2.5  3.5  4.5]
[0 $ 3 : 10]   ⇒ equivale a: [0  3  6  9]
[0 $ 0.1 : 1]  ⇒ equivale a: [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1]
[1 : 1]        ⇒ equivale a: [1]
[1 $ 2 : 1]    ⇒ equivale a: [1]
```

Nótese que, el valor inicial siempre aparece al principio de la lista, el valor final puede aparecer o no, pero no se puede sobrepasar. El incremento por defecto es +1, siempre que el valor inicial sea menor que el valor final. Si fuera al revés (valor inicial mayor que final) el incremento por defecto será -1. Opcionalmente se puede indicar un incremento diferente tras el carácter ‘$’. En los ejemplos anteriores, los incrementos han sido positivos, con valor inicial menor que el valor final, pero también deben poder indicarse incrementos negativos, en esos casos el valor inicial tiene que ser mayor que el valor final, si no se cumplen estas condiciones, la lista es incorrecta. Tampoco tiene sentido un

incremento igual a cero, eso también se considera un caso de error. A la hora de guardar una lista en memoria (ver estructuras de datos más adelante) o de mostrarla por pantalla siempre se hará en su forma explícita, mostrando uno tras otro todos los elementos de la lista. En lo sucesivo, se va a denotar con los identificadores **L1**, **L2**, …, **Lx** cualquier valor de lista válida, incluida la lista vacía.

Las operaciones a realizar se pueden dividir en dos grupos, las de asignación y las de muestra de resultados por pantalla. Adicionalmente habrá algunas operaciones más para guardar o leer datos de ficheros o para operaciones booleanas. Las operaciones de asignación tendrán la siguiente forma:

variable = <expresión>

donde:

- **variable**: es cualquier identificador válido para una variable, es decir, cualquier combinación no vacía de caracteres alfanuméricos (excepto eñes, o vocales con tilde o diéresis). La única regla es que el primer carácter debe ser una letra de la 'a' a la 'z' o de la 'A' a la 'Z', el segundo carácter y siguientes (si los hubiera) podrán ser letras o dígitos numéricos. No serán nombres de variables válidos todos aquello que coincidan con los comandos que se van a describir más adelante.

- **<expresión>**: una expresión es cualquier operación de genera o devuelve un valor, en este programa, los valores que puede tomar una variable solamente pueden ser listas como las descritas anteriormente, por tanto, en una operación de asignación las expresiones válidas únicamente serán aquellas cuyo resultado sea una lista. Ejemplos de expresiones válidas en una asignación son:

  - ➢ L1 .- Expresión que contiene solo una lista y devuelve la propia lista.

  - ➢ L1 + L2 .- Concatenación, devuelve una lista que contiene los elementos de ambas listas respetando el orden, primero todos los de L1, seguidos de todos los de L2.

  - ➢ L1 - L2 .- Diferencia, devuelve una lista con los elementos de L1 que no se encuentran el L2 sin eliminar valores repetidos, es decir, si el elemento 'x' está dos veces en L1 y una vez en L2 solamente se elimina una vez de L1.

  Las expresiones pueden contener también variables que hayan sido creadas previamente y que contengan un valor, por ejemplo, la siguiente secuencia de instrucciones sería correcta:

```
a = [1 : 5] // crea la variable 'a' con el valor [1 2 3 4 5]
b = a + [20 4 8 16] // crea b ⇐ [1 2 3 4 5 20 4 8 16]
c = b - [4 1 8] // crea c ⇐ [2 3 5 20 4 16]
```

```
x = c - a // crea x ⇐ [20 16]
b = [20 $ -4 : 10] + x // modifica b ⇐ [20 16 12 20 16]
```

En cambio, la siguiente instrucción sería incorrecta:

```
c = [1 : 5] + z // No existe la variable ‘z’
```

Nótese que las variables puestas a la izquierda del signo ‘=’, si no existen se crean nuevas; si ya existían con anterioridad se modifican. En cambio, a la derecha del signo igual solo pueden ponerse listas (explícitas o implícitas) y variables que ya se hayan inicializado anteriormente.

Las operaciones de muestra de resultados son expresiones que calculan un valor y lo muestran por pantalla, pero no lo asignan a ninguna variable. En todos los ejemplos anteriores, si se eliminan la variable de la izquierda y el signo igual, la expresión que queda, ella sola, sería una operación de muestra de resultados válida, es decir, se calcularía el resultado de la expresión en una variable temporal, para luego mostrar por pantalla en resultado. Una variable, ella sola, o una lista, también sola, serían igualmente operaciones de muestra de resultados. Véanse los siguientes ejemplos:

```
<*>: a = [1 : 4] // asignación (no muestra resultados)
<*>: a
[1 2 3 4] // resultado mostrado por pantalla
<*>: a + [-7 +7 -7.5]
[1 2 3 4 -7 7 -7.5] // resultado mostrado por pantalla
```

Adicionalmente, se tiene que implementar una expresión que consistirá en una operación booleana, que devolverá TRUE o FALSE (verdadero o falso). Esta expresión, al no devolver una lista no se podrá usar como parte derecha de una asignación, ***solo se podrá usar como expresión de muestra de resultados***, dicho resultado, como se ha indicado, podrá ser TRUE o FALSE (que se mostrarán en verde). Dicha expresión utilizará el carácter ‘#’ (almohadilla) como operador y su significado será averiguar si la lista de la izquierda esta contenida dentro de la lista de la derecha:

```
<*>: L1 # L2 // ¿está L1 contenido en L2?
TRUE // si L1 esta contenido dentro de L2, si no ⇒ FALSE
```

En esta expresión hay que considerar los valores repetidos, por ejemplo, si L1 tiene el elemento ‘x’ dos veces, y L2 tiene ese mismo elemento una sola vez, entonces L1 no estará contenida en L2. Una lista vacía SIEMPRE está contenida en cualquier otra.

Además de los operadores ‘+’ (más), ‘-’ (menos), ‘=’ (igual) y ‘#’ (almohadilla) hay que implementar otros comandos o expresiones adicionales, algunas de las cuales devolverán una lista y otras no. Las que devuelven una lista se podrán usar como parte derecha en una asignación o simplemente como operaciones de muestra de resultados. Las

operaciones que no generan una lista como resultado, solo se podrán usar como operaciones de muestra de resultados, es decir, será un error usarlas como parte derecha de una asignación. Nótese que cada operación tiene un número concreto de parámetros, si al escribir una instrucción se indicaran más o menos elementos (parámetros) de los que corresponda en cada caso, el comando se considerará incorrecto y se indicará con el correspondiente mensaje de error (en rojo). Las operaciones adicionales que hay que implementar son:

`head n L1` ***(devuelve una lista)***

‘n’ representa un número entero, la operación devuelve una lista con los ‘n’ primeros elementos de la lista ‘L1’. Si ‘n’ es menor que 1, se devuelve una lista vacía. Si ‘n’ es mayor que la longitud de la lista se devuelve la lista completa. Si ‘n’ no es un entero o L1 no es una lista válida se indicará con un mensaje de error.

`tail n L1` ***(devuelve una lista)***

‘n’ representa un número entero, la operación devuelve una lista con los ‘n’ últimos elementos de la lista ‘L1’. Si ‘n’ es menor que 1, se devuelve una lista vacía. Si ‘n’ es mayor que la longitud de la lista se devuelve la lista completa. Si ‘n’ no es un entero o L1 no es una lista válida se indicará con un mensaje de error.

`isIn n L1` ***(devuelve un booleano)***

‘n’ representa un número, entero o real, la operación devuelve TRUE si el valor ‘n’ está dentro de la lista ‘L1’, devuelve FALSE en caso contrario. Si ‘n’ no es un número correcto o L1 no es una lista válida se indicará con un mensaje de error.

`vars` ***(no genera ningún valor)***

Muestra por pantalla un listado de las variables que se han generado en el programa y el número de elementos de cada una (longitud de la lista).

`save nombre_fichero` ***(no devuelve ni genera ningún valor; ver página siguiente)***

Guarda en un fichero de texto el workspace (espacio de trabajo) actual, es decir, todas las variables que se hayan creado junto con sus valores.

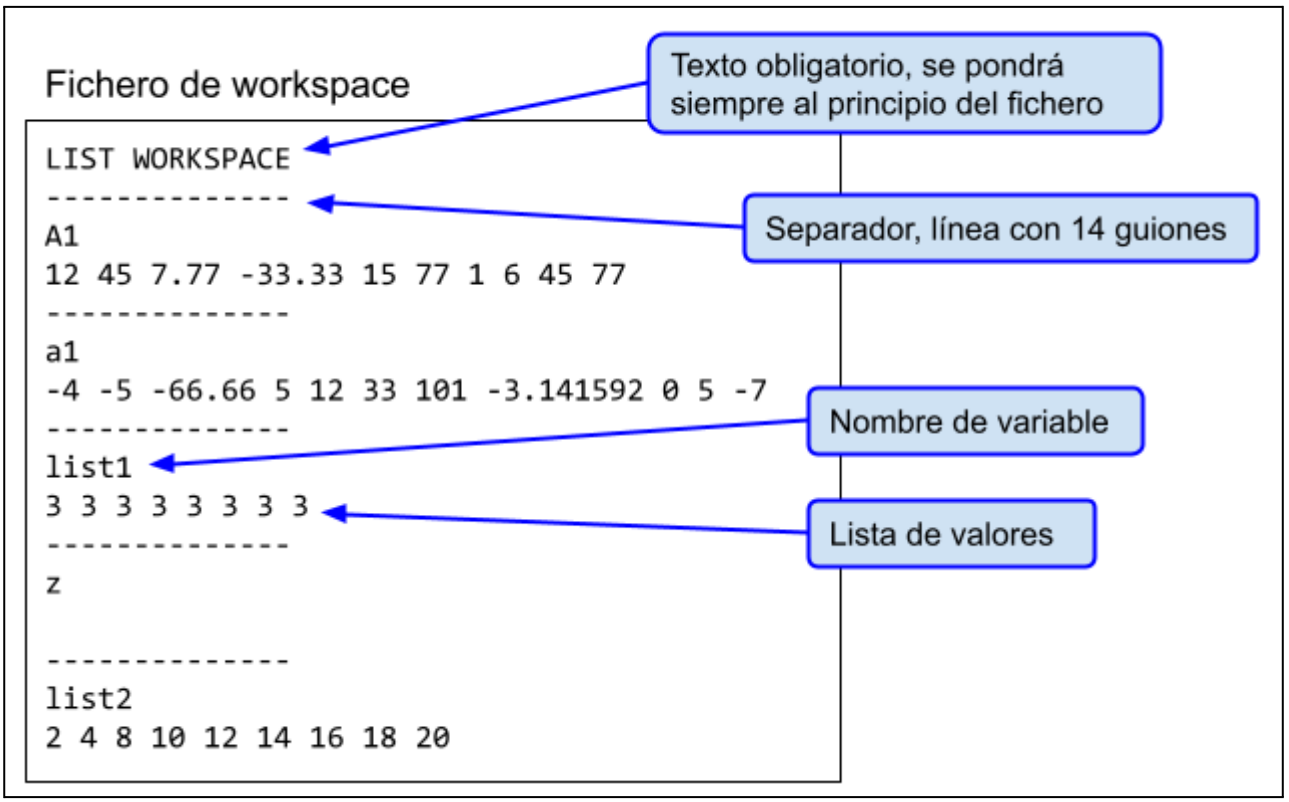
`load nombre_fichero` ***(no devuelve ni genera ningún valor; ver página siguiente)***

Lee un espacio de trabajo o workspace del fichero indicado reemplazando por completo el workspace actual, es decir, se eliminan todas las variables que hubiera en memoria y se sustituyen por las que haya en el fichero leído. Si el fichero no existe o no es un workspace correcto, no se cambia nada y se indica con un mensaje de error.

`exit` ***(no devuelve ni genera ningún valor)***

Finaliza la ejecución del programa, libera toda la memoria que hubiera reservada, muestra el mensaje **`"Bye, bye"`** (en verde) y finaliza el programa (se vuelve al prompt de la consola normal del sistema en color blanco).

Un workspace o espacio de trabajo representa el conjunto de variables y valores generados durante la ejecución de un programa. Almacenado en un fichero de texto deberá tener la siguiente forma:



El fichero contendrá, por cada variable, un separador formado por una línea con 14 guiones (“----------”), en la línea siguiente el nombre de dicha variable, y el listado de elementos de dicha variable separados por espacios justo en la línea siguiente.

El comando “save” deberá generar un fichero con el formato que se acaba de indicar.

El comando “load” deberá ser capaz de leer y capturar los datos contenidos en un fichero como ese mismo formato.

El ejemplo de la figura anterior se corresponde con un espacio de trabajo o workspace que contiene cinco variables: ‘**A1**’, ‘**a1**’, ‘**list1**’, ‘**z**’ y ‘**list**’. Concretamente la variable ‘**z**’ tiene debajo una línea en blanco, eso quiere decir que se trata de una lista vacía. Además, se puede ver como hay dos variables, ‘**A1**’ y ‘**a1**’, que solo se diferencian en que una está en mayúsculas y la otra en minúsculas. Eso quiere decir que el programa debe distinguir entre mayúsculas y minúsculas y, por tanto, identificadores como ‘**var**’, ‘**Var**’ o ‘**VAR**’ serán considerados variables distintas (ojo, ‘**vars**’ no sería una variable válida ya que hay un comando con ese mismo nombre, en cambio, ‘**VARS**’ sí sería una variable válida ya que no coincide con el nombre de ningún comando al estar escrito con letras mayúsculas).

Para almacenar en memoria tanto las listas de valores numéricos como un workspace con todas sus variables y sus valores, OBLIGATORIAMENTE hay que utilizar las siguientes estructuras de datos en C para construir listas doblemente enlazadas:

1) Para implementar listas de números reales

```
typedef struct Numero
{
      double valor;
      int decimales;
      struct Numero *siguiente, *anterior
} TNUM;

typedef struct
{
      int n; // nº de elementos de la lista
      TNUM *primero, *ultimo;
} TLISTA;
```

2) Para implementar la lista de variables que componen el espacio de trabajo

```
typedef struct Var
{
      char nombre[20];
      TLISTA *valor;
      struct Var *siguiente, *anterior
} TVAR;

typedef struct
{
      int n; // nº de elementos de la lista
      TVAR *var1;
} TVARS;
```

Según estas definiciones, un workspace será un puntero a **TVARS**, en plural, o una estructura de tipo **TVARS**. Esta estructura, a su vez contiene internamente una lista '**var1**' de estructuras (nodos) '**TVAR**', en singular. Cada elemento '**TVAR**' representa una variable del workspace, por tanto, será una lista de números, representada por la estructura '**TLISTA**', que a su vez contendrá nodos de tipo '**TNUM**' para guardar cada número de la lista.

Cuando se almacena un número en un nodo '**TNUM**' se almacena su '**valor**' y también el número de '**decimales**' con los que se ha escrito el número, de esa forma, a la hora de imprimirlos en pantalla o en un fichero, se usará esa información para determinar el formato de impresión (la cantidad de decimales).

Tanto la estructura '**TVARS**' (lista de variables) como la estructura '**TLISTA**' (lista de números) tienen un componente '**n**' que debe utilizarse para llevar la cuenta de la cantidad de nodos que hay en cada lista. Cuando una lista esté vacía el valor de '**n**' debe ser '0' (cero) y los punteros de dicha lista, '**var1**' en el caso de '**TVARS**' y, '**primero**' y '**ultimo**' en el caso de '**TLISTA**' deberán tener el valor **NULL**.

La implementación final de la práctica deberá estar formada por tres ficheros:

- **main.c**: con la función 'main()' del programa y, si se desea, algunas otras funciones sencillas.

- **lib.h**: fichero de cabecera para la librería principal del trabajo, debe contener todas las declaraciones de estructuras y prototipos de funciones (las estructuras que se han descrito en la página anterior van en este fichero).

- **lib.c**: fichero con la definición completa de todas las funciones de la librería principal de la práctica.

MUY IMPORTANTE: Se recomienda implementar en la librería del trabajo (lib.h / lib.c), entre otras, todas las funciones necesarias para el tratamiento de listas, tanto para '**TVARS**' como '**TLIST**', incorporando funciones para:

➔ Crear una lista nueva vacía
➔ Eliminar toda una lista y liberando toda la memoria ocupada
➔ Incluir un nuevo elemento en la lista (al principio y al final)
➔ Buscar un elemento dentro de la lista
➔ Eliminar un elemento concreto de la lista
➔ etc…

Todo ello haciendo una buena gestión de memoria, reservando el espacio que sea necesario y liberándolo cuando corresponda, sin dejar basura en memoria.

SI EN ALGÚN MOMENTO, AL RESERVAR MEMORIA, SE DETECTA QUE HAY UN PROBLEMA Y NO SE PUEDE REALIZAR LA OPERACIÓN, EL PROGRAMA LO INDICARÁ CON UN MENSAJE DE ERROR EN ROJO COMO "***FALLO DE MEMORIA***" (o algo similar) Y TERMINARÁ LA EJECUCIÓN (en este caso no es necesario dar la respuesta de "*Bye, bye*").

Ampliación 1
Examen diciembre 2022

Ejercicio 1 (2.5 puntos)

Nuevo comando **`sum`**, devuelve la suma de todos los elementos de una lista. El resultado se imprime por pantalla con un número de decimales apropiado.

`sum L` *(devuelve un número, L puede ser una lista explícita, implícita o una variable)*

Ejercicio 2 (2.5 puntos)

Nuevo comando **`turn`**, devuelve una lista con los elementos en orden inverso al de la lista dada. El resultado se imprime por pantalla o se asigna a una variable si el comando es la parte derecha de una asignación.

`turn L` *(devuelve una lista, L puede ser una lista explícita, implícita o una variable)*

Ejercicio 3 (2.5 puntos)

Nuevo comando **`order`**, devuelve una lista con los elementos ordenados de forma ascendente. El resultado se imprime por pantalla o se asigna a una variable si el comando es la parte derecha de una asignación.

`order L` *(devuelve una lista, L puede ser explícita, implícita o variable)*

Ejercicio 4 (2.5 puntos)

Nuevo comando **`map`**, devuelve una lista donde cada elemento será la suma de 'n' con cada elemento de 'L'. El resultado se imprime por pantalla o se asigna a una variable si el comando es la parte derecha de una asignación.

`map n L` *(devuelve una lista, n: número real, L: lista explícita, implícita o variable)*

 # En todos los ejercicios, cuando corresponda mostrar una salida por pantalla, se hará en color verde, salvo que sea un mensaje de error, que se hará en color rojo.

COMANDOS PARA VALIDAR EL FUNCIONAMIENTO DE LA PRÁCTICA
Y PARA LA CORRECCIÓN DEL EXAMEN - Diciembre 2022

https://docs.google.com/document/d/1zoz6UQzQ1y5ysJRTNWGnoXuYMjUdQantrAPDtF-St5E/edit

Ampliación 2
Examen enero 2023

Ejercicio 1 (2.5 puntos)

Modificar el comando **`vars`** para que muestre por pantalla, por cada variable, su nombre, número de elementos, sumatorio y promedio con el siguiente formato:

`nomVariable: #n, sum=valor, prom=valor` *('n' → nº de elementos de la lista)*
`nomVariable: [vacia!]` *(dar esta salida cuando la variable sea una lista vacía [ ])*

Ejercicio 2 (2.5 puntos)

Modificar el comando **`order`** para que admita el parámetros opcional '**`asc/des`**'. La lista resultante estará ordenada ascendentemente o descendentemente según indique el parámetros (si no hay parámetro el orden será ascendente por defecto).

`order <asc/des> L` *(devuelve una lista, L puede ser explícita, implícita o variable)*

Ejercicio 3 (2.5 puntos)

Nuevo operador **`?`** (interrogación), devuelve una lista igual a la lista 'L' añadiendo el nuevo valor 'n' al final en el caso de que no exista en 'L', si 'n' sí que existe dentro de 'L', entonces se elimina dicho valor de la lista resultante.

`L ? n` *(devuelve una lista, L puede ser una lista explícita, implícita o una variable)*

Ejercicio 4 (2.5 puntos)

Nueva lista implícita con operador **`*`** (asterisco), crea una lista donde el primer valor se repite tantas veces como indique el segundo valor:

`[v * n] ⇒ [v v ... v]` *(v se repite n veces, si n=0 → lista vacía, si n<0 → **error**)*

 # En todos los ejercicios, cuando corresponda mostrar una salida por pantalla, se hará en color verde, salvo que sea un mensaje de error, que se hará en color rojo.
