Práctica III.- Generador de expresiones simbólicas.

I.- Introducción.

En esta tercera y última práctica obligatoria del curso, vamos a construir un generador de expresiones simbólicas, basado en los conocimientos teóricos explicados sobre árboles, recursividad y tablas hash.

La forma de realizar esta práctica, difiere en gran medida de la forma en que han sido realizadas las 2 prácticas previas de este curso. En esta ocasión, se pone a vuestra disposición la definición de la estructura de datos que todos debéis utilizar, que se corresponde con un TAD con forma de árbol para representar expresiones simbólicas. Con el fin de que la práctica sea lo más corta posible, en el código fuente puesto a vuestra disposición se han programado las operaciones necesarias para el trabajo con dicho TAD, de forma que todas las funciones que se piden para realizar esta práctica, deben utilizar las operaciones ya implementadas.

El código fuente puesto a vuestra disposición, será actualizado con el paso de las semanas, añadiéndose al mismo las funcionalidades desarrolladas en las prácticas semanales de laboratorio. Cada semana se irán añadiendo nuevas funcionalidades a dicho código fuente, de tal forma que es recomendable que descarguéis semanalmente los nuevos ficheros fuente, para que podáis tener actualizada la versión del código con la que iremos trabajando.

II.- Visión general de la práctica.

La práctica consiste en completar una serie de funciones para continuar con el desarrollo del analizador de expresiones. En el código fuente desde el que partimos, se han programado las operaciones del TAD expresión simbólica y alguna operación auxiliar. En las clases de laboratorio desarrollaremos alguna funcionalidad más y, por último, deberéis programar 3 funciones para trabajar con el analizador de expresiones, que son las que marcan el objetivo de esta práctica.

Una expresión, consta de una cabecera y una secuencia de sub-expresiones, a las cuales se aplica dicha cabecera. Por ejemplo, la expresión Cos[x], tiene como cabecera “Cos”, y ésta se aplica a una única sub-expresión, “x”.

El usuario será el encargado de introducir **una expresión válida** en el intérprete, el cual se os facilita ya implementado. La expresión introducida por el usuario, deberá ser siempre válida. De esta forma, no es necesario realizar en ningún caso un control sintáctico de las expresiones introducidas, todas deberán ser válidas.

Introducida la expresión por el usuario como una cadena de caracteres, ésta es transformada a una expresión simbólica, apoyándose para ello en el TAD expresión simbólica especificado en la unit ExprTree. La función encargada de realizar esta conversión, aparece implementada en la Unit ExprParser (function ParseExpr).

Es en este momento, en el que la expresión introducida la tenemos en memoria representada como el TAD expresión simbólica, cuando aplicaremos la cabecera de la función que nos hayan pasado a los nodos del árbol, para obtener una nueva expresión ya evaluada. La evaluación de las expresiones en el árbol, se realiza desde los nodos terminales hacia la raíz. Las funciones que se os pedirán, por lo tanto, trabajan sobre este árbol de expresión simbólica.

En la evaluación de la expresión simbólica, podemos distinguir 3 casos:

1.- Que la aplicación sepa como evaluar dicha expresión. Como es lógico, lleva a cabo la evaluación, y devuelve el resultado (una nueva expresión) al intérprete de comandos.

2.- Que la aplicación desconozca la forma de evaluar dicha expresión. En este caso, se limitará a devolver una copia exacta de la expresión introducida al intérprete de comandos. (Aunque sea una copia, también es una nueva expresión).

3.- Híbrido entre los casos 1 y 2: de una expresión, pueden existir partes (sub-expresiones) que el núcleo de la aplicación sabe como evaluar, y otras que desconoce. Aquellas que sabe evaluar, las evalúa devolviendo una nueva expresión resultado de la evaluación, y aquellas que desconoce, se limita a copiar como salida una nueva sub-expresión idéntica a la sub-expresión introducida.

Realizada la evaluación de la expresión, y por tanto obtenida una nueva expresión simbólica, ésta es transformada nuevamente en cadena de texto y mostrada en el intérprete de expresiones, llevando a cabo la liberación de toda la memoria asociada al TAD expresión simbólica.

El árbol asociado a cada expresión, permanecerá en memoria durante un breve instante de tiempo, el necesario para llevar a cabo la evaluación de la expresión y mostrar el resultado. Mostrado dicho resultado, toda la memoria que ha sido asociada a dicha expresión simbólica será liberada. Por ello, nuestro intérprete presenta un par de comandos especiales, que nos permitirán "almacenar" de forma temporal las expresiones que hayamos introducido:

1.- SET[valor, expresión]: Asocia un nombre a una expresión que introduzcamos.
2.- %: Obtiene como resultado la última expresión evaluada.

De esta forma, podemos asociar valores a expresiones. Ejemplos:

- Set[A1,{a,b,c}] ➔ Al valor A1 se asocia la expresión {a,b,c}
- Set[A2,Cos[y]] ➔ Al valor A2 se asocia la expresión Cos[y]

III.- Detalle del funcionamiento de nuestra práctica.

En Agora tenéis disponible el código fuente de la práctica desde el que partiremos. A lo largo del curso, éste código fuente irá siendo actualizado con las prácticas realizadas en el laboratorio, hasta terminarlas y obtener una versión completa y definitiva que os permita desarrollar las 3 funciones pedidas.

A) Descripción de las unidades facilitadas en el código fuente.

El código fuente, consta de las siguientes partes:

- **Program FrontEnd:** Entrada de nuestro código fuente. Es el encargado de gestionar el intérprete de comandos. Se encarga de leer la cadena de la expresión introducida y pasarla al núcleo de la aplicación para que sea evaluada.
- **Unit ExprTree:** Detalla el TAD que utilizaremos para representar el árbol con la expresión a analizar, basándonos en el TAD lista doblemente enlazada y en el TAD expresión simbólica. Tiene programadas todas las operaciones para la utilización de ambos TAD. Además, define un iterador sobre la lista, que será la forma en la que accederemos a los elementos de ésta. Contendrá las funciones que permitirán pasar del TAD árbol de expresión a la representación de la misma en formato texto para el intérprete.
- **Unit ExprParser:** Se encarga de transformar la expresión introducida como cadena en el intérprete, en una expresión de tipo "Expr", correspondiéndose con el TAD expresión simbólica especificado en ExprTree.
- **Unit CoreFunctions:** Implementará funciones fundamentales para la realización de la práctica y que serán desarrolladas en las prácticas de laboratorio, tales como obtener una copia en memoria de una expresión dada o comparar 2 expresiones en memoria.
- **Unit Kernel:** Es el núcleo de la aplicación. Gestiona la tabla hash que os permite de forma transparente almacenar el par (clave, valor) que os facilita la no repetición de código mediante el comando SET, y la asociación de una clave a una expresión introducida. Además se encarga de evaluar la expresión que ha transformado ExprParser, llamando a la función que la tendrá que evaluar si está programada, o copiando como resultado la misma expresión introducida si no está entre las funciones programadas.

B) Detalle de la estructura de datos utilizada para representar expresiones.

La declaración del TAD expresión simbólica, que será el utilizado para representar en memoria el concepto abstracto de árbol, a partir de la expresión introducida en el intérprete, está declarado en la Unit ExprTree, unidad básica y fundamental para desarrollar la práctica.

```
{ TAD Expresión Simbólica - definición adelantada }
Expr = ^TExpr;

{ TAD - Lista de expresiones }
{ nodo en la lista }
PNodoExprList = ^TNodoExprList;
TNodoExprList = record
        Element : Expr;
        { doblemente enlazada }
        Siguiente, Previo : PNodoExprList;
end;

{ lista }
TExprList = record
        Primero, Ultimo : PNodoExprList;
        nNodos : Word;
end;

{ TAD Expresión Simbólica - definición de tipos }
TExpr = record
        Head : String; Terminal : String; SubExprs : TExprList;
end;
```

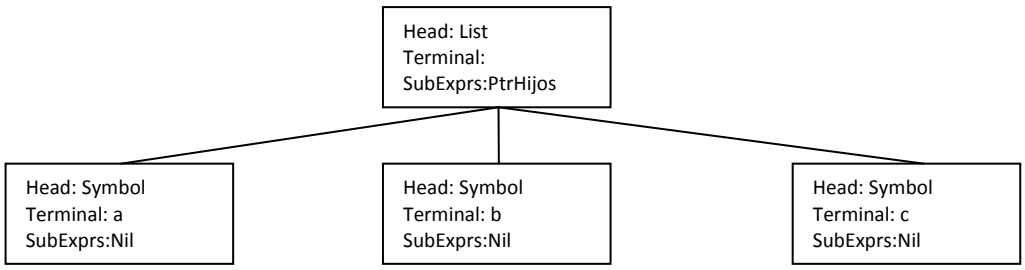
C) Sintaxis aceptada por el intérprete

- ✓ **Símbolos:** Son caracteres o cadenas constantes que no tienen ninguna función asociada. Ejemplos:
  - 25
  - Leon
- ✓ **Listas:** Irán encerrados entre llaves los distintos elementos que componen una lista. Una lista, como es lógico, puede estar formada por símbolos, otras listas, funciones…
  - {a,b,c}
  - {a,{a,b,d},d,e,c,e,f}
- ✓ **Funciones:** Estará formadas por el nombre de la función y, entre corchetes, los parámetros que dicha función recibe. Ejemplos:
  - Cos[x]
  - Flatten[{a,b,c,{e,f},h}]
- ✓ **Carácter "%":** Devuelve el valor de la última expresión analizada.
- ✓ **Carácter "?":** Presenta la ayuda de la aplicación.
- ✓ **Cadena "<<NombreFichero":** Permite cargar un fichero con una colección de expresiones válidas.
- ✓ **Cadena ">>NombreFichero":** Guarda en el nombre del fichero especificado, todas las expresiones que estén en ese instante almacenadas en memoria.
- ✓ **SET[variable, expresión]:** Almacena en el nombre de la variable introducida, la expresión que pasemos. De esta forma, podemos evitar volver a escribir una expresión varias veces si previamente la hemos asociado con algún valor mediante este comando.

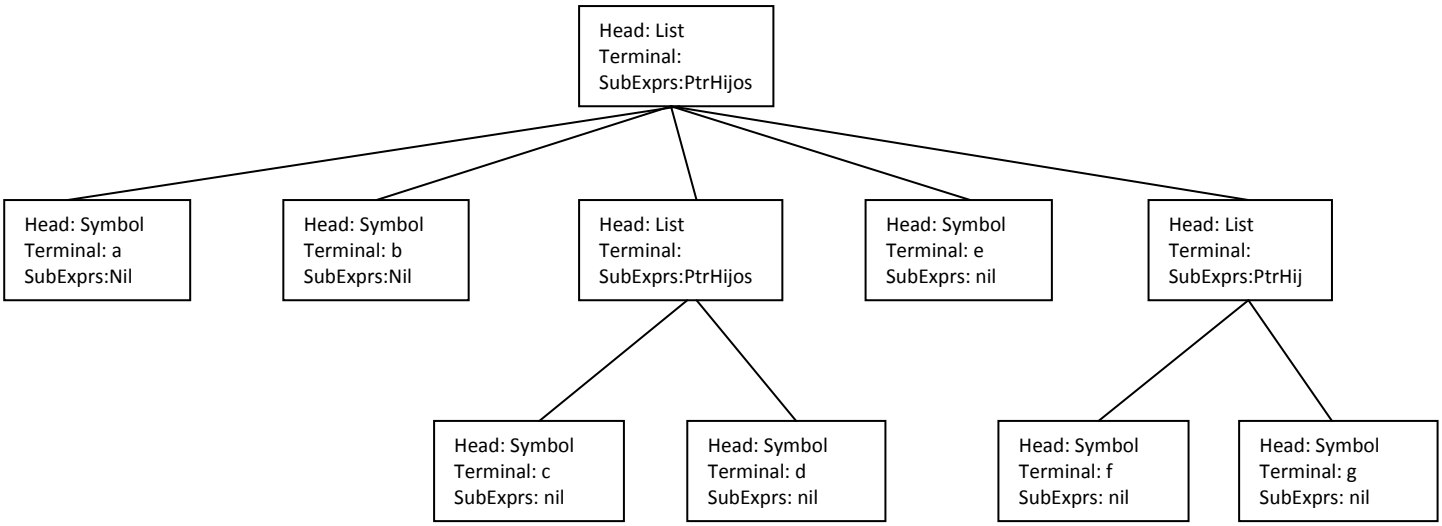
D) Árbol Generado a partir de la expresión introducida.

✓ Símbolos: El ejemplo representa el símbolo A. Tiene asociado el siguiente árbol de expresión:

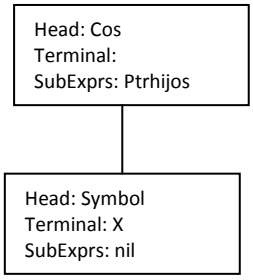
✓ Listas: La lista *{a,b,c}* tiene asociado el siguiente árbol de expresión:



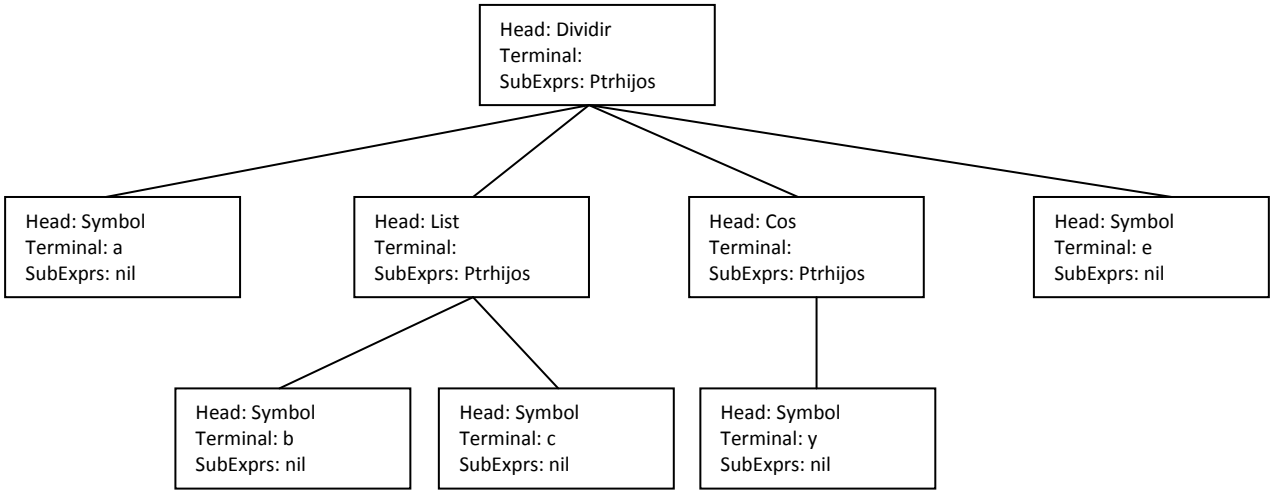
Ejemplo para la lista *{a, b, {c, d}, e, {f, g}}*



✓ Funciones: Árbol asociado a la función *Cos[x]*



Árbol asociado a la función *Dividir[a,{b,c},cos[y],e]*



IV.- Descripción de las operaciones programadas.

A) Operaciones para trabajar con la lista doblemente enlazada.

Las operaciones utilizadas para trabajar con la lista doblemente enlazada son las siguientes:

- ***InitTExprList (var L : TExprList) :*** Inicializa la lista de expresiones a nil.
- ***ReleaseElementsInTExprList (var L : TExprList) :*** Elimina todos los nodos y expresiones de la lista.
- ***InsertAsFirst (var L : TExprList; x : Expr) :*** Inserta un nuevo nodo como primero de la lista.
- ***InsertAsLast (var L : TExprList; x : Expr) :*** Inserta un nuevo nodo como último en la lista.
- ***IsEmtpyTExprList (L : TExprList) : Boolean :*** Devuelve cierto si la lista está vacía.
- ***LengthOfTExprList (L : TExprList) : Word :*** Devuelve la longitud de la lista

B) Operaciones para trabajar con la expresión simbólica.

Las operaciones programadas para trabajar con la expresión simbólica son las siguientes:

- ***AllocExpr (Head : String; Terminal : String) : Expr :*** Inicializa una expresión simbólica con los valores pasados en su cabecera y terminal, e inicializa su lista de sub-expresiones a nil.
- ***ReleaseExpr (var x : Expr) :*** Libera la memoria dinámica asociada a una expresión y sus sub-expresiones.
- ***AddSubExpr (x : Expr; var ToExpr : Expr) :*** Añade una expresión como última sub-expresión.
- ***LengthOf (x : Expr) : Word:*** Devuelve la longitud de la expresión.

C) Utilización de iteradores para recorrer el árbol.

En el diseño de software, un iterador define una interfaz en la que se declaran las operaciones necesarias para acceder secuencialmente a una colección de objetos, sin necesidad de conocer la estructura de datos interna de cada colección.

La forma de recorrer el TAD expresión simbólica, se realiza de forma más eficiente mediante la utilización de iteradores. Desde cada nodo del árbol de expresión simbólica, tenemos el puntero (SubExprs) a una lista doblemente enlazada que son los hijos de dicho nodo. Para recorrer cada lista doblemente enlazada, que es una colección de nodos de una lista del árbol, utilizamos un iterador que vamos posicionando en cada nodo de dicho árbol. La definición de la estructura utilizada para el iterador, está declarada en la Unit ExprTree, y es la siguiente:

```
{ iterador sobre la lista }
TExprIt = record
        Nodo : PNodoExprList;
end;
```

Las operaciones que disponbles para recorrer cada expresión simbólica con iteradores, son las siguientes:

- ***MoveToFirst (L : TExprList; var K : TExprIt) :*** Coloca el iterador apuntando al primer nodo de la lista.
- ***MoveToLast (L : TExprList; var K : TExprIt) :*** Sitúa el iterador apuntando al último nodo de la lista.
- ***MoveToNext (var k : TExprIt) :*** El iterador pasa a apuntar al siguiente nodo a su derecha respecto al nodo que está apuntando en su llamada. Si el iterador apuntaba al último nodo de la lista, pasa a apuntar a nil.
- ***MoveToPrevious (var k : TExprIt) :*** El iterador pasa a apuntar al nodo que está a la izquierda del nodo al que apunta actualmene. Si estaba apuntando al primer nodo de la lista, pasa a apuntar a nil.
- ***IsAtNode (k : TExprIt) : Boolean :*** Devuelve true si el nodo al que apunta el iterador es distinto de nil.
- ***ExprAt (k : TExprIt) : Expr :*** Devuelve la expresión almacenada sobre el nodo al que apunta el iterador.
- ***ExprAtIndex (L : TExprList; n :Word) : Expr :*** Devuelve la expresión almacenada sobre el número de nodo de la lista que se pasa.
- ***RemoveNodeAt (var L : TExprList; var k : TExprIt) : Expr :*** Devuelve la sub-expresión a la que apuntaba el iterador y cuya memoria es liberada, y mueve el iterador al siguiente elemento de la lista.
- ***RemoveNodeAndReleaseExprAt (var L : TExprList; var k : TExprIt) :*** Elimina la expresión contenida por el nodo al que apunta el iterador. El iterador pasa a apuntar al siguiente nodo de la lista.
- ***SwitchExprAt (k : TExprIt; WithExpr : Expr) : Expr :*** Intercambia la expresión a la que apunta el iterador por una nueva, devolviendo la expresión a la que apuntaba antes de ser sustituida.
- ***InsertBefore (k : TExprIt; var InList : TExprList; X : Expr) :*** inserta una expresión X en la lista, delante del nodo sobre el que está el iterador.
- ***InsertAfter (k : TExprIt; var InList : TExprList; X : Expr):*** inserta una expresión X en la lista, detrás del nodo sobre el que está el iterador.

V.- Funciones a implementar para superar la práctica.

Las funciones a implementar tendrán la forma:

```
Function F(E1, E2, … : Expr; var ec : TException ] : Expr;
```

Las expresiones E1, E2, … no deben ser modificadas dentro del código de la función. Pueden ser recorridas, consultadas, etc., pero nunca modificadas.

El resultado devuelto por las funciones debe ser una nueva expresión, y su memoria dinámica pasa a ser propiedad de quien invoque la función.

La variable ec : TException, está definida en la Unit ExprShared, como un registro que nos permita devolver un número de error y una cadena con la descripción de dicho error.

Veamos un Ejemplo: Programar la función First que devuelva la primera sub-expresión de una expresión pasada. En caso de que no tenga sub-expresiones, devolverá el puntero apuntando a nil y un mensaje de error. *(La función DeepCopy será implementada en las prácticas de laboratorio, y se encarga de obtener una copia nueva de la expresión pasada).*

```
function First (A : Expr; var ec : TException) : Expr;
{Recibe una expresion, y devuelve la primera sub-expresión de dicha expresión. Si no tiene, devuelve un error.}
        var
                K : TExprIt; {Iterador para recorrer el árbol}
        Begin
        {Comprobamos si la lista de sub-expresiones tiene al menos un elemento.}
        If (LengthOfTExprList(A^.SubExprs) >= 1) then begin
                {El resultado debe ser una expresión, luego obtenemos una copia de la sub-expresión. Antes de obtener la
                copia, debemos situar el iterador en la primera sub-expresión de A}
                MoveToFirst(A^.SubExprs, k);
                        {Copiamos dicha expresión como resultado}
                First:=DeepCopy(ExprAt(k));
        end
        else begin
                {No se cumple una precondición, ya que la expresión no tiene sub-expresiones.}
                Ec.nError:=1; ec.Msg:='La expresión no tiene sub-expresiones';
                First:=nil;
        end;

End; {Fin de la función}
```

A) Implementar la función ReplaceAll((X : Expr; Y : Expr; var ec : TException) : Expr;

Esta función devolverá una expresión con las apariciones de E1 en X sustituidas por E2.
X – Expresión origen en la que buscar y sustituir.
Y – Expresión de la forma Rule[E1,E2]
E1: Expresión a encontrar en X.
E2: Expresión que sustituirá cada E1 encontrada en X.

Ejemplos:
ReplaceAll[{H,D,S},Rule[D,Z] ➔ *{H,Z,S}*
ReplaceAll[{A,B,C,{B,Z}},Rule[B,Cos[x]]] ➔ *{A,Cos[x],C,{Cos[x],Z}}*

B) Implementar la función Tally(X: Expr; var ec : TException) : Expr;

La función Tally recibe una expresión, y devuelve una nueva expresión (lista) en la que se contabilizan las diferentes apariciones de los diferentes elementos de X. Para cada sub-expresión E de X, aparecerá un elemento en el resultado, indicando el número de veces que la sub-expresión E aparece en X. El elemento será 'List[E,n]'.

Ejemplos:
Tally[{A,A,A}] ➔ *{{A,3}}*
Tally[{A,B,C,A,B,C,Times[2,x],Times[A,B,C]}] ➔ *{{A,2},{B,2},{C,2},{Times[2,x],1},{Times[A,B,C],1}}*

C) Implementar la función Flatten(X: Expr; var ec : TException) : Expr;

La función Flatten, recibirá una expresión a procesar (X), y se encargará de eliminar todas las listas anidadas que contenga, hasta que o bien las ha terminado de procesar todas porque son todas listas, o bien encuentra una expresión que no es "list", en cuyo caso no continúa descendiendo por el árbol y simplemente copia el resto al resultado.

Ejemplos:

Flatten[{A,{B,C},D}] ➔ *{A,B,C,D}*
Flatten[{A,{B,{C,{D,E,F},G},H}} ➔ *{A,B,C,D,E,F,G,H}*
Flatten[{a,b,c,{1,2,Cos[{1,2,3}],3},d}] ➔ *{a,b,c,1,2,Cos[{1,2,3}],3,d}*

En los dos primeros ejemplos, como todo lo que hay son listas, las elimina, dejando una sola lista plana. En el tercer ejemplo, al llegar a la función "Cos" que no es de tipo "List", termina su función y copia como resultado literal el resto de la expresión.

VI.- Observaciones.

- No es necesario realizar control de errores de Entrada/Salida. Cualquier expresión introducida debe ser sintácticamente válida. En caso de que no lo sea, el programa terminará su ejecución de forma errónea.
- Se prohíbe el uso de las funciones: delay, GotoXY, WhereX, WhereY, Textcolor, TextBackground, sound, Nosound, Exit. También se prohíbe usar las unit Graph y crt.
- **No se aceptarán prácticas que presenten efectos laterales en su código.**
- Es **Obligatorio** que el código esté **identado y comentado**.
- Para cada función y procedimiento han de figurar los siguientes datos en los comentarios:
  - Descripción del valor de retorno (en caso de funciones).
  - Descripción de los parámetros de la función o procedimiento (si son de entrada o salida y para qué se utilizan).
  - Pequeña descripción de lo que hace la función o procedimiento.

A) Normas de entrega

- El programa debe implementarse utilizando el compilador Free Pascal versión 2.2.2 (no sirve otra versión) **en plataforma Linux (NO SIRVE OTRA PLATAFORMA).** Para la corrección se compilará el código fuente y posteriormente se ejecutará el programa. Dicha compilación/ejecución se realizará en una máquina con Linux y el compilador Free Pascal. Si el alumno utilizase otro compilador o sistema operativo para desarrollar el programa **es responsabilidad suya** el comprobar que funciona bajo las condiciones mencionadas.
- Se puede realizar la práctica individualmente o en grupos de **dos personas**. En caso de hacerlo en grupo, se entregará una práctica por grupo. La defensa de la práctica se realizará **de forma individual.**
- La entrega de las prácticas ha de seguir las siguientes normas:
  - Hay que entregar código fuente y los ejecutables **para sistemas Linux**.
  - Deberéis entregar un documento PDF -de no más de una hoja- en el que detalléis de forma muy breve lo siguiente:
    - 1.- Encabezado con nombre, apellidos y dni del autor/autores de la práctica.
    - 2.- Breve descripción y funcionamiento de cada una de las 3 funciones pedidas.
  - La entrega se realizará a través de Agora, con la cuenta de uno de los alumnos que la presente, **sólo la de uno**. Debéis subir un sólo fichero **comprimido en zip** que contenga todos los ficheros fuente (los que os dejamos en agora y los que vosotros desarrolléis), el ejecutable para Linux y el documento PDF. El nombre del fichero será:

    Generador–DniAlumno1letra–DniAlumno2letra.zip

    Para dos alumnos cuyos DNI sean 12345678-Z y 87654321–X, el fichero se llamará:

Generador–12345678Z–87654321X.zip

Fijaos que la letra está pegada al número del documento. **El formato del fichero comprimido debe ser ZIP.**

B) Forma de corrección.

Los ficheros de prueba estarán disponibles en Agora después de la segunda práctica de laboratorio relacionada con esta práctica obligatoria.

NO SERÁN APTAS aquellas prácticas que:

- Carezcan del documento pdf con la explicación solicitada.
- Carezcan del ejecutable de Linux, o, por el contrario, incluyan sólo los ejecutables para otros sistemas operativos.
- Presenten fallos en 1 sola de las baterías de prueba, lo que indicará que al menos una de las 3 funciones implementadas es errónea.
- No liberen toda la memoria dinámica que se reserve para la ejecución de la aplicación.

Las prácticas serán compiladas, entre otras, con las opciones –Cr y –gh. La primera comprueba que no se producen salidas de rango en vuestras prácticas. La segunda comprueba que liberáis toda la memoria que habéis reservado, algo que es **obligatorio.**

VII.- Fechas de entrega y defensa:

En esta ocasión, la fecha de entrega **tiene hora límite** antes de la finalización del día señalado.

Para superar por completo esta tercera práctica, cada alumno, de forma individual, deberá superar una defensa de la misma, que tendrá lugar en el laboratorio E5.

- ✓ **Fecha límite de entrega: 11 de junio de 2010, 15:00 horas.**
- ✓ **Fecha defensa: 15 de junio de 2010, 17:00 horas, laboratorio E5.**

*Os recuerdo que para realizar la defensa, debéis conocer el usuario y la contraseña de vuestra cuenta personal **para esta asignatura** en los ordenadores del **laboratorio E5** (Usuario EDI0910email--Password xxxxxxx). Quienes desconozcan esta información, pueden obtenerla en las prácticas semanales de laboratorio, o bien contactando conmigo a través de Agora. **El día de la defensa NO SE FACILITARÁ el usuario o la contraseña a ningún alumno.***