Práctica 11.- Funciones Partición y sub-expresión.

- **Objetivo de la práctica:** Realizar funciones similares a las solicitadas en la práctica obligatoria III, que permitan al alumno acercarse al conocimiento profundo de la estructura de datos utilizada y de las operaciones previamente programadas.
- **Observación:** Los ejercicios que se realicen desde este momento hasta el final, se programarán en la "Unit Ejercicios", que está presente en la versión 1.13 del código fuente publicado en Agora.

Ejercicio 1.- Agrupar los elementos de una lista.

Este primer ejercicio, consiste en programar una función que denominaremos *Partition*, que nos permita agrupar los elementos de una lista dada de n en n elementos.

Si el número de elementos de la lista original es múltiplo del valor n, aparecerán todos los elementos de la lista original en la lista resultado. Si por contra, el citado valor no es múltiplo de n, se despreciarán aquellos elementos de la lista original cuyo número no sea suficiente para completar el último grupo en la lista resultado.

Ejemplos:

Partition[{a,b,c,d,e},2] = {{a,b},{c,d}} ➔ El elemento "e" se desprecia.
Partition[{Cos[x],Sin[y],{z,w},{a,b}},3]={{Cos[x],Sin[y],{z,w}} ➔ El Elemento {a,b} se desprecia.
Partition[Sen[x,y],1] ➔ Error, la función *Partition* sólo trabaja sobre listas.

La pre-condición sobre la que trabaja esta función, es que recibe 2 parámetros de tipo *Expr* con el siguiente formato: el primer parámetro debe ser una lista (Head='List', Terminal = ''), y el segundo parámetro debe representar un valor numérico igual o superior a 1. (Head='Symbol', Terminal='n'). Para pasar de cadena de texto a valor numérico, recordad que existe la función *StrToInt*.

Por último, como tercer parámetro recibe una variable de tipo *Texception,* en la que deberemos devolver un error si la expresión pasada no es una lista, o si la segunda expresión no representa un número igual o superior a 1.

La función devolverá un tipo *Expr* que será una nueva lista, formada a su vez por sub-listas de tamaño n, en las que estarán agrupados de n en n los elementos de la lista original, exceptuando aquellos que sean despreciados por no completar la última sub-lista.

function Partition(X : Expr; n : Expr; var ec:Texception) : Expr;

Ejercicio 2.- Obtener una sub-expresión de una expresión.

En este segundo ejercicio, debéis programar una función que denominaremos *Part*, que nos permita posicionarnos en una sub-expresión de una expresión dada. Esta función recibe 3 parámetros, los 2 primeros de tipo *Expr* y el tercero de tipo *Texception*.

El primer parámetro de tipo *Expr* será la expresión que queremos recorrer, y el segundo parámetro *Expr,* será una expresión de tipo 'List', que representará la sub-expresión a la que queremos acceder.

Veamos un **ejemplo**:

Part[{a,b,{c,{d,e}},f},{3,2}] = {d,e}

La lista pasada como segundo parámetro, indica que queremos posicionarnos en el tercer elemento de la expresión original, y dentro de éste, en su segundo elemento. El anterior ejemplo sólo tiene listas en la expresión a recorrer, pero la función debe trabajar sobre cualquier tipo de expresión. Veamos otro **ejemplo**:

Set[A1,Sen[a,{b,c,d},{a,{b,c}}]
Part[A1,{2,2}] ➔ El resultado es "c"
Part[A1,{3,2}] ➔ El resultado es "{b,c}"
Part[A1,{3,3}] ➔ El resultado es "Sub-expresión inexistente"

Como podéis comprobar, se debe controlar el acceso a una sub-expresión inexistente en el árbol de la expresión pasada, lo que nos obligará a devolver un error en la variable de tipo **TException**. El error será mostrado de forma transparente para vosotros en el intérprete de comandos.

function Part(X : Expr; Spec : Expr; var ec : TException) : Expr;