I. Problemas aritméticos:

1. Escribir una función **`resto`** que calcule el resto de una división entera utilizando restas únicamente.

2. Escribir una función **`cociente`** que calcule el cociente de una división entera utilizando sumas y restas.

3. Escribir una función **`mcd`** que calcule el máximo común divisor de dos números enteros.

4. Escribir una función **`fact`** que calcule el factorial de un número.

5. Escribir una función **`vars`** tal que `vars n r` calcule el número total de variaciones de n elementos tomados de r en r.

6. Escribir una función **`combs`** tal que `combs n r` calcule el número total de combinaciones de n elementos tomados de r en r.

II. Manipulación de dígitos:

7. Escribir una función **`adj`**, con dos argumentos, que adjunte un dígito a la derecha de un número entero; e.d., tal que, por ejemplo, `adj 234 6 = 2346`.

8. Escribir una función **`sep`** que, a partir de un número entero con n dígitos, produzca el par resultante de la separación del último dígito de los restantes anteriores, con arreglo al siguiente comportamiento (sin utilizar la división)
   `sep 4 = (0,4); sep 235 = (23,5).`

9. Escribir una función **`enlazado`** que encadene los dígitos de dos números enteros positivos, con arreglo al comportamiento siguiente
   `enlazado 124 56 = 12456; enlazado 23 0 = 23.`

10. Escribir una función **`binario`** que calcule la representación binaria de un número entero dado en representación decimal y escribir su función recíproca **`decimal`**.

11. Escribir una función **`cambioAbase`** tal que `cambioAbase b n` calcule la representación, en base b menor que 10, del número n, dado en representación decimal.

III. Manipulación de cadenas de caracteres:

12. Escribir una función **`valorCar`** que convierta un número de un dígito en el correspondiente carácter, y su función recíproca **`valorNum`** que convierta un carácter numérico en el correspondiente número entero.

13. Escribir una función **`mayuscula`** que convierta los caracteres correspondientes a letras minúsculas en sus correspondientes mayúsculas y los demás caracteres los devuelva inalterados. Escribir también su función recíproca **`minuscula`**.

14. Escribir una función **`aMays`** que convierta palabras a mayúsculas.

15. Escribir una función **`precede`** con dos argumentos cadenas de caracteres, que determine si el primer argumento precede al segundo utilizando la ordenación lexico-gráfica y con independencia de la tipografía de las letras.

16. Suponiendo que el primer carácter de una cadena corresponde a la posición 0, escribir una función **`carPos`** tal que `carPos n cs` produzca el carácter de la cadena `cs` correspondiente a la posición `n`, y escribir otra función **`posCar`** tal que `posCar c cs` produzca la posición del carácter `c` en la cadena `cs` o un mensaje de error si el carácter no está incluido en la cadena.

17. Escribir una función **`posSubCad`** tal que `posSubCad ps cs` produzca la posición de la primera aparición de la cadena ps dentro de la cadena `cs`.

18. Escribir una función **`adjCar`** tal que `adjCar c cs` produza la cadena resultante de añadir el carácter `c` a la derecha de la cadena `cs`.

19 Escribir una función **`inversa`** para invertir cadenas de caracteres.

20. Escribir una función **`capicua`** para determinar si una cadena es simétrica.

21. Escribir una función **`permutada`**, con tres argumentos, que produzca una reordenación en una cadena, que figura como primer argumento, de acuerdo con la permutación que transforma la cadena que figura como segundo argumento en la que figura como tercer argumento; así, p.e.

```
permutada "AMOR" "abcd" "dabc"= "RAMO"
```

IV. Generación de sucesiones:

22. Escribir una función **`fib`** tal que `fib n` calcule el término n-ésimo de la sucesión de Fibonacci 0,1,1,2, ... teniendo en cuenta que los términos, a partir del segundo, se generan como suma de los dos anteriores. Buscar una solución eficiente.

23. Escribir una función **`raiz`** que calcule la raíz cuadrada de un número real `r` aplicando el algoritmo de Newton-Raphson que consiste en generar una sucesión

$$a_0 = \frac{r}{2}, \cdots, a_{n+1} = \frac{1}{2}\left(a_n + \frac{r}{a_n}\right), \cdots$$

hasta que la diferencia entre dos valores consecutivos sea 0; en cuyo caso el correspondiente término de la sucesión será la raíz buscada.

V. Definiciones de tipos:

24. Definir un tipo de datos **`Temperatura`** que contemple la posibilidad de utilizar grados Celsius y grados Fahrenheit y escribir una función **`conversion`** que transforme temperaturas de un formato a otro, de acuerdo a la siguiente relación

$$Cel = 5 \times \frac{Far - 32}{9}$$

25. Definir un tipo de datos **`Racional`**, correspondiente a los números racionales, que utilice una representación binómica (numerador,denominador). Escribir una función **`raNormal`** que simplifique la representación de un número racional a su forma irreducible. Escribir también las operaciones correspondientes a la suma, resta, multiplicación y división de números racionales, utilizando operadores infijos y tales que produzcan representaciones simplificadas.

26. Definir un tipo **`Real`**, correspondiente a los números con punto decimal, que utilice la representación (mantisa,exponente) de manera que cada número r se representa como `(m,e)`, siendo `r=m*10e`, con la restricción `3276<=m<32768`. Escribir una función **`reNormal`** que simplifique una representación general (mantisa,exponente) de un número real de manera que se ajuste a la restricción dada. Escribir también las operaciones correspondientes a la suma, resta, multiplicación y división de números reales, utilizando operadores infijos, de forma que produzcan representaciones normalizadas.

27. Definir un tipo de datos **`Complejo`**, correspondiente a los números complejos, que utilice las representaciones polar y binómica, y escribir las operaciones correspondientes a la suma, resta, multiplicación y división de números complejos utilizando operadores infijos.

28. Repetir los ejercicios anteriores definiendo los tipos **`Racional`**, **`Real`** y **`Complejo`** como instancias de la clase **`Num`** utilizando los operadores aritméticos habituales para denotar las operaciones.

29. Considerando las definiciones siguientes

```
data Punto  =  Pt Float Float
type Vector =  (Float,Float)
type Recta  =  (Punto,Vector)
type Figura =  [Punto]
```

definir funciones, para puntos y figuras, correspondientes a las siguientes transformaciones geométricas traslaciones según un vector, simetrías axiales y centrales, homotecias respecto a un centro con una razón dada y giros con un cierto centro y un ángulo dado.

30. Dado el siguiente tipo polimorfo correspondiente a secuencias no vacías

```
data Seq a = Mo a | a :< Seq a
```

escribir las funciones siguientes
lenSeq, que calcule la longitud de una secuencia;
catSeq, que encadene dos secuencias;
revSeq, que invierta una secuencia;
palSeq, que construya un palíndromo a partir de una secuencia dada y
seqToList, que construya una lista a partir de una secuencia manteniendo el orden de los elementos, y **listToSeq**, que haga lo contrario.

31. Definir un tipo polimorfo **Array10 a** para representar tablas de 10 elementos, junto con las funciones
act t p x, para actualizar la posición p de la tabla t con el valor x, y
pos t p, para consultar la posición p de la tabla t.

32. Definir un tipo polimorfo **Record a b c** para representar registros con 3 campos, junto con las funciones necesarias para consultar y modificar el contenido de cada campo.

33. Dado el siguiente tipo árbol correspondiente a expresiones aritméticas con números enteros

```
data Exp = Num Int|Op Exp Char Exp
```

donde se utiliza el tipo **Char** para los operadores aritméticos, escribir una función valor para calcular el valor de cualquiera de estos árboles. ¿Cómo se realizaría el ejercicio si en lugar de **Char** admitieramos datos de tipo **Int->Int->Int**?

34. Dado el siguiente tipo árbol

```
data Tree a = Leaf a|Node (Tree a) (Tree a)
```

escribir las funciones
leaf, que calcule el número de hojas de un árbol;
node, que calcule el número de nodos de un árbol;
maxpath, **minpath**, que calculen las longitudes de los caminos máximo y mínimo respectivamente; y
twist, que construya la imagen especular de un árbol.

35. Dado el tipo árbol

```
data Arbol a = Vacio|Nodo (Arbol a) a (Arbol a)
```

escribir las funciones
es_ord, que determine su un árbol está ordenado o no;
es_Heq, que determine su un árbol está equilibrado en altura o no;
es_Peq, que determine su un árbol está equilibrado en peso o no;
insertar, **eliminar**, que inserte o elimine, respectivamente, un elemento en un árbol suponiendo que el árbol está ordenado.

36. Definir un tipo polimorfo **ArbolG a** para representar árboles generales, no vacíos y con cualquier número de ramas (posiblemente ninguna), y escribir las funciones
leafG, que calcule el número de hojas de un árbol;
nodeG, que calcule el número de nodos de un árbol;
maxpathG, **minpathG**, que calculen las longitudes de los caminos máximo y mínimo respectivamente.

37. Repetir el mismo ejercicio anterior, pero suponiendo que los árboles generales pueden ser vacíos.

VI. Funciones de orden superior:

38. Definir una función **`aplicaSeq`** para aplicar una función a todos los elementos de una secuencia (tipo **`Seq`** del ejercicio 30).

39. Definir **`filtrar`** que aplicada a un predicado y una lista produzca la sublista formada por los elementos de la lista para los que el predicado se evalúa a **`True`**, y definir otra función **`rechazar`** con el comportamiento contrario.

40. Definir una función **`parList`**, con una función de dos argumentos y dos listas como argumentos, que construya una lista con los resultados de aplicar la función del argumento a los elementos correspondientes (situados en las mismas posiciones) de ambas listas, permitiendo listas con distinta longitud, p.e.:

```
parList (+) [3,4] [2,2,1] = [5,6]
```

41. Definir una función **`mapBtree`**`::(a -> b) -> (Arbol a) -> (Arbol b)`, para construir un árbol (del tipo definido en el ejercicio 35) con la misma forma que el que aparece en su segundo argumento, pero con los valores del tipo a cambiados por sus transformados por la función que aparece como primer argumento.

    Definir una función **`mapLtree`** parecida a la anterior pero operando sobre árboles de hojas (definidos en el ejercicio 34).

42. Definir una función **`parBtree`** que actúe como la función **`parList`** del ejercicio 40 pero sobre objetos del tipo **`Arbol a`**. Estudiar los problemas que se pueden plantear.

VII. Transformaciones de estructuras:

43. Definir las funciones siguientes, determinando su tipo más general

    a) **`compress`** que, aplicada a una función `f` y una lista $[a_1, a_2, \ldots, a_n]$ produzca como resultado el valor $f\ a_1\ (f\ a_2\ (\ldots(f\ a_{n-1}\ a_n)\ldots))$;

    b) **`condense`** que, aplicada a una función `f` una función `g` y una lista $[a_1, a_2, \ldots, a_n]$ produzca como resultado el valor $f\ a_1\ (f\ a_2\ (\ldots(f\ a_{n-1}\ (g\ a_n))\ldots))$;

    c) **`conSeq`** y **`conTree`** que se comporten igual que condense pero sobre elementos de los tipos **`Seq a`** y **`ArbolH a`** (ej. 47) respectivamente.

44. Con ayuda de la función **`conSeq`** construir:

    a) una función que transforme secuencias en listas (**`seqToList'`**);

    b) una función que calcule la longitud de una secuencia (**`lenSeq'`**);

    c) una función que sume todos los elementos de una secuencia (**`sumSeq`**);

    d) una función que invierta una secuencia (**`revSeq'`**).

45. Con ayuda de la función

```
redTree f b Vacio = b
redTree f b (Nodo izq r dch) = f r (redTree f b izq) (redTree f b dch)
```

para árboles del tipo definido en el ejercicio 35, definir funciones para
a) construir el espejo de un árbol;
b) calcular la altura de un árbol;
c) calcular la media de los elementos de un árbol recorriéndolo una sola vez;
d) calcular la suma, el producto y la suma de los cuadrados de los elementos de un árbol atravesándolo una sola vez.

46. Para el tipo de árbol definido en el ejercicio 35,
a) construir una función que determine si dos árboles tienen la misma forma (con independencia de los valores situados en los nodos);
b) construir una función para determinar si dos árboles tienen el mismo recorrido en inorden.

47. Para el tipo de árbol

```
data ArbolH a = H a|N (ArbolH a) (ArbolH a),
```

construir una función para determinar si dos árboles tienen la misma frontera (e.d., el mismo listado de hojas de izquierda a derecha).

VIII. Cálculos con estructuras infinitas:

48. **a)** Definir una función **`listasuc`** que genere la lista infinita de los números siguientes a uno dado.
b) Con la función anterior y la función **`filtrar`** construir una lista con los 10 primeros múltiplos de 7

49. **a)** Definir una función que construya una lista con los 100 primeros números primos.
b) Escribir una expresión para calcular el menor primo mayor que 10000.

50. Definir una función **`rep`** que genere la lista infinita resultante de la aplicación sucesiva de una función de un argumento a un cierto valor, e.d. tal que

```
rep f x = [x, f x, f (f x), ...].
```

Con ayuda de esta función, definir funciones para calcular
a) la lista de los múltiplos de 5,
b) la lista de potencias de 2,
c) la lista `[True, False, True, False, ....]`,
d) la lista `['*','**','***','****', ...]`.

51. **a)** Escribir una función eficiente para generar la lista infinita de los números factoriales.
b) Escribir una función eficiente para generar la lista infinita de los números de Fibonacci.

52. Un número es perfecto cuando es igual a la suma de todos sus divisores incluido el 1 (pero sin incluirse él mismo). Escribir una función que genere la lista de todos los números perfectos.

53. Teniendo en cuenta que el producto de las columnas de la expresión siguiente

```
1 2 3 4 5 6 7 ...
  1 2 3 4 5 6 ...
    1 2 3 4 5 ...
```

produce los factoriales de los naturales, escribir una función que calcule dicha lista.

54. Escribir expresiones para generar las listas siguientes (una por lista)

```
[[1,4,9, ...], [1,8,27, ...], [1,16,81, ...], ...]
[[],[(1,1)],[(2,1),(1,2)],[(3,1),(2,2),(1,3)], ...]
```

IX. Problemas varios:

55. (Representación de arrays) Un array es esencialmente una lista finita, con longitud fija, donde se suelen consultar y cambiar los elementos de las distintas posiciones. Con esta idea, una forma eficiente de representar arrays consiste en utilizar árboles de hojas equilibrados en peso. Una forma de hacer esto consiste en aplicar el criterio recursivo siguiente: *"los elementos de las posiciones pares de la lista van a la rama izquierda del árbol y los de las posiciones impares a la rama derecha y, dentro de cada rama, se vuelve a aplicar el mismo criterio con las sublistas que les corresponden*"

    a) Definir una función genérica **`mkarray`** que construya un árbol a partir de una lista aplicando el criterio anterior.

    b) Definir otra función genérica **`listarray`** que reconstruya la lista a partir de un árbol construido con la función anterior (árbol equilibrado en peso).

    c) Definir una función **`lookup`** para consultar elementos en el árbol a partir de sus posiciones en la lista.

    d) Definir una función **`update`** para cambiar elementos en el árbol.

56. (Arboles de Huffman) Para reducir el número total de bits requeridos en la codificación binaria de textos se utilizan códigos de longitud variable basados en las frecuencias de aparición de los distintos caracteres. Estos códigos se deben elegir de tal forma que ningún carácter se convierta en un prefijo de otro para evitar problemas en la decodificación. Un método para obtener un código óptimo con estas características (debido a David Huffman) consiste en construir un árbol de hojas a partir de los caracteres que se van a codificar y de sus frecuencias aplicando el procedimiento siguiente:

    1) a cada carácter se convierte en un árbol hoja y se le asigna su frecuencia,
    2) con los dos árboles de menor frecuencia, utilizados como rama izquierda y derecha respectivamente, se construye un nuevo árbol al que se le asigna como frecuencia la suma de las frecuencias de sus ramas, y se sigue así hasta obtener un único árbol.

    a) Definir una función **`cons_Htree`** que construya un árbol de Huffman a partir de una lista de pares (carácter,frecuencia).

    Con el árbol ya construido, a cada carácter se le asigna el número binario correspondiente al camino que conduce desde la raíz hasta la hoja donde está el carácter interpretando izquierda como 0 y derecha como 1.

    b) Definir una función **`codigo`** que produzca el número binario asignado a un caracter en un árbol de Huffman.

    c) Definir una función **`cod`** que produzca la cadena de dígitos binarios asignada a una cadena de caracteres por un árbol de Huffman.

d) Definir una función **`decod`** que produzca la cadena de caracteres correspondiente a una cadena de dígitos binarios de acuerdo con la codificación producida por un árbol de Huffman.

57. (Expresiones currificadas). Las expresiones funcionales se suelen expresar, a nivel sintáctico, mediante árboles generales con el símbolo de la función en la raíz y cada argumento como una rama del árbol; sin embargo hay una forma particular que consiste en expresarlas como árboles binarios de hojas, que se apoya en la currificación de las expresiones, de forma que una expresión f a b c se interpreta como ((f a) b) c y se convierte en el árbol

```
N (N (N (H 'f') (H 'a')) (H 'b')) (H 'c')
```

a) definir una función **`currificar`** que construya el árbol binario del tipo **`ArbolH Char`** (ejercicio 47) correspondiente a un árbol general del tipo **`ArbolG Char`** (ejercicio 36) de acuerdo con el criterio anterior;

b) definir una función **`decurrif`** que construya el árbol general correspondiente a un árbol binario de una expresión currificada.

58. Con el tipo

```
data Base = C Int | O Char
```

se pueden construir expresiones prefijas y postfijas como las siguientes

```
[O '-', C 14, O '+', C 6, C 3],  [C 14, C 6, C 3, O '+', O '-']
```

que representan la expresión 14 - (6+3). Con esta idea y con el tipo **`Exp`**, definido en el ejercicio 33, escribir sendas funciones que construyan árboles del tipo **`Exp`**, a partir de una lista **`[Base]`** que represente una expresión prefija o postfija respectivamente. Las funciones deberán dar error cuando las listas no correspondan a expresiones correctas.