1. **(1 punto)**
   Consideremos el tipo Haskell

   ```
   data Nat = Z | S Nat
   ```

   Inferir sistemáticamente el tipo de la función `f` definida como:

   ```
   f   Z   x = x Z
   f (S x) g = f x g
   ```

   Solución:

```
La declaración de tipos indicada da lugar a las siguientes suposiciones de tipo de partida:

  Z::Nat , S:: Nat -> Nat

Con ellas podemos empezar las distintas fases de la inferencia de tipos de f.

1.- Decoración de tipos
-------------------------
Realizamos algún atajillo con las decoraciones de lados izquierdos y derechos de reglas

(f::a Z::Nat x::a0)::a1 = (x::a0 Z::Nat)::a1
(f::a (S::Nat->Nat x::a2)::a3 g::a4)::a1 = (f:.a x::a2  g::a4)::a1

2.- Generación de restricciones
---------------------------------
a=Nat->a0->a1 , a0=Nat->a1 , a=a3->a4->a1 , Nat->Nat=a2->a3 , a=a2->a4->a1

3.- Resolución de ecuaciones por Martelli-Montanari
-----------------------------------------------------

En cada paso indicamos la ecuación utilizada y la regla de transformación usada.
En algún caso juntamos varios pasos en uno.

a=Nat->a0->a1 , a0=Nat->a1 , a=a3->a4->a1 , Nat->Nat=a2->a3 , a=a2->a4->a1
===> (primera ecuación, ligadura de la variable a)
a=Nat->a0->a1 , a0=Nat->a1 , Nat->a0->a1=a3->a4->a1 , Nat->Nat=a2->a3 , Nat->a0->a1=a2->a4->a1
===> (cuarta ecuación, descomposición)
a=Nat->a0->a1 , a0=Nat->a1 , Nat->a0->a1=a3->a4->a1 , Nat=a2 , Nat=a3 , Nat->a0->a1=a2->a4->a1
===> (reorientación en cuarta y quinta ecuaciones)
a=Nat->a0->a1 , a0=Nat->a1 , Nat->a0->a1=a3->a4->a1 , a2==Nat , a3==Nat , Nat->a0->a1=a2->a4->a1
===> (ligadura en cuarta y quinta ecuaciones, que se resitúan como 2a. y 3a.)
a=Nat->a0->a1 , a2==Nat , a3==Nat , a0=Nat->a1 , Nat->a0->a1=Nat->a4->a1 , Nat->a0->a1=Nat->a4->a1
===> (descomposición  en quinta y sexta ecuaciones)
a=Nat->a0->a1 , a2==Nat , a3==Nat , a0=Nat->a1 , Nat=Nat , a0=a4 , a1=a1 , Nat=Nat , a0=a4 , a1=a1
===> (eliminación trivial en 5a, 7a, 8a, 9a ecuaciones)
a=Nat->a0->a1 , a2==Nat , a3==Nat , a0=Nat->a1 , a0=a4 ,  a0=a4
===> (ligadura de a0 por a4 en 5a ecuación, que resituamos como )
a=Nat->a4->a1 , a2==Nat , a3==Nat , a0=a4 , a4=Nat->a1 ,  a4=a4
===> (eliminación trivial de 6a ecuación y ligadura de a4 en  5a ecuación)
a=Nat->(Nat->a1)->a1 , a2==Nat , a3==Nat , a0=Nat->a1 , a4=Nat->a1

Ya hemos llegado a una forma resuelta, pues no se puede aplicar ninguna transformación.
El tipo inferido para f es pues
f:: Nat->(Nat->a1)->a1
```

2. **(2 puntos)**
Supongamos el tipo Haskell

```
data Racional = Frac Int Int
```

para representar números racionales como fracciones.

(i) Definir una operacion `simp:: Racional -> Racional` para simplificar una fracción.
Notas:

- Se pueden utilizar funciones primitivas para *Int*, en particular *div:: Int* $\rightarrow$ *Int* $\rightarrow$ *Int* y *gcd:: Int* $\rightarrow$ *Int* $\rightarrow$ *Int* que calculan la división y el máximo común divisor de enteros, respectivamente.
- Para los racionales negativos suponemos que al simplificar es el numerador el que queda negativo.

Ejemplos: simp (Frac 18 6) = Frac 3 1 ; simp (Frac 18 12) = Frac 3 2 ; simp (Frac 18 (-12)) = Frac (-3) 2

(ii) Definir la suma de racionales, de modo que la suma quede simplificada

(iii) Definir *Racional* como instancia de la clase *Ord*.

Solución:

```
data Racional = Frac Int Int deriving Show

(i)
simp:: Racional -> Racional
simp (Frac x y)
  | y == 0    = error "Division por cero"
  | otherwise = ajustaSigno (Frac (div x z) (div y z))
                where z = gcd x y

--ajusta (Frac x y) ajusta los signos de la fraccion
ajustaSigno (Frac x y)
  | y == 0     = error "Division por cero"
  | x == 0     = Frac 0 1
  | y > 0      = Frac x y
  | otherwise  = Frac (-x) (-y)

(ii)
suma (Frac x y) (Frac x' y') = simp (Frac (x*y'+x'*y) (y*y'))

(iii)
-- Para definir Racional como instancia de Ord hace falta tenerlo como instancia de Eq
instance Eq Racional where
 frac == frac' = (x,y) == (x',y')
   where Frac x y   = simp frac
         Frac x' y' = simp frac'

instance Ord Racional where
 frac <= frac' = x*y' <= y*x'
   where Frac x y   = simp frac  -- también se podría usar ajustaSigno en lugar de simp
         Frac x' y' = simp frac'
```

3. **(1 punto)** Definir la función

```
prefs::  [a] -> [[a]]
```

`prefs xs` $=_{def}$ lista de todos los prefijos de `xs` *(da igual el orden en el que salgan)*

Ejemplo: prefs [1,2,3] = [[],[1],[1,2],[1,2,3]]

Solución:

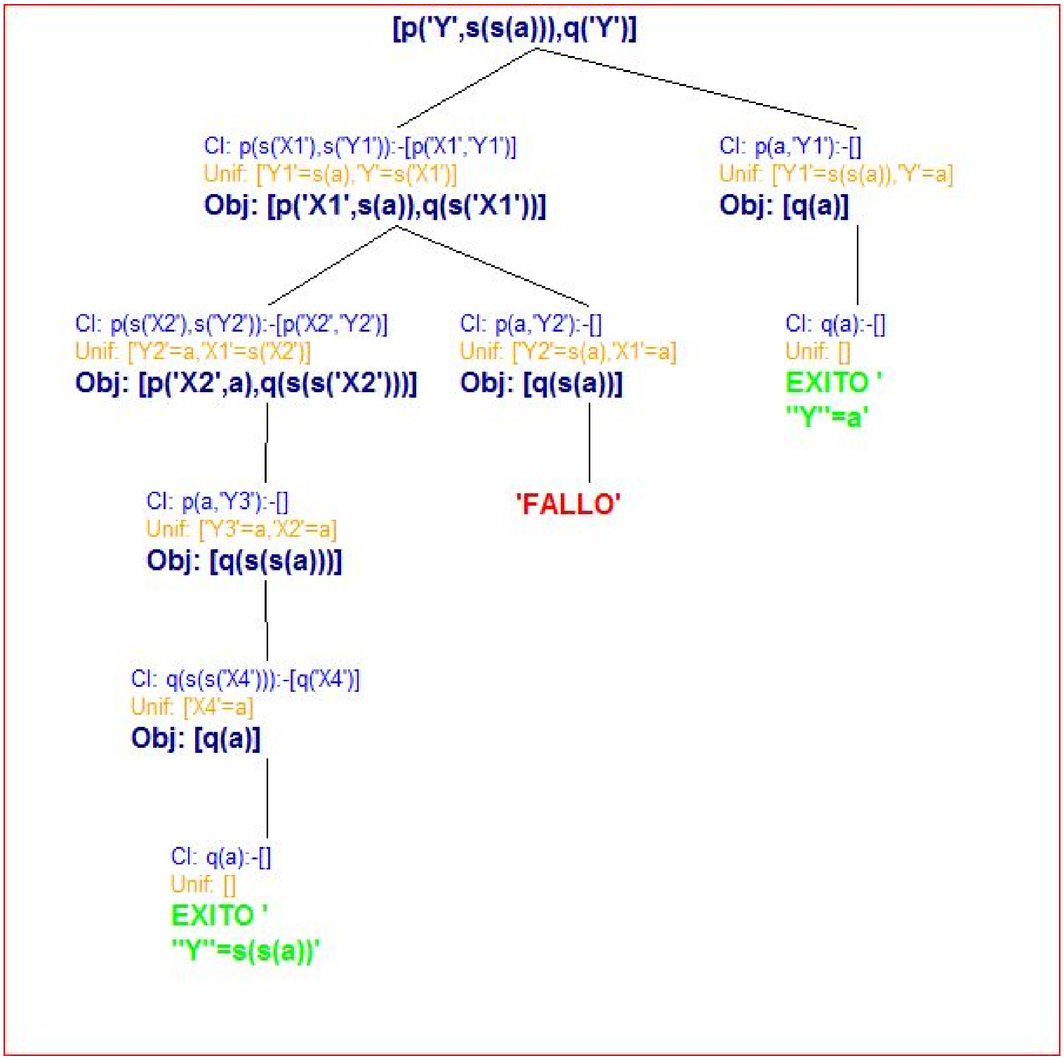
```
prefs []     = [[]]
prefs (x:xs) = [] : [x:ys| ys <- prefs xs]
-- Otra versión, sin usar listas intensionales
prefs' []     = [[]]
prefs' (x:xs) = [] : map (x:) (prefs' xs)
-- Aún otra, más fea, sin usar tampoco orden superior.
prefs'' []     = [[]]
prefs'' (x:xs) = [] : pega x (prefs'' xs)
  where pega x []       = []
        pega x (xs:xss) = (x:xs):pega x xss
```

4. **(1 punto)**
Dado el programa lógico

```
p(s(X),s(Y)) :- p(X,Y).          q(a).
p(a,Y).                          q(s(s(X))) :- q(X).
```

construye el árbol de resolución del objetivo `p(Y,s(s(a))),q(Y)`.

Solución:



5. **(1 punto)**
Programa en Prolog los siguientes predicados:

(a) `doble(Xs,Ys)` ⇔ la lista `Xs` tiene longitud doble que la lista `Ys`.
(No pueden usarse predicados primitivos).

(b) `ceros(T,N)` ⇔ en el término `T` aparece `N` veces el número 0.
(Se puede suponer el número natural `N` *representado mediante las constructoras* `cero/0` *y* `s/1`*, o bien mediante números primitivos del sistema, como cada uno prefiera).*

Ejemplo: `ceros(f(X,0,g(0,1,f(0,Y,a))),N)` *debe tener éxito dando* `N=s(s(s(cero)))` *(o* `N=3` *si se ha preferido usar números primitivos del sistema)*

Solución:

```
(a)
doble([],[]).
doble([_,_|Xs],[_|Ys]) :- doble(Xs,Ys).

(b)
% Utilizando números primitivos para contar
ceros(X,1) :-
   X==0,!.
ceros(X,0) :-
   var(X).
ceros(X,N) :-
   nonvar(X),
   X =.. [_|Args],
   ceros_l(Args,N).

% ceros_l(Xs,N) <-> N es el número de ceros que aparecen en los elementos de la lista Xs
ceros_l([],0).
ceros_l([X|Xs],N) :-
   ceros(X,N1),
   ceros_l(Xs,N2),
   N is N1+N2.
```

1. **(1 punto)**
   Considerando el tipo Haskell

   ```
   data Nat = Z | S Nat
   ```

   Inferir sistemáticamente el tipo de la función `f` definida como:

   ```
   f x  Z     = x Z []
   f y (S x)  = y x [x]
   ```

2. **(1,5 puntos)**

   (i) Definir en Haskell un tipo `Arbol a b` para representar árboles binarios con información de tipo `a` en los nodos internos y de tipo `b` en las hojas.

   (ii) Indicar el tipo y definir la función especificada por
   `info t` $=_{def}$ `(xs,ys)` siendo
   `xs` la lista de informaciones en los nodos internos de `t`
   `ys` la lista de informaciones en las hojas de `t`

   (iii) Declarar `Arbol a b` como instancia de la clase `Eq` de modo que dos árboles sean iguales si coinciden sus *conjuntos* de nodos internos y de hojas.

3. **(1,5 puntos)**

   (i) Definir en Haskell la siguiente función:
   `subs:: [a] -> [[a]]`
   `subs xs` $=_{def}$ lista de todos los subconjuntos de `xs` *(da igual el orden en el que salgan)*

   Ejemplo: subs [1,2,3] = [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]

   (ii) Definir como función booleana, indicando su tipo, la siguiente propiedad:
   `suman n xs` $\leftrightarrow_{def}$ algunos de los elementos de la lista de enteros `xs` suman `n`

   Ejemplo: `suman 5 [1,3,-1,2]` es `True` pero `suman 7 [1,3,-1,2]` es `False`

4. **(1 punto)**
   Dado el programa lógico

   ```
   p(X,c).                          q(c).
   p(g(X),g(Y)) :- p(X,Y).          q(g(g(X))) :- q(X).
   ```

   construir el árbol de resolución del objetivo `q(Y),p(g(g(c)),Y)`.

5. **(1 punto)**
   Programar en Prolog los siguientes predicados:

   (a) `impar(Xs)` $\Leftrightarrow$ la lista `Xs` tiene longitud impar.
   (No pueden usarse predicados primitivos).

   (b) `nvars(T,N)` $\Leftrightarrow$ en el término `T` aparecen `N` variables, distintas o no.
   (Se puede suponer el número natural `N` *representado mediante las constructoras* `cero/0` *y* `s/1`*, o bien mediante números primitivos del sistema, como cada uno prefiera).*

   Ejemplo: `nvars(f(X,0,g(X,1,f(0,Y,a))),N)` *debe tener éxito dando* `N=s(s(s(cero)))` *(o* `N=3` *si se ha preferido usar números primitivos del sistema)*

Esbozo de solución

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación dé la lista de los cuadrados perfectos entre 100 y 2000

   (b) Razona brevemente cuál es el tipo de la función definida por `f x y = x :  f x (y x)`

Solución

(a) Una posibilidad es: `[x | x <- [100..2000],[y|y<-[1..x],y*y==x] /= []]`

(b) Por ser f una función de dos argumentos, tiene que tener un tipo t_f de la forma $t_1 \to t_2 \to t_3$, donde t_1 es el tipo que debe tener x, t_2 el de y y t_3 es el tipo del resultado $x\ :\ f\ x\ (y\ x)$. Pero esta última expresión es una lista de cabeza x, y por tanto su tipo (t_3) ha de ser $[t_1]$. Así pues, t_f debe ser $t_1 \to t_2 \to [t_1]$.

Veamos ahora qué pasa con t_2, el tipo de y. Como en el resultado aparece la subexpresión $(y\ x)$ como segundo argumento de f, sabemos dos cosas:

- $(y\ x)$ tiene el tipo t_2 (que es el tipo del segundo argumento de f)
- y es una función que se aplica a x, y por tanto su tipo es $t_1 \to t_2$, pues t_1 es el tipo de x y t_2 es el tipo de la aplicación $(y\ x)$.

Como el tipo de y es también t_2, tenemos $t_2 = t_1 \to t_2$, que es imposible. As pues, **la función f está mal tipada**.

---

2. **(2 puntos)**
Supongamos el tipo Haskell

```
data Arbol a = Nodo a [Arbol a] deriving Eq
```

para representar árboles no necesariamente binarios (obsérvese, pues, que una hoja `x` vendrá representada como `Nodo x []`). Se pide:

   (a) Programar las siguientes funciones, indicando sus tipos:
   (i) `binario t` devuelve `True` si y solo si el árbol `t` es binario.
   (ii) `nodos t` devuelve el número de nodos de `t`.

   (b) Definir `Arbol a` como instancia de la clase `Ord`, de acuerdo con el siguiente orden: $t < t'$ si t tiene menos nodos que t' o si, teniendo el mismo número de nodos, se cumple $a < a'$, siendo a, a' las raíces de t y t, respectivamente.

Solución

```
data Arbol a = Nodo a [Arbol a] deriving Eq
binario:: Arbol a -> Bool
binario (Nodo _ [])    = True
binario (Nodo _ [i,d]) = binario i && binario d
binario _              = False

nodos:: Arbol a -> Int
nodos (Nodo _ hijos) = 1 + sum (map nodos hijos)

instance Ord a => Ord (Arbol a) where
 x <= x' = x == x' || x < x'
 x < x'  = n < n'  || (n == n' && raiz x < raiz x')
   where n  = nodos x
         n' = nodos x'
         raiz (Nodo x _) = x
```

3. **(1 punto)** Definir, indicando los tipos, las siguientes funciones, usando al menos para una de ellas `foldr` o `foldl`:

   (i) `repetir xs` $=_{def}$ lista resultado de repetir `x` veces cada elemento `x` de la lista de enteros `xs` (los números ≤ 0 se eliminan)
   Ejemplo: repetir [2,3,0,1,2] = [2,2,3,3,3,1,2,2]

   (ii) `cuenta x xs` $=_{def}$ número de veces que aparece `x` en la lista `xs`
   Ejemplos: cuenta True [True,False,True] = 2
   cuenta 3 [2,4,6] = 0

   Solución

   Usando `fold` en ambas:

```
repetir:: [Int] -> [Int]
repetir = foldr f [] where f x xs = take x (repeat x) ++ xs
                     -- o, p.ej., = [x |i <- [1..x]]  ++ xs

cuenta:: Eq a => a -> [a] -> Int
cuenta x = foldl (\u v -> if v==x then u+1 else u) 0
```

   Usando recursión explícita en ambas:

```
repetir’ [] = []
repetir’ (x:xs) = take x (repeat x) ++ repetir’ xs

cuenta’ x [] = 0
cuenta’ x (x’:xs) = if x == x’ then 1 + cuenta’ x xs
                               else cuenta’ x xs
```

---

4. **(1 punto)**
   Dado el programa lógico

```
p(a,Y,Y).                          q(c(X)) :- q(X).
p(c(X),Y,Z) :- p(X,c(Y),Z).        q(c(a)).                     .
```

   (i) Construye el árbol de resolución del objetivo `p(c(Y),X,c(c(a))),q(Y).`

   (ii) Indica cómo cambia el árbol si la primera cláusula de `q` se cambia por `q(c(X)) :- !,q(X).`

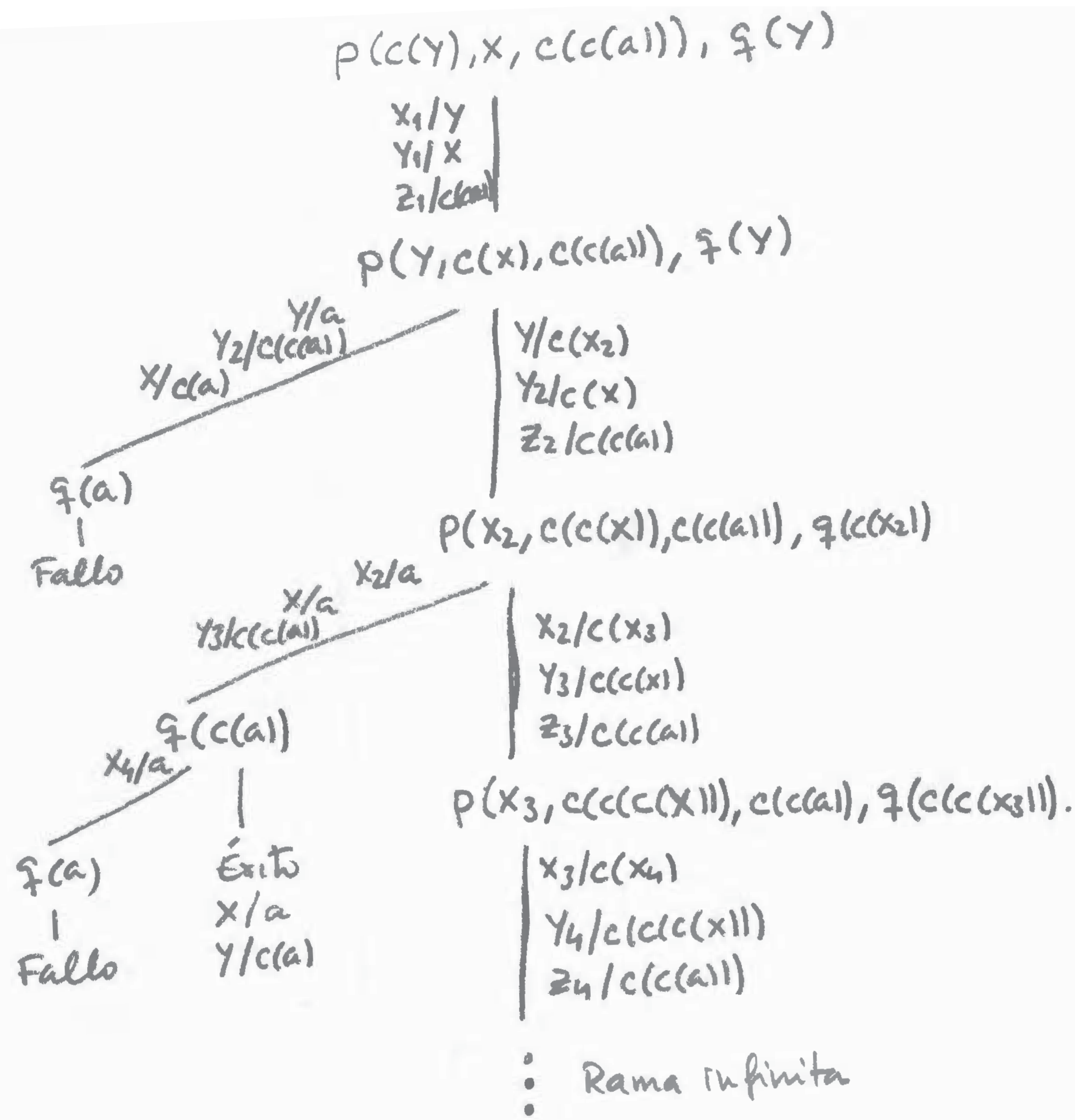
   (ii) Indica cómo cambia el árbol si la primera cláusula de `q` se cambia por `q(c(X)) :- q(X),!.`

   Solución

   (i) *Ver gráfico manuscrito en la hoja siguiente*

   (ii) Se poda la única rama de éxito que hay en el árbol, lo demás queda igual

   (iii) El árbol queda igual que el original


p(c(Y), X, c(c(a))), q(Y)
X1/Y
Y1/X
Z1/c(a)
p(Y, c(X), c(c(a))), q(Y)
Y/a
Y2/c(c(a))
X/c(a)
q(a)
Fallo
Y/c(X2)
Y2/c(X)
Z2/c(c(a))
p(X2, c(c(X)), c(c(a))), q(c(X2))
X2/a
X/a
Y3/c(c(a))
q(c(a))
X4/a
q(a)
Fallo
Éxito
X/a
Y/c(a)
X2/c(X3)
Y3/c(c(X))
Z3/c(c(a))
p(X3, c(c(c(X))), c(c(a)), q(c(c(X3))).
X3/c(X4)
Y4/c(c(c(X)))
Z4/c(c(a))
Rama infinita

5. **(1 punto)**
Programa en Prolog los siguientes predicados:

(a) `incluida(Xs,Ys)` $\Leftrightarrow$ la lista `Xs` está incluida en la lista `Ys`, en el sentido de que todos los elementos de `Xs` lo son también de `Ys`.
(Hay que definir todos los predicados que se usen).

(b) `vecesf(T,N)` $\Leftrightarrow$ en el término `T` aparece `N` veces el símbolo `f` como funtor (constructora de datos) de alguno de los subtérminos de `T`.
(Se puede suponer el número natural `N` *representado mediante las constructoras* `cero/0` *y* `s/1`*, o bien mediante números primitivos del sistema, como cada uno prefiera).*

Ejemplo: `vecesf(f(X,0,g(0,1,f(0,Y,f))),N)` *debe tener éxito dando* `N=s(s(s(cero)))` *(o* `N=3` *si se ha preferido usar números primitivos del sistema)*

Solución

```
incluida([],Ys).
incluida([X|Xs],Ys) :-
  miembro(X,Ys),
  incluida(Xs,Ys).

miembro(X,[X|_]).
miembro(X,[_|Xs]) :-
  miembro(X,Xs).
```

```
vecesf(X,0) :- var(X),!.
vecesf(T,N) :-
  T =..[f|Ts],!,
  vecesf_l(Ts,M),
  N is M+1.
vecesf(T,N) :-
  T =..[_|Ts],
  vecesf_l(Ts,N).

vecesf_l([],0).
vecesf_l([X|Xs],N) :-
  vecesf(X,N1),
  vecesf_l(Xs,N2),
  N is N1+N2.
```

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación produzca el valor $[(1,1),(2,4),(3,9),(4,16),\ldots,(20,400)]$

   (b) Razona brevemente cuál es el tipo de la función definida como `f x (y:ys) = (x y):f x ys`

   Solución

   (a) Una posibilidad es: `[(i,i^2) | i <- [1..20]]`

   (b) Por ser f una función de dos argumentos, tiene que tener un tipo t_f de la forma $t_1 \to t_2 \to t_3$, donde t_1 es el tipo que debe tener x, t_2 el de $y : ys$ y t_3 es el tipo del resultado $(x\ y) :\ f\ x\ ys$. Como $y : ys$ es una lista, su tipo t_2 debe ser $t_2 = [a]$, y además y tiene que tener tipo a e ys tipo $[a]$. Por otra parte, formando parte del resultado aparece $(x\ y)$, lo que indica que x es una función y por tanto su tipo t_1 debe ser $t_1 = a \to b$ (ya que el argumento de x es y, cuyo tipo es a). El tipo de $(x\ y)$ es entonces b, y como $(x\ y)$ es la cabeza del resultado $(x\ y) :\ f\ x\ ys$, el tipo del resultado (que habíamos llamado t_3) debe ser $t_3 = [b]$. Así pues, el tipo que nos resulta para f es $t_f = (a \to b) \to [a] \to [b]$.

   El resto del resultado, $f\ x\ ys$, ya no aporta información nueva sobre el tipo de f, pero su tipo resulta coherente con todo lo anterior (de lo contrario, la función f estaría mal tipada), pues f aparece aplicada a x (de tipo $a \to b$, como se espera) y a ys (de tipo $[a]$, como se espera), produciendo un resultado de tipo $[b]$, como se espera del resto de una lista cuya cabeza es $(x\ y)$ que tiene tipo b.

---

2. **(2 puntos)**

   (a) Definir en Haskell un tipo de datos `Fraccion` para representar fracciones de enteros.

   (b) Definir la siguientes operaciones:

   - Elevar una fracción a un exponente entero.
   - Calcular la media aritmética de una lista de fracciones

   (c) Definir `Fraccion` como instancia de la clases `Eq` y `Ord`, de acuerdo con el orden natural de fracciones.

   Solución

```
data Frac = F Int Int deriving Show

eleva:: Frac -> Int -> Frac
eleva (F x y) n
  | n >= 0    = F (x^n) (y^n)
  | otherwise = F (y^(-n)) (x^(-n))

suma:: Frac -> Frac -> Frac
suma (F x y) (F x' y') = F (x*y'+x'*y) (y*y')

divide:: Frac -> Frac -> Frac
divide (F x y) (F x' y') = F (x*y') (x'*y)

media:: [Frac] -> Frac
media fs = divide (foldl suma (F 0 1) fs) (F (length fs) 1)

instance Eq Frac where
 F x y == F x' y' = x*y' == y*x'

instance Ord Frac where
 F x y <= F x' y'
  | y*y' > 0  = x*y' <= y*x' -- Los dos numeradores son del mismo signo
  | otherwise = x*y' >= y*x'
```

3. **(1 punto)** Programar, indicando los tipos, las siguientes funciones, usando listas intensionales al menos para una de ellas.

   (i) `mappos f [x0,x1,...,xn]` $=_{def}$ `[f 0 x0, f 1 x1,...,f n xn]`
   Ejemplo: mappos (+) [2,3,0,1,2] = [2,4,2,4,6]

   (ii) `g n` $=_{def}$ [[0,1,2,...,n],[1,2,...,n],[2,...,n],...,[n]]
   Ejemplo: g 3 = [[0,1,2,3],[1,2,3],[2,3],[3]]

   Solución

```
mappos:: (Int -> a  -> b) -> [a] -> [b]
mappos f xs = zipWith f [0..length xs - 1] xs

g:: Int -> [[Int]]
g n = [[i..n]|i <- [0..n]]
```

---

4. **(1 punto)**
   Dado el programa lógico

```
p(a,b).     p(a,c).    p(c,d).
q(X,Y) :- p(X,Y).
q(X,Y) :- p(X,Z),q(Z,Y).
```

   (i) Construye el árbol de resolución del objetivo `q(Y,d)`.

   (ii) Indica cómo cambia el árbol si la segunda cláusula de `q` se cambia por `q(X,Y) :- p(X,Z),!,q(Z,Y).`

5. **(1 punto)**
   Programa en Prolog los siguientes predicados:

   (a) `intersecan(Xs,Ys)` ⇔ las listas `Xs` e `Ys` tienen al menos un elemento común.
   (Hay que definir todos los predicados que se usen).

   (b) `tamanyo(T,N)` ⇔ `N` es el número de símbolos (variables, constantes, constructoras) que aparecen en el término `T`.
   (Se puede suponer el número natural `N` *representado mediante números primitivos del sistema, o bien mediante las constructoras* `cero/0` *y* `s/1`*, como cada uno prefiera).*

   Ejemplo: `tamanyo(f(X,g(a,b,f(a,X))),N)` *debe tener éxito dando* `N=8` *(o* `N=s(s(s(s(s(s(s(s(cero))))))))` *si se ha preferido usar constructoras).*

   Solución

```
intersecan(Xs,Ys) :-
  member(X,Xs),
  member(X,Ys).
member(X,[X|_]).
member(X,[_|Xs]) :-
  member(X,Xs).

tamanyo(X,1) :-
  var(X).
tamanyo(X,N) :-
  nonvar(X),
  X=..[F|Xs],
  tamanyo_l(Xs,M),
  N is M+1.

tamanyo_l([],0).
tamanyo_l([X|Xs],N) :-
  tamanyo(X,N1),
  tamanyo_l(Xs,N2),
  N is N1+N2.
```

---

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación produzca la lista infinita `[0,1,-1,2,-2,3,-3,...`
   Nota: si se definen funciones auxiliares o definiciones locales, se quitan la mitad de los puntos.

   (b) Razona brevemente cuál es el tipo de la función definida por las ecuaciones
   ```
   f [] x y      = y
   f (a:as) x y  = x (f as x y) a
   ```

2. **(2 puntos)**
   Se considera el siguiente repertorio de figuras geométricas en el plano:

   - Rectángulos (con base horizontal), determinados por sus vértices inferior izquierdo y superior derecho
   - Círculos, determinados por su centro y su radio

   Se pide:

   (i) Definir un tipo de datos `Figura` para representar esa familia de figuras geométricas.

   (ii) Definir, declarando su tipo, una función que determine si dos figuras se intersecan.

   (iii) Declarar `Figura` como instancia de la clase `Ord`, de modo que las figuras se comparen simplemente por su área.

   (iv) ¿Qué orden entre figuras se tendría si, en lugar de la declaración del apartado anterior, se hubiera utilizado `deriving Ord` al definir el tipo `Figura`?

3. **(1 punto)** Definir, indicando los tipos, las siguientes funciones:

   (i) `apariciones xs` = lista resultado de reemplazar en `xs` cada elemento `x` por el número de veces que aparece `x` en `xs`
   Ejemplo: apariciones [1,2,1,3,2,1,4,2,2] = [3,4,3,1,4,3,1,4,4]

   (ii) `divHasta n` = lista de parejas $(\mathtt{i},\mathtt{d}_i)$, donde `i` va desde `1` hasta `n` y $\mathtt{d}_i$ es el número de divisores de `i`.
   Ejemplo: divHasta 6 = [(1,1),(2,2),(3,2),(4,3),(5,2),(6,4)]

4. **(1 punto)**
   Dado el programa lógico

   ```
   p(X,c(Y,Z)) :- p(X,Z).          q(a).
   p(X,c(X,Y)).                    q(b).
   ```

   (i) Construye el árbol de resolución del objetivo `p(Y,c(d,c(X,c(b,d)))),q(Y).`

   (ii) Indica cómo cambia el árbol si la primera cláusula de `q` se cambia por `q(a) :- !.`

   (ii) Indica cómo cambia el árbol si la primera cláusula de `p` se cambia por `p(X,c(Y,Z)) :- p(X,Z),!.`

5. **(1 punto)**
   Programa en Prolog los siguientes predicados:

   (a) `separa(Xs,Ys,Us,Vs)` ⇔ `Us` es la lista de los elementos de la lista `Xs` que están también en la lista `Ys`, y `Vs` es la lista de los elementos de `Xs` que no están en `Ys`.
   (Hay que definir todos los predicados que se usen).

   (b) `suma(T,N)` ⇔ N es la suma de los números que aparecen como subtérminos en el término `T`.

   Ejemplo: `suma(f(X,2,g(a,[3],f(2,Y,true))),N)` *debe tener éxito con respuesta* `N=7`

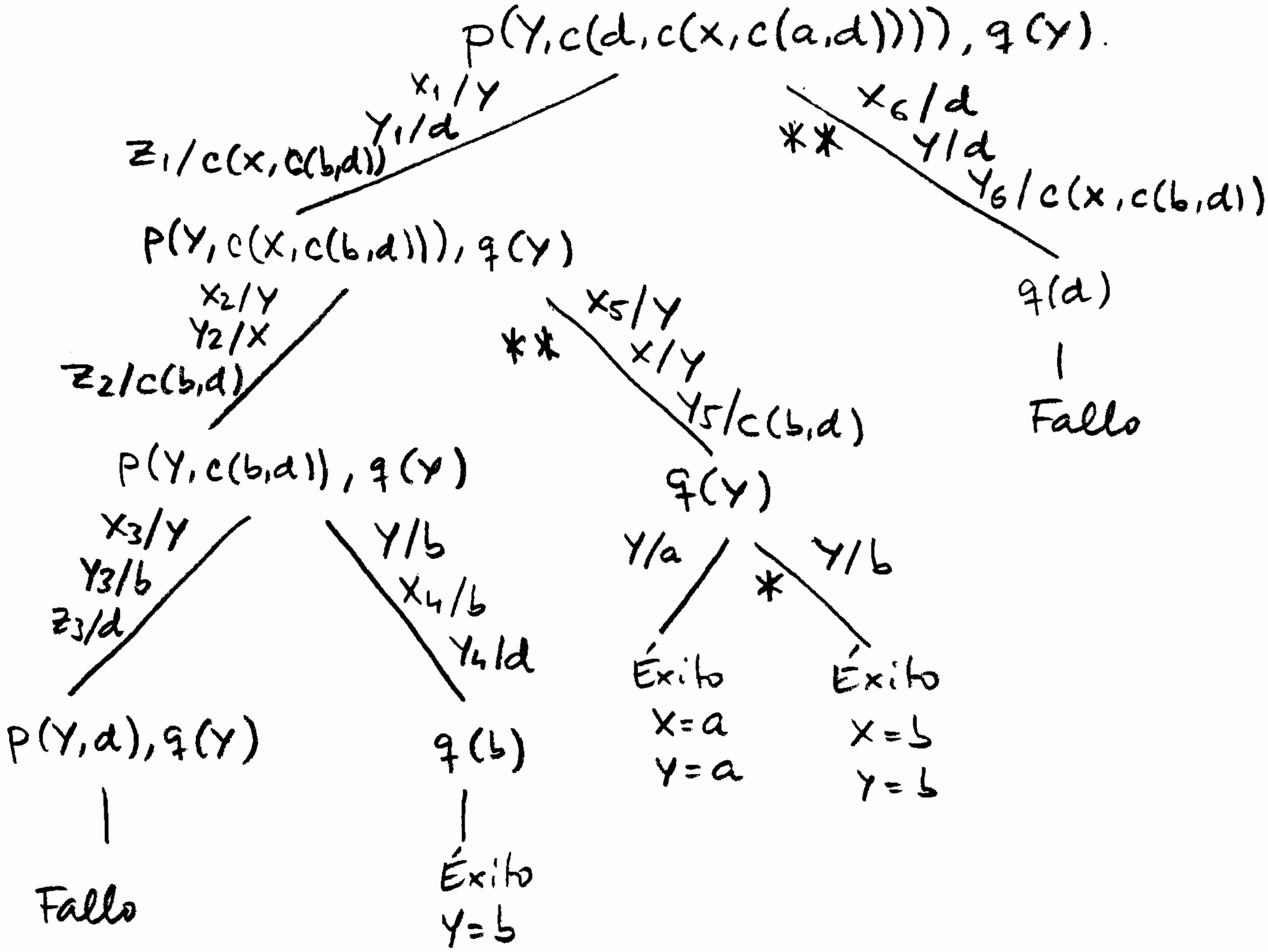
```
%%%%%%%%%%%% Examen Programación Declarativa junio 2015  %%%%%%%%%%%%
%%%%%%%%%%%%            Esbozo de soluciones             %%%%%%%%%%%%

--------------- Ejercicio 1 ------------------

-- Apartado a
l0 = 0:[j|i <- [1..], j <- [-i,i]]
-- Otro par de soluciones; hay muchas más, por supuesto
l1 = iterate (\x -> if x<=0 then -x+1 else -x) 0
l2 = 0:foldr (\x y -> x:(-x):y) [] [1..]
-- Nota: en lugar de [] valdría aquí cualquier otro valor de tipo [Int]
-- pues, al ser [1..] una lista infinita, ese valor no se va a utilizar
-- en el cómputo


{-- Apartado b
f [] x y = y
f (a:as) x y = x (f as x y) a

Llamemos tf, ta, tx, ty, al tipo de f, de a, de x, de y, etc.
f es una función tres argumentos, por tanto su tipo tf será de la forma
   tf = t1 -> t2 -> t3 -> t4.
Por la primera ecuación de f, tenemos que
   t1 = [t] , t2 = tx , t3 = ty , t4 = ty,
O sea, de momento tenemos
   tf = [t] -> tx -> ty -> ty.

De la segunda ecuación obtenemos que (a:as), al ser primer argumento de f,
habrá de tener el tipo [t] y por tanto ta = t, tas = [t].

En el lado derecho de la segunda ecuación vemos que x está aplicado a dos argumentos
y por tanto x es una función. Su primer argumento es (f as x y),
que tiene tipo ty (el tipo devuelto por f) y el segundo a, que tiene tipo t.
Y como lo que devuelve la aplicación de x en esa segunda ecuación
es lo que devuelve f, el tipo devuelto por x será el mismo que devuelve f, o sea, ty.
Así pues tx tiene tipo ty -> t -> ty

Por tanto el tipo de f es
   tf = [t] -> (ty -> t -> ty) -> ty -> ty

Los tipos t y ty ya no tienen ninguna restricción, o sea que pueden ser cualquier
tipo, y por tanto pueden considerarse variables de tipo. Llamándolas a y b, nos queda:
   tf = [a] -> (b -> a -> b) -> b -> b

Nota: por si acaso ayuda a entender mejor, obsérvese que f es como foldr,
pero con los argumentos cambiados de orden.
--}

--------------- Ejercicio 2 ------------------

-- Apartado i
type Punto = (Double,Double)

-- Tipo de datos para figuras
data Figura = R Punto Punto  -- vértice inf. izdo y vértice sup. dcho
            | C Punto Double -- centro y radio
  deriving Show

-- Suponemos en lo que sigue que las figuras están bien formadas,
-- es decir, que en un dato (C centro radio)  se cumple radio>0
-- y que en un dato (R (a,b) (c,d)) se cumple c>a y d>b

-- Apartado ii: intersección de figuras
-- Alguna función útil
```

```
-- vertices fig devuelve una lista con los cuatro 'vértices' de fig
-- En el caso de un círculo, llamamos vértices a los puntos más al sur , norte, este y oeste
-- es decir, los de menor y mayor ordenada, menor y mayor abscisa
vertices:: Figura -> [Punto]
vertices (R (a,b) (c,d)) = [(a,b),(a,d),(c,b),(c,d)]
vertices (C (a,b) r)     = [(a,b-r),(a,b+r),(a-r,b),(a+r,b)]

-- distancia entre puntos
distancia:: Punto -> Punto -> Double
distancia (x,y) (x',y') = sqrt ((x'-x)^2+(y'-y)^2)

-- Comprobación de si una figura contiene a un punto
contiene:: Figura -> Punto -> Bool
contiene (R (a,b) (c,d)) (x,y) = a<=x && x<=c && b<=y && y<=d
contiene (C (a,b) r)     (x,y) = distancia (x,y) (a,b) <= r

intersecan:: Figura -> Figura -> Bool
-- Dos círculos intersecan si la distancia entre sus centros es menor que la suma de los radios
-- Para el resto de casos (dos rectángulos o rectángulo y círculo), dos figuras intersecan
-- si algún vértice de alguna de ellas pertenece a la otra.

intersecan (C c r) (C c' r') = distancia c c' <= r+r'
intersecan fig fig'          = any (contiene fig) (vertices fig')
                               ||
                               any (contiene fig') (vertices fig)

-- Apartado iii: comparación de figuras mediante áreas
area :: Figura -> Double
area (R (a,b) (c,d)) = (c-a)*(d-b)
area (C c r) = pi*r^2

-- Para que Figura sea instancia de Ord debe serlo de Eq
-- No nos dicen nada sobre la igualdad de figuras, pero parece coherente que
-- la definamos también en términos de áreas, y así está hecho a continuación.
-- Alternativamente, podríamos haber usado 'deriving Eq'.
-- Omito la discusión de las diferencias a que daría lugar esto.

instance Eq Figura where
  fig == fig' = area fig == area fig'

instance Ord Figura where
  x <= y = area x <= area y

-- Apartado iv: consultar transparencias de clase sobre orden inducido por 'deriving Ord'

--------------- Ejercicio 3 ------------------

-- Apartado i
apariciones:: Eq a => [a] -> [Int]
apariciones xs = [veces xs x| x <- xs]
-- O, lo que es lo mismo, apariciones xs = map (veces xs) xs
-- Donde veces xs s = núm veces que aparece x en xs
veces:: Eq a => [a] -> a -> Int
veces xs x = length [y|y<-xs,y==x]

-- Apartado ii
divHasta:: Int -> [(Int,Int)]
divHasta n = [(i,ndivisores i) | i <- [1..n]]
-- ndivisores n = número de divisores de n
ndivisores:: Int -> Int
ndivisores n = length [i|i <- [1..n],mod n i == 0]

%%%%%%%%%%%%%%%% Ejercicio 4 %%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%% Ver figura en fichero aparte %%%%%%%%%


%%%%%%%%%%%%%%%% Ejercicio 5 %%%%%%%%%%%%%%%%%%%
% Apartado a
separa([],_,[],[]).
separa([X|Xs],Ys,[X|Us],Vs) :-
  member(X,Ys),!,
  separa(Xs,Ys,Us,Vs).
separa([X|Xs],Ys,Us,[X|Vs]) :-
  separa(Xs,Ys,Us,Vs).

% Apartado b
suma(X,0) :-
  var(X),!.
suma(X,X) :-
  number(X),!.
suma(X,N) :-
  X =..[_|Xs],
  suma_l(Xs,N).
suma_l([],0).
suma_l([X|Xs],N) :-
  suma(X,N1),
  suma_l(Xs,N2),
  N is N1+N2.
```

%%%%%%%%%%%%%%% Ejercicio 4 %%%%%%%%%%%%%%%%%

```
% Apartado i
p(X,c(Y,Z)) :- p(X,Z).            q(a).
p(X,c(X,Y)).                      q(b).

% Objetivo: p(Y,c(d,c(X,c(a,d)))),q(Y).
```




```
% Apartado ii
q(a) :- !.                        q(b).
```

Quedan podadas las ramas marcadas con * y todas sus descendientes

```
% Apartado iii
p(X,c(Y,Z)) :- p(X,Z),!.          p(X,c(X,Y)).
```

Quedan podadas las ramas marcadas con ** y todas sus descendientes

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación produzca la lista infinita `[(0,0),(1,1),(2,3),(3,7),(4,15),(5,31),...`
   Nota: si se definen funciones auxiliares o definiciones locales, se quitan la mitad de los puntos.

   (b) Razona brevemente cuál es el tipo de la función definida por las ecuaciones
   ```
   f x y []      = y
   f x y (a:as)  = f x (x y a) as
   ```

2. **(2 puntos)**

   (a) Define un tipo de datos Haskell **EBool** para representar como datos expresiones booleanas, que pueden ser una de estas cosas: constante para el valor **cierto**, negación de otra expresión booleana, conjunción de otras dos expresiones booleanas.

   (b) Programa una función para evaluar una expresión booleana a su valor booleano **True** o **False**.

   (c) Se considera que dos expresiones son iguales si son sintácticamente iguales excepto posiblemente por el orden de las dos componentes en las conjunciones. Declara el tipo de las expresiones booleanas como instancia de la clase **Eq** teniendo en cuenta esta noción de igualdad.

   (d) Modifica el tipo de las expresiones para que puedan incluir variables y programa una función que reconozca si una expresión es una tautología o no.

3. **(1 punto)** Definir, indicando los tipos, las siguientes funciones:

   (a) `partes xs` = lista de las partes de `xs`
   Ejemplo: `partes [1,2,3] = [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]`, aunque el orden en el que salgan las listas o los elementos dentro de ellas es irrelevante.

   (b) Dada una lista `xs` de enteros no negativos, llamamos un *segmento positivo* a una secuencia de elementos consecutivos de `xs` que sean positivos. Programar una función **sumSeg** que devuelva la lista con las sumas de los segmentos positivos de xs.
   Ejemplo: `sumSeg [1,3,0,2,5,1,0,0,4] = [4,8,4]`

4. **(1 punto)**
   Dado el programa lógico

   ```
   p(X,c(Y,Z)) :- p(X,Z).          q(a).
   p(X,c(X,Y)).                    q(b).
   ```

   (a) Construye el árbol de resolución del objetivo `q(Y),p(Y,c(d,c(X,c(b,d))))`.

   (b) Indica cómo cambia el árbol si la primera cláusula de `p` se cambia por `p(X,c(Y,Z)) :- p(X,Z),!.`

5. **(1 punto)**
   Programa en Prolog los siguientes predicados (y todos los auxiliares que se necesiten):

   (a) `separa(Xs,X,N,Ys)` ⇔ `N` es el número de veces que aparece `X` como elemento de la lista `Xs`, e `Ys` es la lista de los restantes elementos de `Xs`, o sea, los que no son iguales a `X`.

   Ejemplo: `separa([a,b,a,c(a),d],a,N,Ys)` debe tener éxito con respuesta `N=2 , Ys=[b,c(a),d]`.

   (b) `cambia(X,Y,T,S)` ⇔ S es el término que resulta de reemplazar por Y cada aparición de X en T.

   Ejemplo: `cambia(c(X),a,f(c(c(X),Y,c(X),c(X,b)),S)` debe tener éxito con respuesta `S=f(c(a,Y,a,c(X,b))`.

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación produzca la lista infinita `[(0,1),(1,2),(2,4),(3,8),(4,16),...`
   Nota: puedes usar funciones del preludio de Haskell o lambda expresiones, pero no otras funciones auxiliares.

   (b) Razona brevemente cuál es el tipo de la función definida por la ecuación
   ```
   f x y  = x (y x)
   ```

2. **(2 puntos)**
   Considera el siguiente tipo de datos para representar conjuntos finitos de elementos de un tipo cualquiera:
   ```
   data Conjunto a = Con Int [a]
   ```
   donde en un dato `Con n xs` que represente a un conjunto C, el argumento `n` representa el cardinal de C y `xs` es la lista de sus `n` elementos.

   (i) Define una función `toC` que convierta listas en conjuntos.
   Ejemplo: `toC [2,1,2,4,2,1]` *debe devolver* `Con 3 [1,2,4]`, *donde en el resultado es importante que no haya repeticiones de elementos, pero su orden es indiferente.*

   (ii) Define la intersección de conjuntos.

   (iii) Define la función `mapset f c`, que calcula la imagen por `f` de un conjunto `c`, es decir, el conjunto resultado de aplicar la función `f` a cada elemento de `c`.

   (iv) Declara `Conjunto a` como instancia de la clase `Eq`, de modo que dos valores de un tipo `Conjunto` τ sean iguales si representan el mismo conjunto.

   Notas: indica los tipos de todas las funciones que definas; puedes usar funciones del preludio de Haskell.

3. **(1 punto)** Define la función `reverse` (que computa la inversa de una lista) en términos de `foldr` y de `foldl`. Indica cuál de las dos versiones tiene menor complejidad.

4. **(1 punto)** Dado el programa lógico

   ```
   p(s(X),Y,s(Z)) :- p(X,Y,Z).          q(a).
   p(a,Y,Y).                            q(s(s(X))) :- q(X).
   ```

   (i) Construye el árbol de resolución del objetivo `p(Y,X,s(s(a))) , q(Y)`.

   (ii) Indica cómo cambia el árbol si la primera cláusula de `p` se cambia por `p(s(X),Y,s(Z)) :- !, p(X,Y,Z)`.

   (iii) Indica cómo cambia el árbol si la primera cláusula de `q` se cambia por `q(a) :- !`.
   (Este cambio es independiente del del apartado ii)

5. **(1 punto)**
   Programa en Prolog los siguientes predicados:

   (a) `veces(Xs,Ns)` ⇔ `Ns` es la lista cuyo elemento i-ésimo es el número de veces que aparece en `Xs` el elemento i-ésimo de `Xs`.
   Ejemplo: `veces([2,1,2,4,2,1],N)` *deber tener éxito devolviendo* `N = [3,2,3,1,3,2]`.

   (b) `nvars(T,N)` ⇔ N es el número de variables distintas que aparecen en el término `T`.

   Ejemplo: `vars(f(Y,2,g(X,[a],f(X,Y,true))),N)` *debe tener éxito con respuesta* `N=2`, *pues en el primer argumento aparecen dos variables distintas,* `X` *e* `Y`.

```
============== PROGRAMACIÓN DECLARATIVA   JUNIO 2016  ESBOZO DE SOLUCIONES =============

------------------------------------------------------------------------------------
Las soluciones propuestas no buscan tanto la eficiencia como la simplicidad y claridad
------------------------------------------------------------------------------------


1. (1 punto)
(a) Escribe una expresión Haskell cuya evaluación produzca la lista infinita [(0,1),(1,2),(2,4),(3,8),(4,16),...
Nota: puedes usar funciones del preludio de Haskell o lambda expresiones, pero no otras funciones auxiliares.
(b) Razona brevemente cuál es el tipo de la función definida por la ecuación
f x y = x (y x)

---------------- Solución ejercicio 1 ----------------
(a) Hay muchas posibles, claro. Una muy simple es [(n,2^n) | n <- [0..]]

(b) Sabemos que f:: tx -> ty -> t, donde x::ts  e  y::ty, y se tendrá (f x y)::t
Como y aparece aplicado a x, ha de ser ty:: tx -> t' y se tendrá (y x):: t'
Como x aparece aplicado a (y x), ha de ser tx:: t' -> t'' y se tendrá  x (y x):: t''
Pero x (y x) es el valor de f x y, cuyo tipo es t, luego t = t''
De modo que f::(t' -> t) -> ((t' -> t) -> t') -> t
O bien, renombrando variables, f::(a -> b) -> ((a -> b) -> a) -> b
---------------- Fin solución ejercicio 1 ----------------

2. (2 puntos)
Considera el siguiente tipo de datos para representar conjuntos finitos de elementos de un tipo cualquiera:
data Conjunto a = Con Int [a]
donde en un dato Con n xs que represente a un conjunto C, el argumento n representa el cardinal de C
y xs es la lista de sus n elementos.
(i) Define una función toC que convierta listas en conjuntos.
Ejemplo: toC [2,1,2,4,2,1] debe devolver Con 3 [1,2,4], donde en el resultado es importante que no haya repeticiones
de elementos, pero su orden es indiferente.
(ii) Define la intersección de conjuntos.
(iii) Define la función mapset f c, que calcula la imagen por f de un conjunto c, es decir, el conjunto resultado
de aplicar la función f a cada elemento de c.
(iv) Declara Conjunto a como instancia de la clase Eq, de modo que dos valores de un tipo Conjunto τ
sean iguales si representan el mismo conjunto.
Notas: indica los tipos de todas las funciones que definas; puedes usar funciones del preludio de Haskell.

---------------- Solución ejercicio 2 ----------------
```

```
data Conjunto a = Con Int [a]

(i) Definimos primero una función para eliminar repeticiones de una lista
noRep:: Eq a => [a] -> [a]
noRep [] = []
noRep (x:xs) = x:noRep [y | y <- xs, y /= x]

toC:: Eq a => [a] -> Conjunto a
toC xs = (length xs', xs')
         where xs' = noRep xs

(ii) inter:: Eq a => Conjunto a -> Conjunto a -> Conjunto a
inter (Con _ xs) (Con _ ys) = toC [x | x <- xs, elem x ys]

(iii) mapset f (Con _ xs) = toC (map f xs)

(iv) instance Eq a => Eq (Conjunto a) where
      (Con n xs) == (Con m ys) =
         (n == m)
         &&
         [x|x <- xs, not (elem x ys)] == []  -- xs es subconjunto de ys
         &&
         [y|y <- ys, not (elem y xs)] == []  -- ys es subconjunto de xs
```

La igualdad de conjuntos está expresada como la igualdad de
cardinales y la inclusión de mutua de los conjuntos.
Es redundante pues bastaría pedir la igualdad de cardinales y una inclusión,
o bien pedir solamente la doble inclusión.
---------------- Fin solución ejercicio 2 ----------------

3. (1 punto) Define la función reverse (que computa la inversa de una lista) en t´ erminos de foldr y de foldl.
Indica cuál de las dos versiones tiene menor complejidad.

---------------- Solución ejercicio 3 ----------------

```
reverse = foldr (\x xs -> xs++[x]) []
reverse' = foldl (\xs x -> x:xs) []
```

La primera versión se corresponde con la versión 'ingenua' de reverse
y tiene coste cuadrático, pues la concatenación xs++[x], que tiene coste
lineal en la longitud de xs, ha de realizarse con listas xs de tamaño

creciente [],[x1],[x1,x2],...,[x1,...,xn].

La segunda versión se corresponde con la versión de reverse con acumulador
y tiene coste lineal.
---------------- Fin solución ejercicio 3 ----------------

4. (1 punto) Dado el programa lógico

```
p(s(X),Y,s(Z)) :- p(X,Y,Z).            q(a).
p(a,Y,Y).                              q(s(s(X))) :- q(X).
```

(i) Construye el árbol de resolución del objetivo p(Y,X,s(s(a))) , q(Y).
(ii) Indica cómo cambia el árbol si la primera cláusula de p se cambia por p(s(X),Y,s(Z)) :- !, p(X,Y,Z).
(iii) Indica cómo cambia el árbol si la primera cláusula de q se cambia por q(a) :- !.
(Este cambio es independiente del del apartado ii)

---------------- Solución ejercicio 4 ----------------
Ver árbol en hoja aparte
----

5. (1 punto)
Programa en Prolog los siguientes predicados:
(a) veces(Xs,Ns) ⇔ Ns es la lista cuyo elemento i-ésimo es el número de veces que aparece en Xs el elemento i-ésimo de Xs.
Ejemplo: veces([2,1,2,4,2,1],N) deber tener éxito devolviendo N = [3,2,3,1,3,2].
(b) nvars(T,N) ⇔ N es el número de variables distintas que aparecen en el término T.
Ejemplo: vars(f(Y,2,g(X,[a],f

---------------- Solución ejercicio 5 ----------------
(a)
Introducimos un predicado auxiliar veces_aux/3 especificado (y definido más abajo) como

```
% veces_aux(Xs,Ys,Ns) <-> Ns es el número de apariciones en Ys de cada elemento de Xs
```

Y con él expresamos veces/2 como sigue:

```
veces(Xs,Ns) :- veces_aux(Xs,Xs,Ns).
```

La definición de veces_aux/3 puede ser:

```
veces_aux([],_,[]).
veces_aux([X|Xs],Ys,[N|Ns]) :-
```

```
  veces_uno(X,Ys,N),
  veces_aux(Xs,Ys,Ns).

% veces_uno(X,Ys,N) <-> N es el número de apariciones en Ys de X
veces_uno(X,[],0).
veces_uno(X,[X|Xs],N) :-
  veces_uno(X,Xs,M),
  N is M+1.
veces_uno(X,[Y|Xs],N) :-
  X \= Y,
  veces_uno(X,Xs,N).
```

(b)
Introducimos un predicado auxiliar vars_aux/4 especificado como

```
% nvars_aux(T,Vars,NewVars,N) <-> N es el número de variables distintas que aparecen en el término T
                          pero no aparecen en la lista de variables Vars.NewVars es una lista que resulta
                          de ampliar Vars con las nuevas variables encontradas en T.
```

Y con él expresamos nvars/2 como sigue:

```
nvars(T,N) :- nvars_aux(T,[],Vs,N).  % Nótese que Vs va a ser la lista de variables distintas de T
```

La idea es que al ir recorriendo recursivamente la estructura de T, el segundo argumento de vars_aux/4 sirve para controlar qué variables ya han aparecido y el tercer argumento para meter las que vayamos encontrando, de modo que el tercer argumento va a pasarse como segundo argumento en la siguiente llamada recursiva.

```
nvars_aux(X,Vs,Vs,0) :-
  var(X),
  id_member(X,Vs), % comprueba que la variable X aparece tal cual en Vs
  !.
nvars_aux(X,Vs,[X|Vs],1) :-
  var(X),
  !.
nvars_aux(T,Vs,Vs1,N) :-
  T =.. [_|As],
  nvars_aux_l(As,Vs,Vs1,N).

nvars_aux_l([],Vs,Vs,0).
nvars_aux_l([A|As],Vs,Vs2,N) :-
```

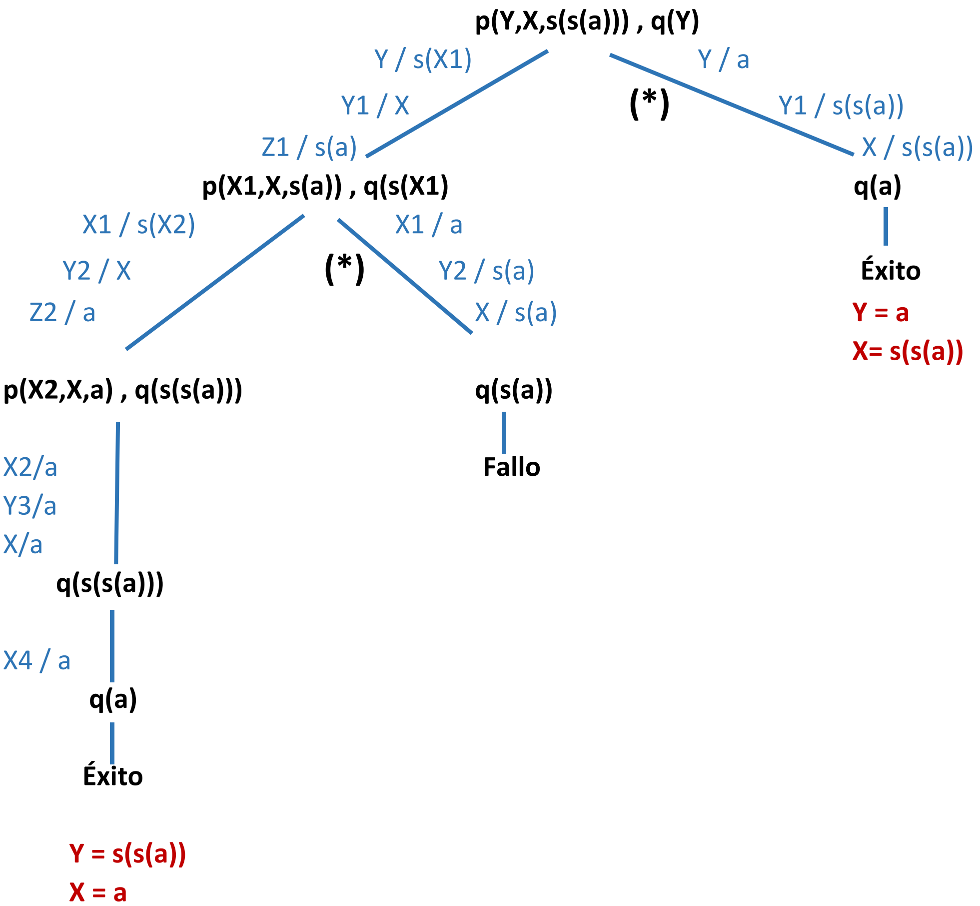
```
  nvars_aux(A,Vs,Vs1,N1),
  nvars_aux_l(As,Vs1,Vs2,N2),
  N is N1+N2.
```

El predicado id_member/2 es similar a member/2, pero con la salvedad importante de que id_member(X,Xs) tiene éxito si hay un elemento de Xs sintácticamente idéntico a X, es decir, que id_member/2 no realiza comprobaciones de igualdad por unificación sino por identidad sintáctica ==. Eso es esencial, porque member(X,Xs) siempre tiene éxito si X es una variable y Xs es una lista no vacía.

```
id_member(X,[Y|_]) :- X == Y,!.
id_member(X,[_|Ys]) :- id_member(X,Ys).
---------------- Fin solución ejercicio 5 ----------------
```

p(s(X),Y,s(Z)) :- p(X,Y,Z). **q(a).**
p(a,Y,Y). **q(s(s(X))) :- q(X).**

(i)



(ii) Se podan las ramas señaladas con **(*)**

(iii) El árbol no cambia

1. **(1 punto)**

   (a) Escribe una expresión Haskell cuya evaluación produzca la lista
   $[1, 2, -2, 4, -4, 8, -8, 16, -16, \ldots, 2^{50}, -2^{50}]$.
   Nota: puedes usar funciones del preludio de Haskell o lambda expresiones, pero no otras funciones auxiliares.

   (b) Razona brevemente cuál es el tipo de la función definida por la ecuación
   `f x y  = y (x y)`

2. **(2 puntos)**
   Considera el siguiente tipo de datos para representar conjuntos finitos de elementos de un tipo cualquiera:
   `data Conjunto a = Con Int [a]`
   donde en un dato `Con n xs` que represente a un conjunto C, el argumento `n` representa el cardinal de C y `xs` es la lista de sus `n` elementos.

   (i) Define la unión de conjuntos.

   (ii) Define la función `zipWithSet f c c'`, an'aloga a la funci'on `zipWith`, pero operando con conjuntos en lugar de con listas.

   (iii) Declara `Conjunto a` como instancia de la clase `Ord`, de modo que el orden `<=` para `Conjunto a` sea la inclusión de conjuntos.

   Notas: indica los tipos de todas las funciones que definas; puedes usar funciones del preludio de Haskell.

3. **(1 punto)** Define la función `concat` (que computa la concatenación de una lista de listas) en términos de `foldr` y de `foldl`. Indica cuál de las dos versiones tiene menor complejidad.

4. **(1 punto)** Dado el programa lógico

```
p(Y,s(Z),s(X)) :- p(Y,Z,X).          q(s(s(X))) :- q(X).
p(Y,Y,a).                            q(a).
```

   (i) Construye el árbol de resolución del objetivo `p(X,s(s(a)),Y) , q(Y)`.

   (ii) Indica cómo cambia el árbol si la primera cláusula de `p` se cambia por `p(Y,s(Z),s(X)) :- !, p(Y,Z,X).`

   (iii) Indica cómo cambia el árbol si la primera cláusula de `q` se cambia por `q(s(s(X))) :- !,q(X).`
   (Este cambio es independiente del del apartado ii)

5. **(1 punto)**
   Programa en Prolog los siguientes predicados:

   (a) `suma(Ns,S)` $\Leftrightarrow$ `S` es la suma de los elementos de `Ns`, que debe ser una lista de longitud impar (podemos suponer sin comprobarlo que `Ns` está formada por números).
   Ejemplo: `suma([2,1,2,4,2],N)` *deber tener éxito devolviendo* `N = 11`*, mientras que* `suma([2,1,2,4],N)` *debe fallar.*

   (b) `lvars(T,Vs)` $\Leftrightarrow$ Vs es la lista de variables que aparecen en el término `T`.

   Ejemplo: `lvars(f(Y,2,g(X,[a],f(X,Y,true))),Vs)` *debe tener éxito con respuesta* `Vs=[Y,X,X,Y]`*, aunque el orden de las variables en* `Vs` *no es importante.*