Ejercicio 1 (1.5 puntos) *Evaluar las siguientes expresiones en Haskell:*

1. `[[x,y] | x <- [1,2], y <- [3,4]]`
2. `[x | (x:y) <- [[7,2,3],[1,3,4],[9,6]], x>sum y]`
3. `[[1..x] | x <- [1..4]]`
4. `foldr (-) 3 [2,3,5]`

Solución:

```
[[x,y] | x <- [1,2], y <- [3,4]]
   ⇝ [[1,3],[1,4],[2,3],[2,4]]
[x | (x:y) <- [[7,2,3],[1,3,4],[9,6]], x>sum y]
   ⇝ [7,9]
[[1..x] | x <- [1..4]]
   ⇝ [[1],[1,2],[1,2,3],[1,2,3,4]]
foldr (-) 3 [2,3,5]
   ⇝ 1
```

Ejercicio 2 (1.5 puntos) *Se considera la función*

```
divideEn :: Int -> [a] -> ([a], [a])
```

tal que `divideEn n l` *es el par formado por la lista de los* `n` *primeros elementos de la lista* `l` *y la lista* `l` *sin los* `n` *primeros elementos. Por ejemplo,*

```
divideEn 3 [5,6,7,8,9,2,3] ⇝ ([5,6,7],[8,9,2,3])
divideEn 4 "sacacorcho"    ⇝ ("saca","corcho")
```

Se pide:

1. *Escribir en Haskell una definición recursiva de la función* `divideEn`
2. *Escribir en Haskell una definición no recursiva de la función* `divideEn`

Solución: La definición recursiva es

```
divideEn :: Int -> [a] -> ([a], [a])
divideEn n xs | n <= 0  = ([],xs)
divideEn _ []           = ([],[])
divideEn n (x:xs)       = (x:xs',xs'')
    where (xs',xs'') = divideEn (n-1) xs
```

La definición no recursiva es

```
divideEn_alt :: Int -> [a] -> ([a], [a])
divideEn_alt n xs  = (take n xs, drop n xs)
```

Ejercicio 3 (1.5 puntos) *El problema de las N reinas consiste en colocar N reinas en un tablero rectangular de dimensiones $N \times N$ de forma que no se encuentren más de una en la misma línea (horizontal, vertical o diagonal). Las soluciones del problema pueden representarse por listas de N números: $[x_1, \ldots, x_N]$ forma que x_i es el número de fila de la reina de la columna $(N+1)-i$. Por ejemplo, la solución del problema de las 4 reinas*

	R		
			R
R			
		R	

se representa por `[3,1,4,2]`

Definir en Haskell la función

```
reinas :: Int -> [[Int]]
```

tal que `reinas n` *es la lista de soluciones del problema de las* `n` *reinas. Por ejemplo,*

```
reinas 4  ⇝ [[3,1,4,2],[2,4,1,3]]
```

Solución:

```
reinas :: Int -> [[Int]]
reinas n = aux n
    where aux 0     = [[]]
          aux (m+1) = [r:rs | rs <- aux m,
                              r <- ([1..n] \\ rs),
                              noAtaca r rs 1]
```

donde la función

```
noAtaca :: Int -> [[Int]] -> Int -> Bool
```

es tal que `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`. Su definición es

```
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

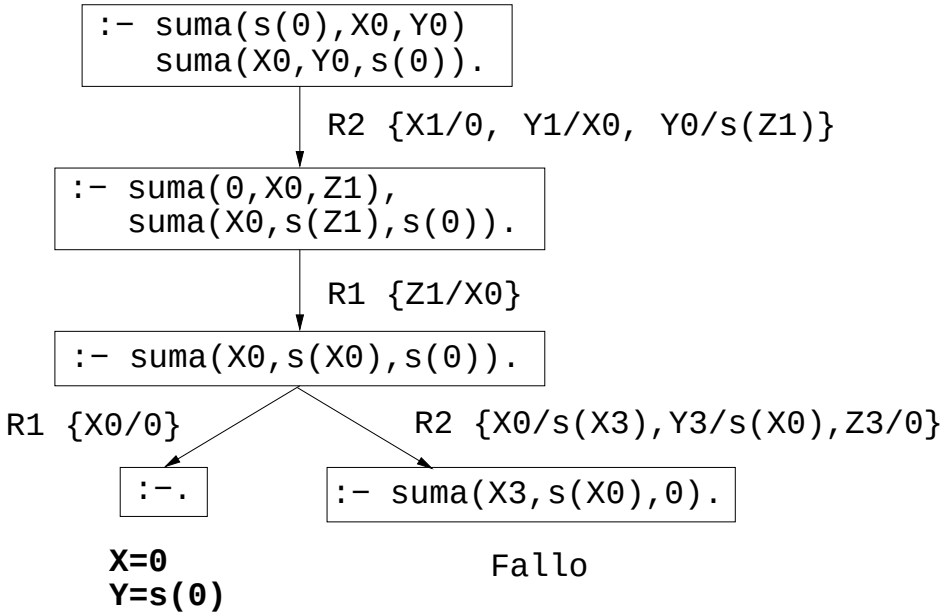
Ejercicio 4 (1 punto) *Se considera el siguiente programa lógico*

```
suma(0,Y,Y).                          % R1
suma(s(X),Y,s(Z)) :- suma(X,Y,Z).     % R2
```

Escribir el árbol de resolución y las respuestas correspondientes al programa y a la pregunta

```
?- suma(s(0),X,Y), suma(X,Y,s(0)).
```

Solución: El árbol de resolución es

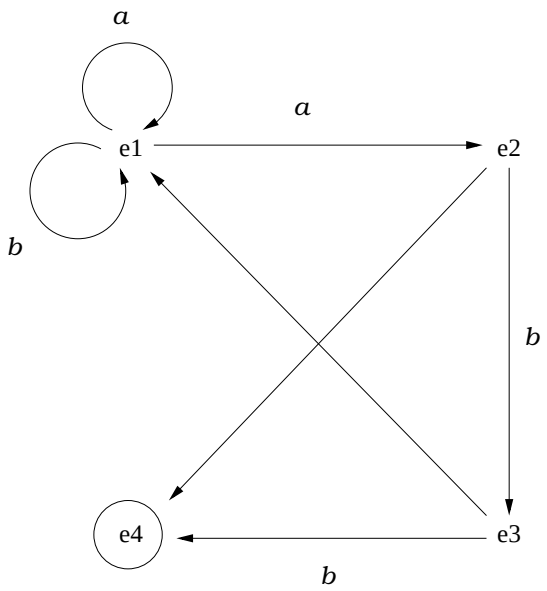


La respuesta obtenida es `X=0` e `Y=s(0)`.

Ejercicio 5 (1.5 puntos) *Un autómata no determinista puede representarse en Prolog mediante tres predicados:*

- `final(E)` *que se verifica si* `E` *es el estado final.*
- `trans(E1,X,E2)` *que se verifica si se puede pasar del estado* `E1` *al estado* `E2` *usando la letra* `X`*.*
- `nulo(E1,E2)` *que se verifica si se puede pasar del estado* `E1` *al estado* `E2` *mediante un movimiento nulo.*

Por ejemplo, el autómata



(con estado final `e3`*) puede representarse mediante los siguientes predicados*

```
final(e3).

trans(e1,a,e1).
trans(e1,a,e2).
trans(e1,b,e1).
trans(e2,b,e3).
trans(e3,b,e4).

nulo(e2,e4).
nulo(e3,e1).
```

Definir en Prolog la relación `acepta(E,L)` *que se verifica si el autómata, a partir del estado* `E` *acepta la lista* `L`*. Por ejemplo,*

```
?- acepta(e1,[a,a,a,b])
Yes
?- acepta(e2,[a,a,a,b])
No
?- acepta(E,[a,b]).
E=e1 ;
E=e3 ;
No
?- acepta(e1,[X,Y,Z]).
X = a   Y = a   Z = b ;
X = b   Y = a   Z = b ;
No
```

Nota: *La definición de* `acepta` *se debe de basar exclusivamente en las relaciones* `final`, `trans` *y* `nulo`.

Solución:

```
acepta(E,[]) :-
   final(E).
acepta(E,[X|L]) :-
   trans(E,X,E1),
   acepta(E1,L).
acepta(E,L) :-
   nulo(E,E1),
   acepta(E1,L).
```

Ejercicio 6 (1.5 puntos) *Un número es capicúa si se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo, 212 y 3553 son capicúas. Definir en Haskell la función,*

```
mayorCapicuaProducto :: Integer -> Integer
```

tal que `(mayorCapicuaProducto n)` *es el mayor número capicúa que es el producto de dos números de* `n` *cifras cada uno. Por ejemplo,*

```
mayorCapicuaProducto 2  ⇝  9009
```

Calcular, usando la función `mayorCapicuaProducto`*, el mayor número capicúa que es producto de dos números de tres cifras.*

Solución: La definición es

```
mayorCapicuaProducto n =
    maximum [x*y | x <- [a..b], y <- [x..b], esCapicua (x*y)]
    where a = 10^(n-1)
          b = 10^n-1
```

donde `(esCapicua x)` se verifica si `x` es capicúa. Cuya definición es

```
esCapicua :: Integer -> Bool
esCapicua x =
    y == reverse y
    where y = show x
```

El mayor número capicúa que es producto de dos números de tres cifras se calcula por

```
Main> mayorCapicuaProducto 3
906609
```

Ejercicio 7 (1.5 puntos) *Definir en Prolog la relación* `mayorCapicuaProducto(+N,-X)` *que se verifique si* `X` *es el mayor número capicúa que es el producto de dos números de* `N` *cifras cada uno. Por ejemplo,*

```
?- mayorCapicuaProducto(2,N).
N = 9009
```

Calcular, usando la relación `mayorCapicuaProducto`*, el mayor número capicúa que es producto de dos números de tres cifras.*

Ayuda: *La potencias enteras se pueden calcular usando* `ceiling` *y* `^`*. Por ejemplo,*

```
?- X is ceiling(2^(4-1)).
X = 8
```

Solución: La definición es

```
mayorCapicuaProducto(N,X) :-
   A is ceiling(10^(N-1)),
   B is ceiling(10^N-1),
   findall(Z,(between(A,B,X),
              between(X,B,Y),
              Z is X*Y,
              esCapicua(Z)),
           L1),
   sort(L1,L2),
   append(_,[X],L2).
```

donde `esCapicua(X)` se verifica si `X` es capicúa y se define por

```
esCapicua(X) :-
   name(X,L),
   reverse(L,L).
```

El mayor número capicúa que es producto de dos números de tres cifras se calcula por

```
Main> mayorCapicuaProducto 3
906609
```