1. De una función `f` solo sabemos que su tipo es `∀a.[a] -> Int`. ¿Cuál de las siguientes situaciones es posible?
   ◯ ⟦`f [1,2]`⟧ = 2 y ⟦`f [True,True]`⟧ = 1
   ⊗ ⟦`f [1,2]`⟧ = 1 y ⟦`f [True,True]`⟧ = 1
   ◯ ⟦`f [1,2]`⟧ = 1 y ⟦`f [True,True]`⟧ = 2

2. De una función `f` solo sabemos que su tipo es `∀a.Eq a => [a] -> Int`. ¿Cuántas de las siguientes situaciones son posibles?
   • ⟦`f [1,2]`⟧ = 2 y ⟦`f [True,True]`⟧ = 1
   • ⟦`f [1,2]`⟧ = 1 y ⟦`f [True,True]`⟧ = 1
   • ⟦`f [1,2]`⟧ = 1 y ⟦`f [True,True]`⟧ = 2
   ⊗ Las tres
   ◯ Solo dos
   ◯ Solo una

3. Considérese la función definida por `f x y = x x y` . El tipo de `f` es:
   ◯ `(a -> a -> a) -> (a -> a) -> a`
   ◯ `(a -> b -> a) -> (b -> a) -> a`
   ⊗ No está bien tipada

4. Considérese la función definida por `f x y = x (y y)` . El tipo de `f` es:
   ◯ `(a -> b) -> (a -> a) -> b`
   ◯ `(a -> a) -> (a -> a) -> a`
   ⊗ No está bien tipada

5. Sea `f` definida por `f g x = x x g`. El tipo de `f` es:

   ◯ `a -> (a -> a -> a) -> a`
   ◯ `∀a,b.a -> (b -> a -> b) -> b`
   ⊗ Está mal tipada

6. ¿Cuál de las siguientes expresiones denota correctamente la acción de leer un carácter y escribirlo dos veces?
   ◯ `let x=getChar in [x,x]`
   ◯
   ```
   do  x <- getChar
       return x
       return x
   ```
   ⊗
   ```
   do  x <- getChar
       putStr [x,x]
   ```

7. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
   `f x y z = if x <= y then z + 1 else z` será:
   ◯ `f :: Num a => a -> a -> a -> a`
   ⊗ `f :: (Ord a, Num b) => a -> a -> b -> b`
   ◯ `f :: (Ord a, Num a) => a -> a -> a -> a`

8. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
   `f x y z = if x <= y z then z + 2 else z` será:
   ◯ `f :: Num a => a -> a -> a -> a`
   ◯ `f :: (Ord a, Num b) => a -> b -> b -> a`
   ⊗ `f :: (Ord a, Num b) => a -> (b -> a) -> b -> b`

9. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
   `f x y z = if x then y <= z else x` será:

○ `f :: (Ord a, Bool a) => a -> a -> a -> a`
⊗ `f :: Ord a => Bool -> a -> a -> Bool`
○ Esa definición dará un error de tipos

10. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y z = if x <= y then z+1 else x` será:
    ⊗ `f :: (Num a,Ord a) => a -> a -> a -> a`
    ○ `f :: (Ord a, Num b) => a -> a -> b -> b`
    ○ `f :: (Ord a, Num b) => a -> a -> b -> a`

11. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y z = if x <= y then z else not x` será:
    ○ `f :: (Ord a, Bool a) => a -> a -> a -> a`
    ○ `f :: Ord a => Bool -> a -> a -> Bool`
    ⊗ `Bool -> Bool -> Bool -> Bool`

12. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y z = z (x <= y+1)` será:
    ⊗ `f :: (Num a, Ord a) => a -> a -> (Bool -> b) -> b`
    ○ `f :: (Ord a, Num b, Ord b) => a -> b -> (Bool -> c) -> c`
    ○ Dará un error de tipos

13. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y z = if x <= y then z + x else z` será:
    ○ `f :: Num a => a -> a -> a -> a`
    ○ `f :: (Ord a, Num b) => a -> a -> b -> b`
    ⊗ `f :: (Ord a, Num a) => a -> a -> a -> a`

14. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y = if x <= 0 then y + 1 else y` será:
    ○ `f :: Num a => a -> a -> a`
    ⊗ `f :: (Num a, Ord a, Num b) => a -> b -> b`
    ○ `f :: (Ord a, Num a) => a -> a -> a`

15. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y = if x == y+1 then y else y+1` será:
    ○ `f :: Eq (Num a) => a -> a -> Num a`
    ⊗ `f :: (Eq a, Num a) => a -> a -> a`
    ○ `f :: (Eq a, Num b) => a -> b -> b`

16. El tipo que inferirá Haskell, teniendo en cuenta clases de tipos, para una función `f` definida por
    `f x y z = if not x then z <= y else x` será:
    ○ `f :: Ord Bool => Bool -> Bool -> Bool -> Bool`
    ○ `f :: Bool -> Bool -> Bool -> Bool`
    ⊗ `f :: Ord a => Bool -> a -> a -> Bool`

17. ¿Cuántas de las siguientes definiciones de tipos (independientes unas de otras) son correctas?

```
data Tip = A  | C Int Tip  | (Int,Int,Tip)
data Tap = A  | C Int Tap  | D Int Int Tap
data Top = A  | C a Top    | D a b Top
```

◯ Las tres
◯ Ninguna de las tres
⊗ Una de las tres

18. ¿Cuántas de las siguientes definiciones de tipos (independientes unas de otras) son correctas?

```
data Tip      = A   | C Int Tip        | C (Int,Int,Tip)
data Tap      = A   | Int | B Int Tap
data Top a b  = A   | C a a            | D a b
```

Nota: esta pregunta fue anulada en su día en un examen, porque es ambigua, o más exactamente, hay una discrepancia entre la respuesta correcta en un sentido estricto, con una interpretación puramente sintáctica de la noción de 'definición correcta de tipo', y la respuesta ante una la lectura más semántica e 'interesante' de la pregunta. Me explico:

- En un sentido técnico estricto, puramente sintáctico, solo la definición de `Tip` es incorrecta (la constructora `C` se repite dos veces); pero `Tap` es correcta sintácticamente, aunque es importante entender que en `Tap` el indentificador `Int` no se refiere al tipo `Int` de los enteros, sino que es una nueva constructora de datos de aridad cero que no tiene nada que ver con el tipo `Int`.
- Sin embargo, si uno interpretase la definición de `Tap` como un intento de que una de las alternativas para ser de tipo `Tap` fuese ser de tipo `Int` (es decir, que los valores del tipo `Int` fuesen también valores del tipo `Tap`), entonces la definición de `Tap` es incorrecta, pues no existe tal noción de subtipado en Haskell.

⊗ Una de las tres *(respuesta correcta en la interpretanción puramente sintáctica)*
⊗ Dos de las tres *(respuesta correcta en la otra interpretanción)*
◯ Las tres

19. ¿Cuántas de las siguientes definiciones de tipos (independientes unas de otras) son correctas?

```
data Tip    = A   | C Int Tip   | C (Int,Int,Tip)
data Tap    = A   | C Int Tap   | D (Int,Int,Tap)
data Top a  = A   | C a         | D a a
```

◯ Una de las tres
⊗ Dos de las tres
◯ Las tres

20. En lo que sigue, $\leq$ indica el orden estándar (el obtenido por `deriving Ord`) para tipos con constructoras de datos. Considérense las afirmaciones siguientes: (i) $[\,] \leq [True] \leq [False, True]$

21. (ii) $[\,] \leq [False, True] \leq [True, False]$. Entonces, teniendo en cuenta que para los booleanos False ≤ True, se tiene:
◯ (i) es cierta pero (ii) no
⊗ (ii) es cierta pero (i) no
◯ Las dos son falsas

22. Considérese la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)`. ¿Cuál de las siguientes afirmaciones es cierta?
⊗ `A <= B && B <= C A A` se evalúa a `True`
◯ `A <= B && B <= C A A` se evalúa a `False`
◯ `C loop loop == C loop loop` se evalúa a `True`, donde `loop` está definido por `loop = loop`

23. Considérese la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)` y la función `mal = head []`. ¿Cuál de las siguientes afirmaciones es cierta?
   ◯ `C mal A <= C mal B` se evalúa a `True`
   ◯ `C A mal == C B mal` se evalúa a `True`
   ⊗ `A <= C mal mal && B <= C mal mal` se evalúa a `True`

24. Considérese la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)` y la función `loop = loop`. ¿Cuál de las siguientes afirmaciones es cierta?
   ⊗ `A <= B && B <= C loop loop` se evalúa a `True`
   ◯ `A <= B && B <= C loop loop` se evalúa a `False`
   ◯ `C loop loop == C loop loop` se evalúa a `True`

25. Considérese la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)` y la función `loop = loop`. ¿Cuál de las siguientes afirmaciones es **falsa**?
   ⊗ `loop <= B && B <= C loop loop` se evalúa a `True`
   ◯ `A <= C loop loop && B <= C loop loop` se evalúa a `True`
   ◯ `C A loop == C B loop` se evalúa a `False`

26. Considérense la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)` y la función `mal = head []`. ¿Cuál de las siguientes afirmaciones es **cierta**?
   ◯ `C mal A <= C mal B` se evalúa a `True`
   ⊗ `A <= C mal mal && B <= C mal mal` se evalúa a `True`
   ◯ `C A mal == C B mal` se evalúa a `True`

27. Considérese la definición del tipo `data T = A | B | C T T deriving (Eq,Ord)` y la función `mal = head []` y considérense las siguientes afirmaciones:
   (i) `C mal A <= C mal B` se evalúa a `True`
   (ii) `C A mal == C B mal` se evalúa a `True`
   (iii) `A <= C mal mal && B <= C mal mal` se evalúa a `True`
   ⊗ Exactamente una es cierta
   ◯ Exactamente dos son ciertas
   ◯ Las dos anteriores son falsas.

28. En el siguiente fragmento de código
```
data T a = A | (Int,T a)
f x (y:xs) = y
```
   ⊗ La definición de `T` contiene algún error sintáctico, pero la de `f` no.
   ◯ La definición de `f` contiene algún error sintáctico, pero la de `T` no.
   ◯ Las dos anteriores son falsas.

29. En el siguiente fragmento de código
```
data T a = A | (Int,T a)
f x (x:xs) = True
f x (y:xs) = f x xs
```
   ◯ La definición de `T` contiene algún error, pero la de `f` no.
   ◯ La definición de `f` contiene algún error, pero la de `T` no.
   ⊗ Las dos anteriores son falsas.

30. En lo que sigue, $\leq$ indica el orden estándar (el obtenido por `deriving Ord`) para tipos con constructoras de datos. Considérense las afirmaciones siguientes: (i) $[\,] \leq [0] \leq [0,1]$

31. (ii) $[\,] \leq [0,1] \leq [1,0]$. Entonces:
   ◯ (i) es cierta pero (ii) no
   ◯ (ii) es cierta pero (i) no
   ⊗ Las dos son ciertas

32. Considérese la declaración de clase
```
class C a where  f, g, h::  a -> Int
                 f x = g x + 1
                 g x = f x - 1
```
   ¿Qué afirmación es correcta?
   ◯ El sistema dará un error con esta definición
   ◯ Al declarar una instancia de `C` no es obligado definir `h` ni redefinir `f, g`
   ⊗ Al declarar una instancia de `C` es obligado definir `h` y razonable redefinir `f` o bien `g`

33. Considérense la declaraciones de clase e instancia
```
class C a where  f, g::  Int -> a
                 f x = g (x + 1)
                 g x = f (x - 1)
```
```
instance C Bool where  g x = (x == 0)
```
```
instance C Int where  f x = x
```
   ¿Qué afirmación es correcta?
   ◯ `f 0 && g 0` se evalúa a `False` y `f 1` se evalúa a `1`
   ⊗ La evaluación de exactamente una de las expresiones del caso anterior da un error
   ◯ Las dos anteriores son falsas.

34. Considérense la declaraciones de clase e instancia
```
class C a where  f, g::  Int -> a
                 f x = g 0
                 g x = f 1
```
```
instance C Bool where  g 0 = True
                       g _ = False
```
```
instance C Int where  f x = x
                      g x = 2*x
```
   ¿Qué afirmación es **falsa**?
   ◯ Al intentar evaluar `f 0` resulta un error de ambigüedad de tipos
   ◯ `not (f 0)` se evalúa a `False`
   ⊗ `f 0 == f 0` se evalúa a `True`

35. Considérense la declaraciones de clase e instancia
```
class C a where  f, g::  a -> Int
                 f x = g x + 1
                 g x = f x - 1
```
```
instance C Bool where  g True = 0
                       g False = 1
```
```
instance C Int where  f x = x
                      g x = 2*x
```
   ¿Qué afirmación es correcta?
   ⊗ `f False` se evalúa a 2 y `g 1` se evalúa a 2
   ◯ La evaluación de una de las expresiones del caso anterior da un error
   ◯ Las dos anteriores son falsas.

36. Considérese la declaración de clase
```
class C a where  f, g::  a -> Int
                 f x = g x + 1
                 g x = f x - 1
```
   ¿Qué afirmación es correcta?

○ Esa declaración es errónea, porque `f` y `g` no terminarán nunca
⊗ {`f`} es un conjunto minimal suficiente de métodos de `C`
○ {`f,g`} es un conjunto minimal suficiente de métodos de `C`

37. Considérese la declaración de clase

```
class C a where  f, g::  a -> Int
                 f x = g x + 1
```

¿Qué afirmación es correcta?
○ Esa declaración es errónea, porque `g` no está definida
⊗ {`g`} es un conjunto minimal suficiente de métodos de `C`
○ {`f`} es un conjunto minimal suficiente de métodos de `C`