Soluciones *Preguntas Tipo*

Lukas Haring

Test *listas.*

NOTAS.

1. Los **paréntesis añaden preferencia** a las operaciones.
2. La **lista vacía** [] contiene **cualquier tipo** dentro.

---

1. Dadas las expresiones:
   `[True,[]] True:[] [True]:[] [[True],[]]`

```
> [True,[]]    = --Error de tipado
> True:[]      = [True]
> [True]:[]    = [[True]]
> [[True],[]]  = [[True],[]]
```

☒ Exactamente una de las expresiones está mal tipada

☐ Exactamente dos de las expresiones están mal tipadas

☐ Las dos anteriores son falsas.

---

2. Sean las cuatro expresiones:
   `[True:[]] []:[True] [True]:[] [[True],[]]`

```
> [True:[]]    = [[True]]
> []:[True]    = --Error de tipado
> [True]:[]    = [[True]]
> [[True],[]]  = [[True],[]]
```

☐ Dos de ellas están mal tipadas

☒ Dos de ellas son sintácticmente equivalentes y una está mal tipada

☐ Las dos anteriores son falsas.

3. Dadas las expresiones:
`0:[1] 0:[1]:[2] 0:[1,2] [0,1]:[[2]] ([]:[],2)`

```
> 0:[1]        = [0, 1]
> 0:[1]:[2]    = --Error de tipado
> 0:[1,2]      = [0, 1, 2]
> [0,1]:[[2]]  = [[0, 1],[2]]
> ([]:[],2)    = ([[]], 2)
```

☒ Exactamente una de las expresiones está mal tipada

☐ Exactamente dos de las expresiones están mal tipadas tipada

☐ Las dos anteriores son falsas.

---

4. Dadas las expresiones:
`[]:[1] [1:[2]]:[] [1:[2]]:[[]] [1,1]:(2:[]) (1:[]):[]`

```
> []:[1]        = --Error de tipado
> [1:[2]]:[]    = [[1:[2]]]      = [[[1, 2]]]
> [1:[2]]:[[]]  = [[1:[2]], []]  = [[[1, 2]], []]
> [1,1]:(2:[])  = [1, 1]:([2])   = --Error de tipado
> (1:[]):[]     = ([1]):[]       = [[1]]
```

☐ Exactamente tres de las expresiones están mal tipadas

☐ Exactamente dos de las expresiones están mal tipadas

☒ Las dos anteriores son falsas.

---

5. Dadas las expresiones:
`0:[1] 0:[1]:[2] 0:[1,2] [0,1]:[2] (1:[]):[]`

```
> 0:[1]      = [0, 1]
> 0:[1]:[2]  = --Error de tipado
> 0:[1,2]    = [0, 1, 2]
> [0,1]:[2]  = --Error de tipado
> (1:[]):[]  = ([1]):[] = [[1]]
```

☐ Exactamente tres de las expresiones están mal tipadas

☒ Exactamente dos de las expresiones están mal tipadas

☐ Las dos anteriores son falsas.

6. Dadas las expresiones:
   `[1]:[] [[]]:[] []:[] (1:2):[] 1:(2:[])`

```
> [1]:[]    = [[1]]
> [[]]:[]   = [[[]]]
> []:[]     = [[]]
> (1:2):[]  = --Error de tipado
> 1:(2:[])  = 1:([2]) = [1, 2]
```

☒ Exactamente una de las expresiones está mal tipada

☐ Exactamente dos de las expresiones están mal tipadas tipada

☐ Las dos anteriores son falsas.

---

7. Dadas las expresiones:
   `[0]:[1] []:[[]]:[] [0]:[[]]:[] [0]:[[1,2]] ([[]]:[],[1])`

```
> [0]:[1]        = --Error de tipado
> []:[[]]:[]     = []:[[[]]] = [[],[[]]]
> [0]:[[]]:[]    = [0]:[[[]]] = --Error de tipado
> [0]:[[1,2]]    = [[0], [1, 2]]
> ([[]]:[],[1])  = ([[[]]], [1])
```

☒ Exactamente una de las expresiones está mal tipada

☐ Exactamente dos de las expresiones están mal tipadas tipada

☐ Exactamente tres de las expresiones están mal tipadas tipada

---

8. Dadas las expresiones:
   `[1]:[] [1]:[[]]:[2] [1]:[[[]]] 0:1:2 (0:[1],2)`

```
> [1]:[]         = [[1]]
> [1]:[[]]:[2]   = --Error de tipado
> [1]:[[[]]]     = --Error de tipado
> 0:1:2          = --Error de tipado
> (0:[1],2)      = ([0,1],2)
```

☐ Exactamente una de las expresiones está mal tipada

☐ Exactamente dos de las expresiones están mal tipadas tipada

☒ Las dos anteriores son falsas.

9. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `[[[0],[],[2,2]]]`?

```
> ((0:[]):[[],2:2:[]]):[] = (([0]):[[],2:[2]]):[] =
  ([[0],[],[2,2]]):[] = [[[0],[],[2,2]]]
> [0]:[]:[2,2]:[]:[]      = [0]:[]:[2,2]:[[]] = --Error de tipos
```

☒ `((0:[]):[[],2:2:[]]):[]`

☐ `[0]:[]:[2,2]:[]:[]`

☐ Ninguna de las anteriores, porque de hecho la expresión está mal tipada

---

10. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `[[1,[]],[3,4]]`?

☐ `((1:[]):[[],3:4:[]]):[]`

☐ `[0]:[]:[2,2]:[]:[]`

☒ Ninguna de las anteriores, porque de hecho la expresión está mal tipada

---

11. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `[[3,4],[]]`?

```
> 3:4:([],[])     = --Mal tipada
> (3:4:[]):[]:[]  = (3:[4]):[]:[] = ([3, 4]):[[]] = [[3, 4],[]]
```

☐ `3:4:([],[])`

☒ `(3:4:[]):[]:[]`

☐ Ninguna de las anteriores, porque de hecho la expresión está mal tipada

---

12. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `(1:[]):(1:2:[]):[]`?

```
> (1:[]):(1:2:[]):[] = ([1]):(1:[2]):[] = ([1]):([1,2]):[]
 = ([1]):[[1,2]] = [[1],[1,2]]
```

☒ `[[1],[1,2]]`

☐ `[[1],[1,2],[]]`

☐ Ninguna de las anteriores, porque de hecho la expresión está mal tipada

13. ¿Cuántas de las siguientes expresiones son sintácticamente equivalentes a `[[1,2],[]]`?
`[1:2:[]] 1:2:[[]] [1,2]:[[]] [1:[2],[]]`

```
> [1:2:[]]    = [1:[2]] = [[1, 2]]
> 1:2:[[]]    = --Error de tipado
> [1,2]:[[]]  = [[1,2],[]]
> [1:[2],[]]  = [[1,2],[]]
```

☒ Exactamente dos

☐ Exactamente tres

☐ Exactamente cuatro

---

14. Considérense las expresiones
`[[1],[2]] [1]:[[2]]:[] [1]:[[2]] [1]:[2,[]] [1,2]:[]`
¿Cuál de las siguientes afirmaciones es cierta?

```
> [[1],[2]]      = [[1],[2]]
> [1]:[[2]]:[]   = [1]:[[[2]],[]] = --Error de tipado
> [1]:[[2]]      = [[1],[2]]
> [1]:[2,[]]     = -- Error de tipado
> [1,2]:[]       = [[1,2]]
```

☐ La primera, la tercera y al menos otra más son sintácticamente equivalentes entre sí

☐ La segunda, la cuarta y al menos otra más son sintácticamente equivalentes entre sí

☒ Las dos anteriores son falsas.

---

15. Considérense las expresiones
`[[1,2]] ([1]:[[2]]):[] (1:[2]):[] [1,2]:[] [1]:[2]:[]`
¿Cuál de las siguientes afirmaciones es cierta?

```
> [[1,2]]          = [[1,2]]
> ([1]:[[2]]):[]   = ([[1],[2]]):[] = [[[1],[2]]]
> (1:[2]):[]       = ([1,2]):[] = [[1,2]]
> [1,2]:[]         = [[1,2]]
> [1]:[2]:[]       = [1]:[[2]] = [[1],[2]]
```

☒ La primera, la tercera y al menos otra más son sintácticamente equivalentes entre sí

☐ La segunda, la cuarta y al menos otra más son sintácticamente equivalentes entre sí

☐ Las dos anteriores son falsas.

16. ¿Cuántas de las siguientes expresiones son sintácticamente equivalentes a `[[1,2],[1]]`?
`[1:2:[],1:[]] [1:[2],[1]] [1,2]:[1]:[] (1:2:[]):[[1]]`

```
> [1:2:[],1:[]]    = [1:[2],[1]] = [[1,2],1]
> [1:[2],[1]]      = [[1,2],[1]]
> [1,2]:[1]:[]     = [1,2]:[[1]] = [[1,2],[1]]
> (1:2:[]):[[1]]   = (1:[2]):[[1]] = ([1,2]):[[1]] = [[1,2],[1]]
```

☐ Exactamente dos

☐ Exactamente tres

☒ Exactamente cuatro

Test *equivalencia de expresiones.*

Aplicaciones

NOTAS.

1. Añadir paréntesis a la expresión general y aplicar recursivamente, quitar paréntesis aquellos que a su izquierda tenga otro paréntesis de apertura.
2. Equivalencias de operadores.
   $a \oplus b = (\oplus) a b = (\oplus b) a = ((\oplus) a) b = (a \oplus) b.$
3. Los operadores tienen menos precedencia que las funciones, (+) (f a b) (g b) = (f a b) + (g b) = f a b + g b
4. Si infixr <o> <0-9> => A<o>B<o>C = A<o>(B<o>C) o si infixl <o> <0-9> => A<o>B<o>C = (A<o>B)<o>C

---

1. Considérense las expresiones (solo difieren en los paréntesis):

```
{ e_1 = (f ((x y) y)) (f 0)
{ e_2 = f (x y y) (f 0)
{ e_3 = f (x y y) f 0
```

```
(e_1) = ((f ((x y) y)) (f 0)) = ((f (x y y)) (f 0)) = (f (x y y) (f 0))
```

☐ $e_1 \equiv e_2 \equiv e_3$

☐ $e_1 \not\equiv e_2 \equiv e_3$

☒ $e_1 \equiv e_2 \not\equiv e_3$

---

2. Considérense las expresiones (solo difieren en los paréntesis):

```
{ e_1 = f (f x (y^2)) y
{ e_2 = f ((f x) ((^) y 2)) y
{ e_3 = f (f x (y^) 2) y
```

```
e_2 = f ((f x) ((^) y 2)) y = f ((f x) (y ^ 2)) y = f (f x (y ^ 2)) y
e_3 = f (f x (y^) 2) y = f (f x (y ^ 2)) y
```

☒ $e_1 \equiv e_2 \equiv e_3$

☐ $e_1 \equiv e_2 \not\equiv e_3$

☐ $e_1 \not\equiv e_2 \not\equiv e_3$

3. Considérense las expresiones (solo difieren en los paréntesis):

$$\begin{cases} \texttt{e\_1 = f (z (y x)) ((z 0) x)} \\ \texttt{e\_2 = (f (z (y x))) (z 0 x)} \\ \texttt{e\_3 = f z (y x) (z 0 x)} \end{cases}$$

```
(e_1) = (f (z (y x)) ((z 0) x)) = (f (z (y x)) (z 0 x))
(e_2) = ((f (z (y x))) (z 0 x)) = (f (z (y x)) (z 0 x))
(e_3) = (f z (y x) (z 0 x))
```

☐ $e_1 \equiv e_2 \equiv e_3$

☒ $e_1 \equiv e_2 \not\equiv e_3$

☐ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

---

4. Considérense las expresiones (solo difieren en los paréntesis):

$$\begin{cases} \texttt{e\_1 = f x (g x,y/2)} \\ \texttt{e\_2 = f x (g x) (y/2)} \\ \texttt{e\_3 = (f x) (g x,(/y) 2)} \end{cases}$$

```
(e_3) = ((f x) (g x,(/y) 2)) = (f x (g x, 2/y))
```

☐ $e_1 \equiv e_2 \equiv e_3$

☒ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

☐ $e_1 \equiv e_3 \not\equiv e_2$

---

5. Considérense las expresiones (solo difieren en los paréntesis):

$$\begin{cases} \texttt{e\_1 = f x (g (x+1) y)} \\ \texttt{e\_2 = (f x) (g (((+) x) 1) y)} \\ \texttt{e\_3 = (f x) (g (x+1)) y} \end{cases}$$

```
(e_2) = ((f x) (g (((+) x) 1) y)) = (f x (g (x+1) y))
(e_3) = ((f x) (g (x+1)) y) = (f x (g (x+1)) y)
```

☒ $e_1 \equiv e_2 \not\equiv e_3$

☐ $e_1 \not\equiv e_2 \equiv e_3$

☐ $e_1 \equiv e_2 \equiv e_3$

6. Considérense las expresiones (solo difieren en los paréntesis):
```
e_1 = f x (g x,y+1)
e_2 = f x (g x) (y+1)
e_3 = (f x) (g x,(+) y 1)
```

```
(e_3) = ((f x) (g x,(+) y 1)) = (f x (g x, y+1))
```

☐ $e_1 \equiv e_2 \equiv e_3$

☐ $e_1 \not\equiv e_2 \not\equiv e_3$

☒ $e_1 \equiv e_3 \not\equiv e_2$

---

7. Considérense las expresiones (solo difieren en los paréntesis):
```
e_1 = f x g (x+1) y
e_2 = (f x) (g (x+1) y)
e_3 = (f x) g ((+) x 1) y
```

```
(e_2) = ((f x) (g (x+1) y)) = (f x (g (x+1) y))
(e_3) = ((f x) g ((+) x 1) y) = (f x g (x+1) y)
```

☐ $e_1 \equiv e_2 \not\equiv e_3$

☒ $e_1 \equiv e_3 \not\equiv e_2$

☐ Las dos anteriores son falsas

---

8. Considérense las expresiones (solo difieren en los paréntesis):
```
e_1 = f x 1 (x + y)
e_2 = (f x 1) (x + y)
e_3 = f x 1 ((+) x y)
```

```
(e_2) = ((f x 1) (x + y)) = (f x 1 (x + y))
(e_3) = (f x 1 ((+) x y)) = (f x 1 (x + y))
```

☐ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

☐ $e_1 \equiv e_3 \not\equiv e_2$

☒ $e_1 \equiv e_2 \equiv e_3$

9. Suponiendo la declaración `infixr 9 !`, considérense las expresiones (que solo difieren en los paréntesis):

$$\left\{\begin{array}{l}\texttt{e\_1 = ((! g) f) ! ((h !) i) ! j} \\ \texttt{e\_2 = ((!) (f ! g)) ((h ! i) ! j)} \\ \texttt{e\_3 = (!) ((!) f g) ((!) ((!) h i) j)}\end{array}\right.$$

```
e_1 = ((! g) f) ! ((h !) i) ! j = (f ! g) ! (h ! i) ! j
e_2 = ((!) (f ! g)) ((h ! i) ! j) = (f ! g) ! ((h ! i) ! j)
e_3 = (!) ((!) f g) ((!) ((!) h i) j) = (f ! g) ! ((h ! i) ! j)
```

☐ $e_1 \equiv e_2 \equiv e_3$

☐ $e_1 \equiv e_2 \not\equiv e_3$

☒ $e_1 \not\equiv e_2 \equiv e_3$

---

10. Considérense las expresiones (solo difieren en los paréntesis):

$$\left\{\begin{array}{l}\texttt{e\_1 = f x (y-1):z} \\ \texttt{e\_2 = ((:) f) x ((-) y 1) z} \\ \texttt{e\_3 = (: z) ((f x) ((-) y 1))}\end{array}\right.$$

```
(e_2) = (((:) f) x ((-) y 1) z) = (f : x (y-1) z)
(e_3) = ((: z) ((f x) ((-) y 1))) = (((f x) (y-1)):z) = ((f x (y-1)):z)
(e_3) = (f x (y-1):z)
```

☐ $e_1 \equiv e_2 \equiv e_3$

☐ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

☒ $e_1 \equiv e_3 \not\equiv e_2$

---

11. Considérense las expresiones (solo difieren en los paréntesis):

$$\left\{\begin{array}{l}\texttt{e\_1 = f ((g x) (x y)) x} \\ \texttt{e\_2 = f (g x) (x y) x} \\ \texttt{e\_3 = (f (g x (x y))) x}\end{array}\right.$$

```
e_1 = f ((g x) (x y)) x = f (g x (x y)) x
(e_3) = ((f (g x (x y))) x) = (f (g x (x y)) x)
```

☐ $e_1 \equiv e_2 \not\equiv e_3$

☒ $e_1 \equiv e_3 \not\equiv e_2$

☐ $e_1 \equiv e_2 \equiv e_3$

12. Considérense las expresiones (solo difieren en los paréntesis):

```
e_1 = f x y + z 4
e_2 = f x ((+) y (z 4))
e_3 = (+) ((f x) y) (z 4)
```

```
e_1 = f x y + z 4 = (f x y) + (z 4)
e_2 = f x ((+) y (z 4)) = f x (y + (z 4))
(e_3) = ((+) ((f x) y) (z 4)) = (((f x) y) + (z 4)) = ((f x y) + (z 4))
```

☐ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

☐ $e_1 \equiv e_2 \equiv e_3$

☒ $e_1 \equiv e_3 \not\equiv e_2$

---

13. Considérense las expresiones (solo difieren en los paréntesis):

```
e_1 = f x + g z 4
e_2 = f x ((+) g (z 4))
e_3 = (+) (f x) (g z 4)
```

Igual que el ejercicio anterior.

```
e_1 = f x + g z 4 =  (f x) + (g z 4)
e_2 = f x ((+) g (z 4)) = f x (g + (z 4))
(e_3) = ((+) (f x) (g z 4)) = ((f x) + (g z 4))
```

☐ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

☐ $e_1 \equiv e_2 \equiv e_3$

☒ $e_1 \equiv e_3 \not\equiv e_2$

Flecha de los tipos.

NOTAS.

1. Añadir paréntesis a la expresión general y aplicar recursivamente, quitar paréntesis aquellos que a su derecha tenga otro paréntesis de apertura.

---

1. Considérense las expresiones de tipo (solo difieren en los paréntesis):

```
t_1 = (a -> a) -> (a -> a) -> (a -> a)
t_2 = (a -> a) -> (a -> a -> a -> a)
t_3 = (a -> a) -> (a -> a) -> a -> a
```

```
(t_1) = ((a -> a) -> (a -> a) -> (a -> a))
(t_1) = ((a -> a) -> (a -> a) -> a -> a)

(t_2) = ((a -> a) -> (a -> a -> a -> a))
(t_2) = ((a -> a) -> a -> a -> a -> a)
```

☐ $t_1 \equiv t_2 \not\equiv t_3$

☒ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \equiv t_2 \equiv t_3$

---

2. Considérense las expresiones de tipo (solo difieren en los paréntesis):

```
t_1 = ((b -> a) -> a) -> ((a -> b) -> (b -> b))
t_2 = (b -> (a -> a)) -> (a -> b) -> b -> b
t_3 = (b -> a -> a) -> (a -> b -> b -> b)
```

```
(t_1) = (((b -> a) -> a) -> ((a -> b) -> (b -> b)))
(t_1) = (((b -> a) -> a) -> ((a -> b) -> b -> b))
(t_1) = (((b -> a) -> a) -> (a -> b) -> b -> b)

t_2 = (b -> (a -> a)) -> (a -> b) -> b -> b
t_2 = (b -> a -> a) -> (a -> b) -> b -> b

(t_3) = ((b -> a -> a) -> (a -> b -> b -> b))
(t_3) = ((b -> a -> a) -> a -> b -> b -> b)
```

☐ $t_1 \equiv t_2 \not\equiv t_3$

☐ $t_1 \equiv t_2 \equiv t_3$

☒ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

3. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = (a -> (b -> a) -> a) -> b -> b} \\ \texttt{t\_2 = (a -> ((b -> a) -> a)) -> (b -> b)} \\ \texttt{t\_3 = a -> b -> a -> a -> b -> b} \end{cases}$$

```
(t_2) = ((a -> ((b -> a) -> a)) -> (b -> b))
(t_2) = ((a -> (b -> a) -> a) -> b -> b)
```

☒ $t_1 \equiv t_2 \not\equiv t_3$

☐ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

☐ $t_1 \equiv t_3 \not\equiv t_2$

---

4. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = (a -> b -> a) -> (a -> b -> b)} \\ \texttt{t\_2 = (a -> b -> a) -> a -> (b -> b)} \\ \texttt{t\_3 = (a -> (b -> a)) -> a -> b -> b} \end{cases}$$

```
(t_1) = ((a -> b -> a) -> (a -> b -> b))
(t_1) = ((a -> b -> a) -> a -> b -> b)

(t_2) = ((a -> b -> a) -> a -> (b -> b))
(t_2) = ((a -> b -> a) -> a -> b -> b)

(t_3) = ((a -> (b -> a)) -> a -> b -> b)
(t_3) = ((a -> b -> a) -> a -> b -> b)
```

☐ $t_1 \equiv t_2 \not\equiv t_3$

☐ $t_1 \equiv t_3 \not\equiv t_2$

☒ $t_1 \equiv t_2 \equiv t_3$

---

5. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = a -> (b -> a -> a) -> b -> b} \\ \texttt{t\_2 = (a -> ((b -> a) -> a)) -> b -> b} \\ \texttt{t\_3 = a -> (b -> (a -> a)) -> (b -> b)} \end{cases}$$

```
t_2 = (a -> ((b -> a) -> a)) -> b -> b
t_2 = (a -> (b -> a) -> a) -> b -> b

(t_3) = (a -> (b -> (a -> a)) -> (b -> b))
(t_3) = (a -> (b -> a -> a) -> b -> b)
```

☐ $t_1 \equiv t_2 \not\equiv t_3$

☒ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \equiv t_2 \equiv t_3$

6. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = (a -> b -> a -> a) -> b -> b} \\ \texttt{t\_2 = a -> b -> a -> a -> b -> b} \\ \texttt{t\_3 = (a -> b -> (a -> a)) -> (b -> b)} \end{cases}$$

```
(t_3) = ((a -> b -> (a -> a)) -> (b -> b))
(t_3) = ((a -> b -> a -> a) -> b -> b)
```

☐ $t_1 \equiv t_2 \not\equiv t_3$

☒ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \equiv t_2 \equiv t_3$

---

7. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = a -> a -> (a -> a) -> (b -> b)} \\ \texttt{t\_2 = a -> (a -> ((a -> a) -> b -> b))} \\ \texttt{t\_3 = (a -> a) -> (a -> a) -> b -> b} \end{cases}$$

```
(t_1) = (a -> a -> (a -> a) -> (b -> b))
(t_1) = (a -> a -> (a -> a) -> b -> b)

(t_2) = (a -> (a -> ((a -> a) -> b -> b)))
(t_2) = (a -> (a -> (a -> a) -> b -> b))
(t_2) = (a -> a -> (a -> a) -> b -> b)
```

☐ $t_1 \equiv t_2 \equiv t_3$

☐ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

☒ $t_1 \equiv t_2 \not\equiv t_3$

---

8. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = a -> ((a -> a) -> a)} \\ \texttt{t\_2 = a -> (a -> a) -> a} \\ \texttt{t\_3 = a -> a -> a -> a} \end{cases}$$

```
(t_1) = (a -> ((a -> a) -> a))
(t_1) = (a -> (a -> a) -> a)
```

☒ $t_1 \equiv t_2 \not\equiv t_3$

☐ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

9. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = (a -> a) -> (a -> a) -> (a -> a)} \\ \texttt{t\_2 = a -> a -> (a -> a) -> (a -> a)} \\ \texttt{t\_3 = (a -> a) -> (a -> a) -> a -> a} \end{cases}$$

```
(t_1) = ((a -> a) -> (a -> a) -> (a -> a))
(t_1) = ((a -> a) -> (a -> a) -> a -> a)

(t_2) = (a -> a -> (a -> a) -> (a -> a))
(t_2) = (a -> a -> (a -> a) -> a -> a)
```

☐ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

☒ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \equiv t_2 \not\equiv t_3$

---

10. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = (b -> a -> a) -> (a -> b) -> b} \\ \texttt{t\_2 = (b -> (a -> a)) -> ((a -> b) -> b)} \\ \texttt{t\_3 = (b -> a -> a) -> a -> b -> b} \end{cases}$$

```
(t_2) = ((b -> (a -> a)) -> ((a -> b) -> b))
(t_2) = ((b -> a -> a) -> (a -> b) -> b)
```

☐ $t_1 \equiv t_2 \equiv t_3$

☐ $t_1 \equiv t_3 \not\equiv t_2$

☒ $t_1 \equiv t_2 \not\equiv t_3$

---

11. Considérense las expresiones de tipo (solo difieren en los paréntesis):

$$\begin{cases} \texttt{t\_1 = a -> b -> (c -> d)} \\ \texttt{t\_2 = a -> (b -> c -> d)} \\ \texttt{t\_3 = a -> b -> c -> d} \end{cases}$$

```
(t_1) = (a -> b -> (c -> d))
(t_1) = (a -> b -> c -> d)

(t_2) = (a -> (b -> c -> d))
(t_2) = (a -> b -> c -> d)
```

☒ $t_1 \equiv t_2 \equiv t_3$

☐ $t_1 \equiv t_3 \not\equiv t_2$

☐ $t_1 \not\equiv t_2 \not\equiv t_3$

12. Considérense las expresiones de tipo (solo difieren en los paréntesis):

```
t_1 = (a -> a) -> (a -> a) -> (a -> a)
t_2 = (a -> a) -> (a -> a) -> a -> a
t_3 = (a -> a) -> ((a -> a) -> (a -> a))
```

```
(t_1) = ((a -> a) -> (a -> a) -> (a -> a))
(t_1) = ((a -> a) -> (a -> a) -> a -> a)

(t_3) = ((a -> a) -> ((a -> a) -> (a -> a)))
(t_3) = ((a -> a) -> (a -> a) -> a -> a)
```

$\square$ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

$\square$ $t_1 \equiv t_3 \not\equiv t_2$

$\boxtimes$ $t_1 \equiv t_2 \equiv t_3$

---

13. Considérense las expresiones de tipo (solo difieren en los paréntesis):

```
t_1 = (a -> a -> a) -> (a -> a) -> (a -> a)
t_2 = (a -> (a -> a)) -> (a -> a) -> a -> a
t_3 = (a -> a -> a) -> ((a -> a) -> a -> a)
```

```
(t_1) = ((a -> a -> a) -> (a -> a) -> (a -> a))
(t_1) = ((a -> a -> a) -> (a -> a) -> a -> a)

t_2 = (a -> (a -> a)) -> (a -> a) -> a -> a
t_2 = (a -> a -> a) -> (a -> a) -> a -> a

t_3 = ((a -> a -> a) -> ((a -> a) -> a -> a))
t_3 = ((a -> a -> a) -> (a -> a) -> a -> a)
```

$\square$ $t_1 \not\equiv t_2 \not\equiv t_3 \not\equiv t_1$

$\square$ $t_1 \equiv t_3 \not\equiv t_2$

$\boxtimes$ $t_1 \equiv t_2 \equiv t_3$

Test *evaluación de expresiones.*

NOTAS.

1. Poner Paréntesis desde `(let ... in ...)` para todos los que encontremos, luego mirar si alguna de las variables no está en el scope
2. Una vez hecho eso, sustituir el valor de las variables declaradas en `let` por las que aparecen en `in`.
3. El orden declarado en el `let {}` **no importa el orden de inicialización**.
4. La declaración `let x = x in ...` es totalmente válida, **provoca una recursividad**.
5. El orden de **declaración** de variables **en listas es importante**. ***
6. Considérense las expresiones siguientes:

```
1> (let x=5 in x+x) + 3
2> let x=2 in let y=x+x in y*y*x
3> let y=(let x=2 in x+x) in y*y
4> let x=2 in let y=x+x in y*y
5> let y=x+x in let x=2 in y*y*x
6> let y=(let x=2 in x+x) in y*y*x
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> (let x=5 in x+x) + 3                -- Bien, valor 13
2> (let x=2 in (let y=x+x in y*y*x)) -- Bien, valor 32
3> (let y=(let x=2 in x+x) in y*y))  -- Bien, valor 16
4> (let x=2 in (let y=x+x in y*y))   -- Bien, valor 8
5> (let y=x+x in (let x=2 in y*y*x)) -- Mal, x fuera del ámbito
6> (let y=(let x=2 in x+x) in y*y*x) -- Mal, x fuera del ámbito
```

- ☐ Exactamente tres de ellas
- ☒ Exactamente dos de ellas
- ☐ Todas están correctamente formadas

2. Considérense las expresiones siguientes:

```
1> (let x=5 in x+x) + 5
2> let x=2 in let y=x+x in y*x
3> let x=2 in let y=x+x in x
4> let x=y in let x=2 in y*y*x
5> let y=(let x=2 in x+x) in y*y
6> let y=(let x=2 in x+x) in y*y*x
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> (let x=5 in x+x) + 5                -- Bien, valor 15
2> (let x=2 in (let y=x+x in y*x))     -- Bien, valor 8
3> (let x=2 in (let y=x+x in x))       -- Bien, valor 2
4> (let x=y in (let x=2 in y*y*x))     -- Mal, y fuera del ámbito
5> (let y=(let x=2 in x+x) in y*y)     -- Bien, valor 16
6> (let y=(let x=2 in x+x) in y*y*x)   -- Mal, x fuera del ámbito
```

☐ Exactamente una de ellas

☒ Exactamente dos de ellas

☐ Exactamente tres de ellas

---

3. Considérense las expresiones siguientes:

```
1> (let x=5 in x+x) + x
2> let x=2 in let y=x+x in y*y*x
3> let y=x+x in let x=2 in y*y*x
4> let {y=x+x;x=2} in y*y*x
5> let y=(let x=2 in x+x) in y*y*x
6> let y=(let x=2 in 3) in y*y
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> (let x=5 in x+x) + x                -- Mal, x fuera del ámbito
2> (let x=2 in (let y=x+x in y*y*x))   -- Bien, valor 32
3> (let y=x+x in (let x=2 in y*y*x))   -- Mal, x fuera del ámbito
4> (let {y=x+x;x=2} in y*y*x)          -- Bien, valor 32
5> (let y=(let x=2 in x+x) in y*y*x)   -- Mal, x fuera del ámbito
6> (let y=(let x=2 in 3) in y*y)       -- Bien, valor 9
```

☐ Exactamente dos de ellas

☒ Exactamente tres de ellas

☐ Exactamente cuatro de ellas

4. Considérense las expresiones siguientes:

```
1> (let x=5 in x+x) + 5
2> let x=2 in let y=x+x in y*x
3> let x=2 in let y=x+x in x
4> let x=y in let x=2 in y*y*x
5> let y=(let x=x in x+x) in y*y
6> let y=(let x=2 in x+x) in y*y*x
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> (let x=5 in x+x) + 5              -- Bien, valor 15
2> (let x=2 in (let y=x+x in y*x))   -- Bien, valor 8
3> (let x=2 in (let y=x+x in x))     -- Bien, valor 4
4> (let x=y in (let x=2 in y*y*x))   -- Mal, y fuera del ámbito
5> (let y=(let x=x in x+x) in y*y)   -- Bien, valor indefinido, recursivo.
6> (let y=(let x=2 in x+x) in y*y*x) -- Mal, x fuera del ámbito.
```

☐ Exactamente una de ellas

☒ Exactamente dos de ellas

☐ Exactamente tres de ellas

---

5. Considérense las expresiones siguientes:

```
1> (let x=5 in x+x) + (let x=3 in 2*x)
2> let y=x+x in let x=2 in y*y*x
3> let x=2 in let y=x+x in y*y*x
4> let {y=x+x;x=2} in y*y*x
5> [i | i<-[1..j],j<-[0..100],mod j 3 == 0]
6> [i | j<-[0..100],i<-[1..j],mod j 3 == 0]
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> (let x=5 in x+x) + (let x=3 in 2*x)        -- Bien, valor 16
2> (let y=x+x in (let x=2 in y*y*x))          -- Mal, x fuera del ámbito
3> (let x=2 in (let y=x+x in y*y*x))          -- Bien, valor 32
4> (let {y=x+x;x=2} in y*y*x)                 -- Bien, valor 32
5> [i | i<-[1..j],j<-[0..100],mod j 3 == 0]   -- Mal, j declarada después
6> [i | j<-[0..100],i<-[1..j],mod j 3 == 0]   -- Bien
```

☐ Exactamente una de ellas

☒ Exactamente dos de ellas

☐ Exactamente tres de ellas

6. Considérense las expresiones siguientes:

```
1> let x=1:x in head x
2> (\x -> (\y -> x+y)) x
3> let x=[1,2,3] in let y= x!!2 in y*last x
4> let {y=2*x;x=5} in y*y*x
5> [i+j | i<-[1..j],j<-[0..100],mod j i == 0]
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> let x=1:x in head x                         -- Bien, valor 1 (Ev. Perezosa)
2> (\x -> (\y -> x+y)) x                       -- Mal, x fuera de ámbito
3> (let x=[1,2,3] in (let y= x!!2 in y*last x)) -- Bien, valor 9
4> let {y=2*x;x=5} in y*y*x                    -- Bien, valor 500
5> [i+j | i<-[1..j],j<-[0..100],mod j i == 0]   -- Mal, j declarada después
```

☒ Exactamente dos de ellas

☐ Exactamente tres de ellas

☐ Exactamente cuatro de ellas

---

7. Considérense las expresiones siguientes:

```
1> \x -> ((\y -> x) x)
2> \x -> ((\y -> x+y) y)
3> let y=[1,2,3] in let x= y!!1 in x*head y
4> let {y=2*x;x=5} in y*y*x
5> [i+j | i<-[1..100],j<-[0..i],mod j i == 0]
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> \x -> ((\y -> x) x)                          -- Bien, valor es una función
2> \x -> ((\y -> x+y) y)                        -- Mal, y fuera de ámbito
3> let y=[1,2,3] in let x= y!!1 in x*head y     -- Bien, valor 2
4> let {y=2*x;x=5} in y*y*x                     -- Bien, valor 500
5> [i+j | i<-[1..100],j<-[0..i],mod j i == 0]   -- Bien
```

☒ Exactamente una de ellas

☐ Tres o más de ellas

☐ Las dos anteriores son falsas.

8. Considérense las expresiones siguientes:

```
1> \x -> ((\x y -> x+y) x y)
2> \x -> ((\x y -> x+y) x x)
3> let y= (let x = 1 in x+x) in x+y
4> let y= (let x = 1 in x+x) in y+y
5> [j | i<-[1..100],j<-[0..i]]
```

¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?

```
1> \x -> ((\x y -> x+y) x y)         -- Mal, y fuera de ámbito
2> \x -> ((\x y -> x+y) x x)         -- Bien, valor una función
3> let y= (let x = 1 in x+x) in x+y  -- Mal, y fuera de ámbito
4> let y= (let x = 1 in x+x) in y+y  -- Bien, valor 4
5> [j | i<-[1..100],j<-[0..i]]       -- Bien
```

☐ Exactamente una de ellas

☐ Tres o más de ellas

☒ Las dos anteriores son falsas.

Test *Funciones.*

NOTAS.

1. Composición de funciones `(.)::(b -> c) -> (a -> b) -> a -> c`

---

1. En el siguiente fragmento de código

```
> data T a = A | (Int,T a)
> f x (y:xs) = y
```

```
> data T a = A | (Int,T a)   -- Mal, falta <constructor> (Int, T a).
> f x (y:xs) = y             -- Bien, devuelve la cabeza del vector
```

☒ 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.

---

2. 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
```

```
> data T a = A | (Int,T a)   -- Mal, falta <constructor> (Int, T a)
> f x (x:xs) = True          -- Mal, doble declaración de x.
> f x (y:xs) = f x xs        -- Bien
```

☐ 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.

---

3. En el siguiente fragmento de código

```
> data T a = A | B (Int,T a)   -- Bien
> f x (x:xs) = xs              -- Mal, doble definición de x.
```

☐ 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.

4. Supongamos que `1::Int`, `(+)::Int->Int->Int`, y considérese la función `f` definida por las dos reglas siguientes:

```
> f True x y = (x,y)
> f False y x = (y,x+1)
```

Reescribimos.

```
> f True x y = (x,y)
> f False x y = (x,y+1)
```

Observamos que el **primer argumento es un** `Bool`, el **segundo argumento puede ser cualquier tipo** y el **tercero debe ser un número** ya que se le suma 1, como estamos en los **enteros** por la definición, es un `Int`.

☒ El tipo que se infiere para `f` es `Bool -> a -> Int -> (a,Int)`

☐ El tipo que se infiere para `f` es `Bool -> Int -> Int -> (Int,Int)`

☐ `f` está mal tipada.

---

5. Considérese la función definida por `f x y = y x x`. El tipo de `f` es:

```
> f x y = y x x
```

El **primer argumento puede cualquier tipo** (`x=a`), luego vemos que `y` es una **función con dos argumentos** que depende de `x`, que internamente puede devolver lo que quiera (Independientemente de la operación `a`).

```
> f::a->(a->a->b)->b
```

☒ `a -> (a -> a -> b) -> b`

☐ `a -> (a -> a -> a) -> a`

☐ No está bien tipada

---

6. Considérese la función definida por `f g x = x (g True) g`. El tipo de `f` es: Descomponemos desde más profundo a menos.

```
-- g Coge un booleano y devuelve lo que sea.
> g::(Bool -> c)
-- (g True) es constante por lo que es lo que se devolvió.
> x::c -> (Bool -> c) -> b
-- Devuelve el valor de x.
> f::(Bool -> c) -> (c -> (Bool -> c) -> b) -> b
```

☒ $\forall a, b.$`(Bool->a) -> (a -> (Bool->a) -> b) -> b`

☐ $\forall a.$`(Bool->a) -> (a -> (Bool->a) -> a) -> a`

☐ Está mal tipada.

7. Considérese la función definida por `f g x = x g g`. El tipo de `f` es:

```
-- g Puede ser lo que sea
> g::a
-- x tiene como argumento dos g y devuelve tipo a.
> x::a -> a -> b
-- Juntamos todo
> f::a -> (a -> a -> b) -> b
```

☒ $\forall a, b.$`a -> (a -> a -> b) -> b`

☐ $\forall a, b.$`(a -> a -> b) -> a -> b`

☐ Está mal tipada.

---

8. Considérese la función definida por `f g x = x g g`. El tipo de `f` es:

Igual que el ejercicio anterior.

☐ Está mal tipada.

☐ $\forall a.$`(a -> a -> a) -> a -> a`

☒ $\forall a, b.$`a -> (a -> a -> b) -> b`

---

9. Considérese la función definida por `f x y = x (y x)`. El tipo de `f` es:

```
-- Vemos que x acepta y y nos devuelve lo que sea.
> x::a -> b
-- Vemos que y depende de x y lo que se supone que y devolvió.
> y::(a -> b) -> a
-- Juntamos todo
> f::(a -> b) -> ((a -> b) -> a) -> b
```

☒ `(a -> b) -> ((a -> b) -> a) -> b`

☐ `(a -> b -> a) -> (b -> a) -> a`

☐ Está mal tipada.

---

10. Considérese la función definida por `f x y = y (x x)`. El tipo de `f` es:

Vemos que es un **tipo recurivo** ya que `x` es el propio argumento de su función, por lo que está **mal definido**.

☐ `(a -> a -> a) -> (a -> a) -> a`

☐ `(a -> b -> a) -> (b -> a) -> a`

☒ No está bien tipada.

11. Considérese la función definida por `f x y = x x y`. El tipo de `f` es:

Vemos que es un **tipo recurivo** ya que `x` es el propio argumento de su función, por lo que está **mal definido**.

- ☐ `(a -> a -> a) -> (a -> a) -> a`
- ☐ `(a -> b -> a) -> (b -> a) -> a`
- ☒ No está bien tipada.

---

12. Considérese la función definida por `f x y = x (y y)`. El tipo de `f` es:
    Igual que el ejercicio 10
    - ☐ `(a -> b) -> (a -> a) -> b`
    - ☐ `(a -> a) -> (a -> a) -> a`
    - ☒ No está bien tipada.

---

13. Considérese la función definida por `f g = g (f g)`. El tipo de `f` es:

```
-- El tipo de f deve devolverlo también g
> f:: (...) -> a
> g:: (...) -> a
-- Vemos que g necesita un argumento que tiene que ser al mismo de f.
> g:: a -> a
-- Juntamos todo.
> f:: (a -> a) -> a
-- Ejemplo.
> g x = True
> f g = g (f g) = True
```

- ☐ `a -> a -> a`
- ☒ `(a -> a) -> a`
- ☐ No está bien tipada.

14. Sea `f` definida por `f g x = x (x g)`. El tipo de `f` es:

```
-- Suponemos que g es de tipo a.
> g::a
-- Ahora vemos que x tiene como argumento g.
> x::a -> b
-- Ahora vemos que x vuelve a englobar a x. Eso implica que b = a.
> x::a -> a
-- Juntamos esto.
> f a -> (a -> a) -> a
```

☐ $\forall a$.`a -> (a -> b) -> b`

☒ $\forall a$.`a -> (a -> a) -> a`

☐ Está mal tipada.

---

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

```
-- Vemos que x necesita dos argumentos y que uno de sus argumentos
-- es el tipo que devuelve y g es un valor arbitrario.
> g::a
> x::b -> a -> b
-- juntamos todo.
> f::a -> (b -> a -> b) -> b
```

☐ $\forall a$.`a -> (a -> a -> a) -> a`

☒ $\forall a$.`a -> (b -> a -> b) -> b`

☐ Está mal tipada.

---

16. Sea `f` definida por `f x y z = x (y z)`. El tipo de `f` es:

```
-- z es un tipo cualquiera e "y" es una función que toma z y devuelve otro tipo.
> z::c
> y::c -> a
-- por último, x recibe otro argumento que es "y" y devuelve otro valor.
> x::a -> b
-- juntamos todo.
> f::(a -> b) -> (c -> a) -> c -> b
```

☐ `(a -> b) -> (b -> c) -> a -> c`

☒ `(a -> b) -> (c -> a) -> c -> b`

☐ Está mal tipada.

17. Sea `f` definida por `f x g = x (x (g True)))`. El tipo de `f` es:

```
-- Vemos que g es una función de un argumento booleano y devuelve cualquier tipo.
> g::Bool -> a
-- x toma el tipo de g y devuelve cualquier otro tipo
> x::a -> c
-- Pero ahora vemos que x vuelve a englobar x, por lo que c == a.
> x::a -> a
-- finalmente.
> (a -> a) -> (Bool -> a) -> a
```

- ☐ $\forall a, b.$`(a -> b) -> (Bool -> a) -> b`
- ☒ $\forall a.$`(a -> a) -> (Bool -> a) -> a`
- ☐ Está mal tipada.

---

18. Sea `f` definida por `f x y = (x y).(x y)`. El tipo de `f` es:

```
-- Vemos que x tiene 1 argumento, que es y (cuyo valor es un tipo)
> y::a
> x::a -> b
-- Ahora bien la solución es la composición consigomismo
-- (x::a -> b) . b (Ya que (x y) (solución x y = b) )
-- El resultado es otra función (sin evaluar) b -> b
> f::(a -> b -> b) -> a -> (b -> b)
```

- ☐ `(a -> a) -> (a -> a)`
- ☒ `(a -> b) -> (a -> b) -> (a -> b)`
- ☐ `(a -> b -> b) -> a -> b -> b`

---

19. Sea `f` definida por `f x y = x (x y)`. El tipo de `f` es:

```
-- Vemos que x tiene 1 argumento, que es y (cuyo valor es un tipo)
> y::a
> x::a -> b
-- Ahora vemos que x vuelve a englobar a x => b = a.
> x::a -> a
-- Juntamos todo
> f::(a -> a) -> a -> a
```

- ☒ `(a -> a) -> (a -> a)`
- ☐ `(a -> b) -> a -> b`
- ☐ `a -> b -> a`

Test *Clase de tipos.*

NOTAS.

1. Los **operadores** `(<=)`, `(=>)`, `...`, `(==)`, `(\=)` obligan a que los tipos sean **necesariamente Ordenables** (`Ord`) o que deriven de este y **ambos** sean del **mismo tipo**.
2. Los **operadores aritméticos** obligan a **derivar de** `Num`.

---

1. 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:

```
--
> if x <= y ... -- Deben ser comparable, por lo que tipo de x e y son Ord a.
> z + 1         -- Debe ser numérico, para poder sumarse, Num b.
-- Juntamos todo
> (Ord a, Num b) => a -> a -> b -> b
```

☐ `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`

---

2. 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:

```
--
> if x <= y z ... -- Deben ser comparable, por lo que tipo de x e y son Ord a.
> y z             -- Es una función de un argumento b y devuleve a. y::b->a
> z + 2           -- z debe ser numérico (y es arg de y), y::Num b->a, z::Num b.
-- Juntamos todo
> (Ord a, Num b) => a -> (b -> a) -> b -> b
```

☐ `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`

3. 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:

```
--
> if x        -- Debe un booleano, por lo que x::Bool.
> y <= z      -- Deben ser comparables, por lo que y, z::Bool, devulve Bool.
> x           -- Es un Bool, devuelve Bool
-- Juntamos todo
> (Ord a) => Bool -> a -> a -> Bool
```

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

---

4. 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:

```
--
> if x <= y   -- Deben ser comparables, x, y::Ord x
> z + 1       -- Debe ser numérico.
> x           -- También debe ser numérico (por z + 1) => y también lo es.
-- Juntamos todo
> (Ord a, Num a) => a -> a -> a -> a
```

- ☒ `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`

---

5. 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:

```
--
> if x <= y   -- Deben ser comparables, x, y::Ord x
> z           -- Puede ser cualquier tipo b
> not x       -- x debe ser un Bool => y::Bool, que Bool ya es Ord y z = Bool.
-- Juntamos todo
> f::Bool -> Bool -> Bool -> Bool
```

- ☐ `f :: (Ord a, Bool a) => a -> a -> a -> a`
- ☐ `f :: Ord a => Bool -> a -> a -> Bool`
- ☒ `Bool -> Bool -> Bool -> Bool`

6. 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:

```
> y + 1 -- Debe ser numérico.
> x <= y + 1 -- Deben ser numéricos y a su vez son comparables.
> z (..<=..) -- Función que acepta un Bool. y devuelve lo que sea )b).
-- Juntamos todo.
> f::(Num a, Ord a) => a -> a -> (Bool -> b) -> b
```

☒ `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

---

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 + x else z` será:

```
> z + x   -- Deben ser numéricos para poder sumarse (Es lo que devuelven).
> x <= y  -- x e y deben ser comparables y por el apartado anterior, numéricos.
-- Juntamos todo.
> f::(Num a, Ord a) => a -> a -> a -> a
```

☐ `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 = if x <= 0 then y + 1 else y` será:

```
> y + 1    -- y debe de ser numérico Num b. (Cualquier Numero).
> x <= 0   -- x debe de ser ordenable y numérico. Ord, Num::a.
-- Juntamos todo.
f::(Num b, Num a, Ord a) => a -> b -> b
```

☐ `f :: Num a => a -> a -> a`

☒ `f :: (Num a, Ord a, Num b) => a -> b -> b`

☐ `f :: (Ord a, Num a) => a -> a -> a`

9. 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:

```
> y + 1     -- y debe de ser numérico Num a.
> x == y+1 -- x e y deben de ser ordenable y numéricos Ord,Num::a
-- Juntamos todo.
f::(Num a, Ord a) => a -> a -> a
```

☐ `f :: Eq (Num a) => a -> a -> Num a`

☒ `f :: (Eq a, Num a) => a -> a -> a`

☐ `f :: (Eq a, Num b) => a -> b -> b`

10. 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:

```
> not x   -- x debe ser Bool.
> z <= y -- z e y deben ser Ordeables.
-- Juntamos todo.
f::(Ord a) => Bool -> a -> a -> Bool
```

- ☐ `f :: Ord Bool => Bool -> Bool -> Bool -> Bool`
- ☐ `f :: Bool -> Bool -> Bool -> Bool`
- ☒ `f :: Ord a => Bool -> a -> a -> Bool`

---

11. ¿Cuál de los siguientes tipos para `f` hacen que la expresión `(curry f 0) . (|| True)` esté bien tipada?

```
-- Supongamos que lo aplicamos a un X
> ((curry f 0) . (|| True)) X = (curry f 0 ((|| True) X))
> (curry f 0 ((|| True) X)) = (curry f 0 (X || True))
> f (0, X || True)
> f::(Int, Bool) -> a
```

- ☒ `f::(Int,Bool) -> Int`
- ☐ `f:: Int -> Bool -> Int`
- ☒ Esa expresión está mal tipada, sea cual sea el tipo de `f`.

12. ¿Cuál de los siguientes tipos para `f` hacen que la expresión `(|| True).(uncurry f)` esté bien tipada?

```
-- Supongamos que lo aplicamos a un X
> ((|| True).(uncurry f)) X = (|| True)(uncurry f X)
> (uncurry f X) || True
> f: X -> Bool y X = (a, b)
> f::(a, b) -> Bool -- Si a, b = Int.
```

- ☒ `f::(Int,Int)-> Bool`
- ☐ `f:: Int -> Bool -> Int`
- ☐ Esa expresión está mal tipada, sea cual sea el tipo de `f`.

Test *Tipo de datos en Métodos Prolog.*

NOTAS.

---

1. Sea `f` de tipo `t -> t` , y `unaLista` de tipo `[t]`. El tipo de la expresión `map (take 2) (map (iterate f) unaLista)` es:

```
> Supongamos que unaLista = [a1, .., an]
> map (iterate f) unaLista => [iterate f a1, ..., iterate f an] = ls
> map (take 2) ls => [take 2 (iterate f a1), ..., take 2 (iterate f an)]
-- Sabemos que iterate es un vector [a1, ..., f(..f(a1)..)]
-- Y Por la evaluación Perezosa es equivalente
> [[a1, f a1], ... [an, f an]]
```

☐ `[t]`

☒ `[[t]]`

☐ Esa expresión está mal tipada.

---

2. Sea `f` de tipo `t -> t` , y `unaLista` de tipo `[t]`. El tipo de la expresión `map (iterate f) (map (take 2) unaLista)` es:

```
> Supongamos que unaLista = [a1, .., an]
> map (take 2) unaLista = [take 2 a1, .., take 2 an]
-- Vemos que da un error si ai != [a11, .., a1k]
> [take 2 a1, .., take 2 an] = [[a11, a12], .., [ak1, akn]] = ls
> map (iterate f) ls = [iterate f [a11, a12], .., iterate f [ak1, akn]]
```

☐ `[t]`

☒ `[[t]]`, si es que t es de la forma `[t']`

☐ Esa expresión está en cualquier caso mal tipada.

---

3. ¿Cuál de los siguientes tipos para la expresión `e` hace que la expresión `zipWith filter [(> 0),(< 0)] e` esté bien tipada?

```
-- Supongamos que e es una lista, e = [a1, .., an]
> zipWith filter [(> 0),(< 0)] e = [filter (> 0) a1, filter (< 0) a2]
-- Vemos que filter actua con una función y una lista => ai = [ai1, ..., aik]
> [filter (> 0) [a11, ..., a1k], filter (< 0) [a21, a2k]]
```

☐ `[Int]`

☒ `[[Int]]`

☐ `[(Int, Int)]`

4. ¿Cuál de los siguientes tipos para la expresión `e` hace que la expresión `zipWith filter e [[1..4],[-2..3]]` esté bien tipada?

```
-- Sabemos que e tiene que ser una lista, e = [e1,..,en]
> zipWith filter e [[1..4],[-2..3]] = [filter e1 [1..4], filter e2 [-2..3]]
-- Vemos que e1,2::Int->Bool
```

☐ `Int -> Bool`

☒ `[Int -> Bool]`

☐ Las dos anteriores son falsas

---

5. ¿Cuál de los siguientes tipos para la expresión `e` hace que la expresión `takeWhile e zip (iterate not True) [0..10]` esté bien tipada?

```
> iterate not True = [True, False, True, ...] = tf
> zip tf [0..10] = [(True, 0), (False, 1), ... (True, 10)] = zp
-- takeWhile filtra los n primeros de zp hasta encontrar uno que no cumpla.
> e::(Bool, Int) -> Bool
```

☐ `Int -> Int`

☐ `Int -> Bool`

☒ `(Bool,Int) -> Bool`

---

6. ¿Cuál de los siguientes tipos para la expresión `e` hace que la expresión `zipWith e (iterate not True) (iterate (+ 1) 0)` esté bien tipada?

```
> iterate not True = [True, False, True, ...] = tf -- Tipo Bool
> iterate (+ 1) 0 = [0, 1, 2, ...] = na            -- Tipo Int
> zipWith e tf na = [e tf1 na1, e tf2 na2, ...]
-- Vemos que
e::Bool -> Int -> a
-- Podemos sustituir a = Char como la 3a opción.
```

☐ `[Bool] -> [Int] -> [Bool]`

☐ `[Bool] -> [Int] -> [(Bool,Int)]`

☒ `Bool -> Int -> Char`

8. ¿Cuál de los siguientes tipos para la expresión `e` hace que la expresión `(head.e) (zip (iterate not True) [1..5])` esté bien tipada?

```
> iterate not True = [True, False, True, ...] = tf -- Tipo Bool
> zip (iterate not True) [1..5] = [(True, 1), ..., (False, 5)] = zp
> (head.e) zp = head (e zp)
-- Vemos que (e zp) debe devolver un vector para compilar.
> [(Bool, Int)] -> [a]
-- Podemos cambiuar a por Int.
```

- ☒ `[(Bool,Int)] -> [Int]`
- ☐ `(Bool,Int) -> Int`
- ☐ `(Bool,Int) -> [Int]`

Test *Definiciones de Tipos.*

NOTAS.

`data <constructor> <template> = <nombre> [<tb1> <tbn>] | ...`

1. `Template` es un dato básico (`Int`, `Bool`, `Integral`...).
2. Donde tbi son tipos de datos:
    1. `Datos Básicos` (`Int`, `Bool`, `Integral`...).
    2. `[Datos Básicos]` (Un **vector de Tipos**)
    3. El valor del `template`
    4. `(<tbi>, .., <tbk>)`.
        1. Puede ser uno de los anteriores.
        2. Puede ser el propio `<constructor>` (Con argumento si lo tiene).

---

1. ¿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
```

```
1> Tip::(Int,Int,Tip) -- Falta nombre => Mal.
2> Tap                -- Bien.
3> Top::a,Top::b      -- ¿a, b? => Mal.
```

☐ Las tres

☐ Ninguna de las tres

☒ Una de las tres

---

2. ¿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
```

```
1> Tip::C       -- Declarada dos veces.
2> Tap          -- Bien.
3> Top          -- Bien.
```

☐ Una de las tres

☒ Dos de las tres

☐ Las tres

Comparar los tipos que derivan la clase Eq o Ord, argumentos lexicográficamente.

3. Considérese la definición del tipo
   `data T = A | B | C T T deriving (Eq,Ord)`.
   ¿Cuál de las siguientes afirmaciones es cierta?

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
1> A <= B && B <= C A A
-- Cambiamos por los números.
1> (1 <= 2 && 2 <= 3) = True && True = True
```

☒ `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`.

---

4. 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?

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
mal = head[]
--    undefined (Bottom _|_)
-- Cambiamos por los números.
1> C mal A <= C mal B
1> 1 _|_ 1 <= 1 _|_ 2 -- Error, _|_ <= _|_
2> C A mal == C B mal
2> 3 1 _|_ == 3 2 _|_ -- False (Evaluación perezosa)
```

☐ `C mal A <= C mal B` se evalúa a `True`

☒ `C A mal == C B mal` se evalúa a `False`

☐ `A <= C mal mal && B <= C mal mal` se evalúa a `False`.

5. 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?

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
loop = loop -- error (Bottom _|_)
-- Cambiamos por los números.
1> A <= B && B <= C loop loop
1> 1 <= 2 && 2 <= 3 _|_  _|_ -- True (Evaluación perezosa)
```

- ☒ `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`.

---

6. 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**?

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
loop = loop -- error (Bottom _|_)
-- Cambiamos por los números.
1> A <= C loop loop && B <= C loop loop
1> 1 <= 3 _|_  _|_  && 2 <= 3 _|_  _|_  -- Cierto
2> C A loop == C B loop
2> 3 2 _|_  == 3 1 _|_  -- 3==3,2!=1 => False -- Cierto
```

- ☒ `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`.

7. 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**?

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
mal =  head [] -- error (Bottom _|_)
1> C mal A <= C mal B
1> 3 _|_ 2 <= C _|_ B                -- Mal, _|_=_|_
2> A <= C mal mal && B <= C mal mal
2> 1 <= 3 _|_ _|_ && 2 <= 3 _|_ _|_ -- True, Evaluación Perezosa
```

☐ `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`.

---

8. 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`

```
data T = A | B | C T T deriving (Eq,Ord)
--       1 | 2 | 3
mal =  head [] -- error (Bottom _|_)
i> C mal A <= C mal B
i> 3 _|_ 1 <= 3 _|_ 2                 -- Error _|_ = _|_
ii> C A mal == C B mal
ii> 3 1 _|_ == 3 2 _|_                -- False 2 != 1
iii> A <= C mal mal && B <= C mal mal
iii> 1 <= 3 _|_ _|_ && 2 <= 3 _|_ _|_ -- Bien
```

☒ Exactamente una es cierta

☐ Exactamente dos son ciertas

☐ Las dos anteriores son falsas.

Test *Fold*.

NOTAS.

1. `foldr o A [a1,.., an] = o a1 (... (o an1 (o an A)...))`
2. `foldl o A [a1,.., an] = o (o... (o (o A a1) a2) ... ) an`

---

1. Considérese la función `f` definida como `f xs = foldr g [] xs where g x y = y++[x]`. Entonces:

```
> g x y = y++[x] = \x y -> y ++ [x]
> f xs = foldr g [] xs
-- Es más facil con un ejemplo
> xs = [2, 3]

> ((\x y -> y ++ [x]) 2 ((\x y -> y ++ [x]) (3 [])))
> ((\x y -> y ++ [x]) 2 ([] ++ [3]))
> (([] ++ [3]) ++ [2]) = [3, 2]
```

☒ `f xs` computa la inversa de `xs`

☐ `f xs` computa la propia lista `xs`

☐ `f` está mal tipada

---

2. Considérense las funciones

$$\begin{cases} \texttt{f xs = foldr g [] xs where g x y = x:filter (/= x) y} \\ \texttt{f' xs = foldl g [] xs where g y x = x:filter (/= x) y} \end{cases}$$

```
> g1 x y = x:filter (/= x) y = (\x y -> x:filter (/= x) y)
> g2 x y = x:filter (/= x) y = (\y x -> x:filter (/= x) y)
> xs = [a1,..., an]

> f xs = (...(\x y -> x:filter (/= x) y) an [])
> f xs = (...(an:filter (/= an) []))
> f xs = (...g1 a(n-1) (an:filter (/= an) []))

> f' xs = ((\y x -> x:filter (/= x) y) [] a1) ...) ...
> f' xs = ((a1:filter (/= a1) []) ...) ...
-- Vemos que ambos tienen los mismos elementos pero desordenados.
```

☒ `f xs` y `f' xs` coinciden, para cualquier lista finita `xs`.

☐ Los elementos de `f xs` y `f' xs` coinciden, quizás en otro orden, para cualquier lista finita `xs`.

☐ Una de las dos está mal tipada.

3. Considérese la función `f` definida como `f xs = foldl g [] xs where g y x = y++[x]`. Entonces:

```
> xs = [a1, a2, ..., an]
> g y x = y++[x] = (\x y -> x++[y])
> f xs = foldl g [] xs
> f xs = ((\x y -> x++[y]) [] a1) ... = (..(g ([]++[a1]) a2)...)
> f xs = ((\x y -> x++[y]) ([]++[a1]) a2) ... = (g (([]++[a1])++[a2])) ...
-- Vemos que va [] ++ [a1] ++ ... ++ [an] = xs
```

☐ `f xs` computa la inversa de `xs`

☒ `f xs` computa la propia lista `xs`

☐ `f` está mal tipada

---

4. La evaluación de `foldl (\e x -> x:x:e) [] [1,2,3]` produce como resultado

```
> (((\e x -> x:x:e) [] 1) ... = ((\e x -> x:x:e) (1:1:[]) 2)...
> ((\e x -> x:x:e) (2:2:(1:1:[])) 3)... = 3:3:(2:2:(1:1:[]))
```

☐ `[1,1,2,2,3,3]`

☒ `[3,3,2,2,1,1]`

☐ `[3,2,1,3,2,1]`

---

5. La evaluación de `foldr (\x e -> x:[1..length e]) [0] [1,2,3]` produce como resultado

```
> (..(\x e -> x:[1..length e]) 3 [0]) =
> (..(\x e -> x:[1..length e]) 2 (3:[1..1])) =
> (..(\x e -> x:[1..length e]) 2 [3, 1])
> (..(\x e -> x:[1..length e]) 1 (2:[1..2]))
> (..(\x e -> x:[1..length e]) 1 [2, 1, 2])
> 1:[1..3] = [1, 1, 2, 3]
```

☐ `[1,2,3,0]`

☐ `[3,1,2,3]`

☒ `[1,1,2,3]`

6. La evaluación de `foldr (\x y -> x y) 1 [\x -> x*x,\x -> x-1,(+ 3)]` produce como resultado

```
> (...(\x y -> x y) (+ 3)  1) =
> (...(\x y -> x y) (\x -> x-1) ((+3) 1))) =
> (...(\x y -> x y) (\x -> x*x) ((\x -> x-1) 4))
> (\x -> x*x) ((\x -> x-1) 4) = (\x -> x*x) 3) = 3 * 3 = 9
```

☒ `9`

☐ `[1,0,4]`

☐ Una lista de funciones

Test *Reducción de expresiones.*

NOTAS.

---

1. La reducción de la expresión `(\x y -> (\z -> y (z+2)) (y x)) 3 (\x -> x+1)` producirá el resultado:

```
> y1 = (\x -> x+1)
> x1 = 3
> (\x1 y1 -> (\z -> y1 (z+2)) (y1 x1)) =
> (\z -> (\x -> x+1) (z+2)) ((\x -> x+1) 3)
> (\z -> (z+2) + 1) 4 = (4+2) + 1 = 7
```

☐ 8

☒ 7

☐ 6

---

2. La reducción de la expresión `(\x y -> x (x y)) (\x -> x + 3) 4` producirá el resultado:

```
> y1 = 4
> x1 = (\x -> x + 3)
> (\x1 y1 -> x1 (x1 y1)) =
> (\x -> x + 3) ((\x -> x + 3) 4) = (\x -> x + 3) (4 + 3)
> (\x -> x + 3) (4 + 3) = (4 + 3) + 3 = 10
```

☐ 7

☒ 10

☐ Las dos anteriores son falsas.

---

3. La reducción de la expresión `(\x y -> x (x y)) (\x -> x + y) 4` producirá el resultado:

☒ 8

☐ 12

☐ Las dos anteriores son falsas.