1. Dadas las expresiones: `[True,[]]` `True:[]` `[True]:[]` `[[True],[]]`
   - (x) 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],[]]`
   - ( ) Dos de ellas están mal tipadas
   - (x) Dos de ellas son sintácticmente equivalentes y una está mal tipada
   - ( ) Las dos anteriores son falsas.

Nota: en las preguntas siguientes, sobre sintaxis de listas, suponemos que los numerales `1,2,...` **tienen un tipo concreto, por ejemplo** `Int`**.**

3. Dadas las expresiones: `0:[1]` `0:[1]:[2]` `0:[1,2]` `[0,1]:[[2]]` `([]:[],2)`
   - (x) Exactamente una de las expresiones está mal tipada
   - ( ) Exactamente dos de las expresiones están mal tipadas
   - ( ) Las dos anteriores son falsas.

4. Dadas las expresiones: `[]:[1]` `[1:[2]]:[]` `[1:[2]]:[[]]` `[1,1]:(2:[])` `(1:[]):[]`
   - ( ) Exactamente tres de las expresiones están mal tipadas
   - (x) 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:[]):[]`
   - ( ) Exactamente tres de las expresiones están mal tipadas
   - (x) Exactamente dos de las expresiones están mal tipadas
   - ( ) Las dos anteriores son falsas.

6. Dadas las expresiones: `[1]:[]` `[[]]:[]` `[]:[]` `(1:2):[]` `1:(2:[])`
   - (x) Exactamente una de ellas está mal tipada
   - ( ) Exactamente dos de ellas están mal tipadas
   - ( ) Las dos anteriores son falsas.

7. Dadas las expresiones: `[0]:[1]` `[]:[[]]:[]` `[0]:[[]]:[]` `[0]:[[1,2]]` `([[]]:[],[1])`
   - ( ) Exactamente una de las expresiones está mal tipada
   - (x) Exactamente dos de las expresiones están mal tipadas
   - ( ) Exactamente tres de las expresiones están mal tipadas

8. Dadas las expresiones: `[1]:[]` `[1]:[[]]:[2]` `[1]:[[[]]]` `0:1:2` `(0:[1],2)`
   - ( ) Exactamente una de las expresiones está mal tipada
   - ( ) Exactamente dos de las expresiones están mal tipadas
   - (x) Las dos anteriores son falsas.

9. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `[[0],[],[2,2]]`?
   - (x) `((0:[]):[[],2:2:[]]):[]`
   - ( ) `[0]:[]:[2,2]:[]:[]` = [[0],[],[2,2],[]]
   - ( ) Ninguna de las anteriores, porque de hecho la expresión está mal tipada

([0]:[[],[2,2]]) : []

[[[0],[],[2,2]]]

9/9

10. ¿Cuál de las siguientes expresiones es sintácticamente equivalente a `[[1,[]],[3,4]]`?
   - ◯ `((1:[]):[[],3:4:[]]):[]`
   - ◯ `[1]:[]:[3,4]:[]:[]`
   - ☒ 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:([],[])`
   - ☒ `(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],[]]`
   - ◯ Ninguna de las anteriores, porque de hecho la expresión está mal tipada

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

   `[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]]`   `[1]:[2,[]]` ✗   `[1,2]:[]`

   ¿Cuál de las siguientes afirmaciones es cierta?
   - ◯ 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]]`   `([1]:[[2]]):[]`   `(1:[2]):[]`   `[1,2]:[]`   `[1]:[2]:[]`

   ¿Cuál de las siguientes afirmaciones es cierta?
   - ☒ 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. ¿Cúantas 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]]
   - ◯ Exactamente dos
   - ◯ Exactamente tres
   - ☒ Exactamente cuatro

   [[1,2],[1]]   [[1,2],[1]]   [[1,2],[1]]

17. 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`

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

18. Considérense las expresiones (que 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`

- ◯ $e_1 \equiv e_2 \equiv e_3$
- ⊗ $e_1 \equiv e_2 \not\equiv e_3$
- ◯ $e_1 \not\equiv e_2 \not\equiv e_3$

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

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

Entonces:

- ◯ $e_1 \equiv e_2 \equiv e_3$
- ⊗ $e_1 \equiv e_2 \not\equiv e_3$
- ◯ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

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

$e_1 =$ `f x (g x,y/2)`
$e_2 =$ `(f x (g x)) (y/2)`
$e_3 =$ `(f x) (g x,(/y) 2)`

(g x, 2/y)

- ◯ $e_1 \equiv e_2 \equiv e_3$
- ⊗ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$
- ◯ $e_1 \equiv e_3 \not\equiv e_2$

21. Considérense las expresiones (que 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`

- ⊗ $e_1 \equiv e_2 \not\equiv e_3$
- ◯ $e_1 \not\equiv e_2 \equiv e_3$
- ◯ $e_1 \equiv e_2 \equiv e_3$ ✗

22. Considérense las expresiones (que 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)`

y+1

- ◯ $e_1 \equiv e_2 \equiv e_3$
- ◯ $e_1 \not\equiv e_2 \not\equiv e_3$
- ⊗ $e_1 \equiv e_3 \not\equiv e_2$

23. Considérense las expresiones (que 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`

(x+1)

- ◯ $e_1 \equiv e_2 \not\equiv e_3$
- ⊗ $e_1 \equiv e_3 \not\equiv e_2$
- ◯ Las dos anteriores son falsas.

24. Considérense las expresiones (que 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)`

x+y

- ◯ $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$

- $e_1 \equiv e_3 \not\equiv e_2$
- $e_1 \equiv e_2 \equiv e_3$

25. Suponiendo la declaración `infixr 9 !`, considérense las expresiones (que solo difieren en los paréntesis):
    $e_1 =$ `((! g) f) ! ((h !) i) ! j`
    $e_2 =$ `((!) (f ! g)) ((h ! i) ! j)`
    $e_3 =$ `(!) ((!) f g) ((!) ((!) h i) j)`
    - $e_1 \equiv e_2 \equiv e_3$
    - $e_1 \equiv e_2 \not\equiv e_3$
    - $e_1 \not\equiv e_2 \equiv e_3$

26. Considérense las expresiones: $e_1 =$ `f x (y-1):z`
    $e_2 =$ `((:) f) x ((-) y 1) z`
    $e_3 =$ `(: z) ((f x) ((-) y 1))`
    - $e_1 \equiv e_2 \equiv e_3$
    - $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$
    - $e_1 \equiv e_3 \not\equiv e_2$

MAL

$e_1 \equiv e_3$

27. Considérense las expresiones (que solo difieren en los paréntesis): $e_1 =$ `f ((g x) (x y)) x`
    $e_2 =$ `(f (g x) (x y)) x`
    $e_3 =$ `(f (g x (x y))) x`
    - $e_1 \equiv e_2 \not\equiv e_3$
    - $e_1 \equiv e_3 \not\equiv e_2$
    - $e_1 \equiv e_2 \equiv e_3$

28. Considérense las expresiones (que 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)`
    - $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$
    - $e_1 \equiv e_2 \equiv e_3$
    - $e_1 \equiv e_3 \not\equiv e_2$

MAL

POTRA

29. Considérense las expresiones (que 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))`
    - $e_1 \not\equiv e_2 \not\equiv e_3 \not\equiv e_1$
    - $e_1 \equiv e_2 \equiv e_3$
    - $e_1 \equiv e_3 \not\equiv e_2$

30. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `(a -> a) -> (a -> a) -> (a -> a)`
    $\tau_2 =$ `(a -> a) -> a -> a -> a -> a`
    $\tau_3 =$ `(a -> a) -> (a -> a) -> a -> a`
    - $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
    - $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
    - $\tau_1 \equiv \tau_2 \equiv \tau_3$

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

$\tau_1 =$ `((b -> a) -> a) -> ((a -> b) -> (b -> b))`
$\tau_2 =$ `(b -> (a -> a)) -> (a -> b) -> b -> b`
$\tau_3 =$ `(b -> a -> a) -> (a -> b -> b -> b)`

Entonces:

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \equiv \tau_2 \equiv \tau_3$
- ◯ $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$

32. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `a -> (b -> a) -> a -> b -> b`
$\tau_2 =$ `(a -> ((b -> a) -> a)) -> (b -> b)`
$\tau_3 =$ `a -> b -> a -> a -> b -> b`

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$
- ◯ $\tau_1 \equiv \tau_3 \not\equiv \tau_2$

33. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `(a -> b -> a) -> a -> b -> b`
$\tau_2 =$ `(a -> b -> a) -> a -> (b -> b)`
$\tau_3 =$ `(a -> (b -> a)) -> a -> b -> b`

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- ◯ $\tau_1 \equiv \tau_2 \equiv \tau_3$

34. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `a -> (b -> a -> a) -> b -> b`
$\tau_2 =$ `(a -> ((b -> a) -> a)) -> b -> b`
$\tau_3 =$ `a -> (b -> (a -> a)) -> (b -> b)`

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- ◯ $\tau_1 \equiv \tau_2 \equiv \tau_3$

35. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `(a -> b -> a -> a) -> b -> b`
$\tau_2 =$ `a -> b -> a -> a -> b -> b`
$\tau_3 =$ `(a -> b -> (a -> a)) -> (b -> b)`

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- ◯ $\tau_1 \equiv \tau_2 \equiv \tau_3$

36. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `a -> a -> (a -> a) -> (b -> b)`
$\tau_2 =$ `a -> (a -> ((a -> a) -> b -> b))`
$\tau_3 =$ `(a -> a) -> (a -> a) -> b -> b`

- ◯ $\tau_1 \equiv \tau_2 \equiv \tau_3$
- ◯ $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$
- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$

37. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `a -> ((a -> a) -> a)`
$\tau_2 =$ `a -> (a -> a) -> a`
$\tau_3 =$ `a -> a -> a -> a`

- ◯ $\tau_1 \equiv \tau_2 \not\equiv \tau_3$
- ◯ $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- ◯ $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$

38. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `(a -> a) -> (a -> a) -> (a -> a)`, $\tau_2 =$ `a -> a -> (a -> a) -> (a -> a)`, $\tau_3 =$ `(a -> a) -> (a -> a) -> a -> a`

- $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$
- $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- $\tau_1 \equiv \tau_2 \equiv \tau_3$

39. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `(b -> a -> a) -> (a -> b) -> b`, $\tau_2 =$ `(b -> (a -> a)) -> ((a -> b) -> b)`, $\tau_3 =$ `(b -> a -> a) -> a -> b -> b`

- $\tau_1 \equiv \tau_2 \equiv \tau_3$
- $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- $\tau_1 \equiv \tau_2 \not\equiv \tau_3$

40. Considérense las expresiones de tipo (solo difieren en los paréntesis): $\tau_1 =$ `a -> b -> (c -> d)`, $\tau_2 =$ `a -> (b -> c -> d)`, $\tau_3 =$ `a -> b -> c -> d`

- $\tau_1 \equiv \tau_2 \equiv \tau_3$
- $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3$

41. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `(a -> a) -> (a -> a) -> (a -> a)`, $\tau_2 =$ `(a -> a) -> (a -> a) -> a -> a`, $\tau_3 =$ `(a -> a) -> ((a -> a) -> (a -> a))`

- $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$
- $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- $\tau_1 \equiv \tau_2 \equiv \tau_3$

42. Considérense las expresiones de tipo (que solo difieren en los paréntesis): $\tau_1 =$ `(a -> a -> a) -> (a -> a) -> (a -> a)`, $\tau_2 =$ `(a -> (a -> a)) -> (a -> a) -> a -> a`, $\tau_3 =$ `(a -> a -> a) -> ((a -> a) -> a -> a)`

- $\tau_1 \not\equiv \tau_2 \not\equiv \tau_3 \not\equiv \tau_1$
- $\tau_1 \equiv \tau_3 \not\equiv \tau_2$
- $\tau_1 \equiv \tau_2 \equiv \tau_3$

43. Considérense las expresiones siguientes:

$e_1 \equiv$ `(let x=5 in x+x) + 3` $\quad e_2 \equiv$ `let x=2 in let y=x+x in y*y`
$e_3 \equiv$ `let x=2 in let y=x+x in y*y*x` $\quad e_4 \equiv$ `let y=x+x in let x=2 in y*y*x`
$e_5 \equiv$ `let y=(let x=2 in x+x) in y*y` $\quad e_6 \equiv$ `let y=(let x=2 in x+x) in y*y*x`

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

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

44. Considérense las expresiones siguientes:

$e_1 \equiv$ `(let x=5 in x+x) + 5` $\quad e_2 \equiv$ `let x=2 in let y=x+x in y*x`
$e_3 \equiv$ `let x=2 in let y=x+x in x` $\quad e_4 \equiv$ `let x=y in let x=2 in y*y*x`
$e_5 \equiv$ `let y=(let x=2 in x+x) in y*y` $\quad e_6 \equiv$ `let y=(let x=2 in x+x) in y*y*x`

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

- ○ Exactamente una de ellas
- ○ Exactamente dos de ellas
- ○ Exactamente tres de ellas

45. Considérense las expresiones siguientes:
    - $e_1 \equiv$ `(let x=5 in x+x) + x`
    - $e_2 \equiv$ `let x=2 in let y=x+x in y*y*x`
    - $e_3 \equiv$ `let y=x+x in let x=2 in y*y*x`
    - $e_4 \equiv$ `let {y=x+x;x=2} in y*y*x`
    - $e_5 \equiv$ `let y=(let x=2 in x+x) in y*y*x`
    - $e_6 \equiv$ `let y=(let x=2 in 3) in y*y`

    ¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?
    - ○ Exactamente dos de ellas
    - ○ Exactamente tres de ellas
    - ○ Exactamente cuatro de ellas

46. Considérense las expresiones siguientes:
    - $e_1 \equiv$ `(let x=5 in x+x) + 5`
    - $e_2 \equiv$ `let x=2 in let y=x+x in y*x`
    - $e_3 \equiv$ `let x=2 in let y=x+x in x`
    - $e_4 \equiv$ `let x=y in let x=2 in y*y*x`
    - $e_5 \equiv$ `let y=(let x=x in x+x) in y*y`
    - $e_6 \equiv$ `let y=(let x=2 in x+x) in y*y*x`

    MAL

    ¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?
    - ○ Exactamente una de ellas
    - ○ Exactamente dos de ellas
    - ○ Exactamente tres de ellas

47. Considérense las expresiones siguientes:
    - $e_1 \equiv$ `(let x=5 in x+x) + (let x=3 in 2*x)`
    - $e_2 \equiv$ `let y=x+x in let x=2 in y*y*x`
    - $e_3 \equiv$ `let x=2 in let y=x+x in y*y*x`
    - $e_4 \equiv$ `let {y=x+x;x=2} in y*y*x`
    - $e_5 \equiv$ `[i | i<-[1..j],j<-[0..100],mod j 3 == 0]`
    - $e_6 \equiv$ `[i | j<-[0..100],i<-[1..j],mod j 3 == 0]`

    ¿Cuántas de ellas son sintácticamente erróneas por problemas de ámbito de variables?
    - ○ Exactamente una de ellas
    - ○ Exactamente dos de ellas
    - ○ Exactamente tres de ellas

48. Considérense las expresiones siguientes:
    - $e_1 \equiv$ `let x=1:x in head x`
    - $e_2 \equiv$ `(\x -> (\y -> x+y)) x`
    - $e_3 \equiv$ `let x=[1,2,3] in let y= x!!2 in y*last x`
    - $e_4 \equiv$ `let {y=2*x;x=5} in y*y*x`
    - $e_5 \equiv$ `[i+j | i<-[1..j],j<-[0..100],mod j i == 0]`

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

    MAL
    - ○ Exactamente dos de ellas
    - ○ Exactamente en tres de ellas
    - ○ Exactamente en cuatro de ellas

49. Considérense las expresiones siguientes:
    - $e_1 \equiv$ `\x -> ((\y -> x) x)`
    - $e_2 \equiv$ `\x -> ((\y -> x+y) y)`
    - $e_3 \equiv$ `let y=[1,2,3] in let x= y!!1 in x*head y`
    - $e_4 \equiv$ `let {y=2*x;x=5} in y*y*x`
    - $e_5 \equiv$ `[i+j | i<-[1..100],j<-[0..i],mod j i == 0]`

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

- ○ Exactamente una de ellas
- ○ Tres o más de ellas
- ○ Las dos anteriores son falsas.

50. Considérense las expresiones siguientes:

$e_1 \equiv$ `\x -> ((\x y -> x+y) x y)` $e_2 \equiv$ `\x -> ((\x y -> x+y) x x)`

$e_3 \equiv$ `let y= (let x = 1 in x+x) in x+y` $e_4 \equiv$ `let y= (let x = 1 in x+x) in y+y`

$e_5 \equiv$ `[j | i<-[1..100],j<-[0..i]]`

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

- ○ Exactamente una de ellas
- ○ Exactamente dos de ellas
- ○ Las dos anteriores son falsas.

51. 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.

52. 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
```

Mal

- ○ 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.

53. En el siguiente fragmento de código

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

- ○ La definición del tipo `T` contiene algún error sintáctico, pero la de `f` no.
- ○ La definición de la función `f` contiene algún error sintáctico, pero la de `T` no.
- ○ Las dos anteriores son falsas.

54. 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)
```

- ○ 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.

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

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

56. Sea `f` definida por `f g x = x (g True) g`. El tipo de `f` es:
   - ◯ `∀a,b.(Bool->a) -> (a -> (Bool->a) -> b) -> b`
   - ◯ `∀a.(Bool->a) -> (a -> (Bool->a) -> a) -> a`
   - ◯ Está mal tipada

57. Sea `f` definida por `f g x = x g g`. El tipo de `f` es:
   - ◯ `∀a,b.a -> (a -> a -> b) -> b`
   - ◯ `∀a,b.(a -> a -> b) -> a -> b`
   - ◯ Está mal tipada

58. Sea `f` definida por `f g x = x g g`. El tipo de `f` es:
   - ◯ Está mal tipada
   - ◯ `∀a.(a -> a -> a) -> a -> a`
   - ◯ `∀a,b.a -> (a -> a -> b) -> b`

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

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

61. 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

62. 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

63. Considérese la función definida por `f g = g (f g)` . El tipo de `f` es:
   - ◯ `a -> a -> a`
   - ◯ `(a -> a) -> a`
   - ◯ No está bien tipada

64. Considérese la función definida por `f g = g (g f)` . El tipo de `f` es:
   - ◯ `a -> a -> a`
   - ◯ `(a -> a) -> a`
   - ◯ No está bien tipada

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

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

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

- [ ] `∀a.a -> (a -> a -> a) -> a`
- [ ] `∀a,b.a -> (b -> a -> b) -> b`
- [x] Está mal tipada

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

- [ ] `(a -> b) -> (b -> c) -> a -> c`
- [x] `(a -> b) -> (c -> a) -> c -> b`
- [ ] `f` está mal tipada

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

- [ ] `∀a,b.(a -> b) -> (Bool -> a) -> b`
- [x] `∀a.(a -> a) -> (Bool -> a) -> a`
- [ ] Ninguno, `f` está mal tipada

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

- [ ] `(a -> a) -> (a -> a)`
- [ ] `(a -> b) -> (a -> b) -> (a -> b)`
- [x] `(a -> b -> b) -> a -> b -> b`

COMPOSICION

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

- [x] `(a -> a) -> (a -> a)`
- [ ] `(a -> b) -> a -> b`
- [ ] `a -> b -> a`

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

- [ ] `a -> a -> a`
- [ ] `(a -> a) -> a`
- [x] No está bien tipada

72. 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`
- [x] `f :: (Ord a, Num b) => a -> a -> b -> b`
- [ ] `f :: (Ord a, Num a) => a -> a -> a -> a`

73. 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`
- [x] `f :: (Ord a, Num b) => a -> (b -> a) -> b -> b`

z

74. 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`
- [x] `f :: Ord a => Bool -> a -> a -> Bool`
- [ ] Esa definición dará un error de tipos

75. 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`

76. 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`

77. 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

78. 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`

79. 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`

80. 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`

81. 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`

82. ¿Cuál de los siguientes tipos para `f` hacen que la expresión `(curry f 0).(|| True)` esté bien tipada?
- ☒ `f::(Int,Bool) -> Int`
- ◯ `f:: Int -> Bool -> Int`
- ◯ Esa expresión está mal tipada, sea cual sea el tipo de `f`

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

- [ ] [τ]
- [x] [[τ]]
- [ ] Esa expresión está mal tipada

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

[x1 .. xn] τ = [Int] [[1], 2

[[[1], f[1], ...], ...] [[1],[2]]

- [ ] [τ]
- [x] [[τ]], si es que τ es de la forma [τ']
- [ ] Esa expresión está en cualquier caso mal tipada

85. ¿Cuál de los siguientes tipos para `f` hace que la expresión `(|| True).(uncurry f)` esté bien tipada?
- [ ] `f::(Int,Int)-> Bool`
- [x] `f:: Int -> Int -> Bool`
- [ ] Esa expresión está mal tipada, sea cual sea el tipo de `f`

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

87. ¿Cuál de los siguientes tipos para la expresión e hace que la expresión `zipWith filter` e `[[1..4],[-2..3]]` esté bien tipada?
- [ ] `Int -> Bool`
- [x] `[Int -> Bool]`
- [ ] Las dos anteriores son falsas.

88. ¿Cuál de los siguientes tipos para la expresión e hace que la expresión `takeWhile` e `(zip (iterate not True) [0..10])` esté bien tipada?
- [ ] `Int -> Int`
- [ ] `Int -> Bool`
- [x] `(Bool,Int) -> Bool`

89. ¿Cuál de los siguientes tipos para la expresión e hace que la expresión `zipWith` e `(iterate not True) (iterate (+ 1) 0)` esté bien tipada?
- [ ] `[Bool] -> [Int] -> [Bool]`
- [ ] `[Bool] -> [Int] -> [(Bool,Int)]`
- [x] `Bool -> Int -> Char`

zipWith e [T,F,T...] [0,1 ...]

90. ¿Cuál de los siguientes tipos para la expresión e hace que la expresión `(head.`e`) (zip (iterate not True) [1..5])` esté bien tipada?
- [x] `[(Bool,Int)] -> [Int]`
- [ ] `(Bool,Int) -> Int`
- [ ] `(Bool,Int) -> [Int]`

head.e [(T,1), (F,2) . ...]

91. ¿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

92. ¿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
```

MAL

- ○ Una de las tres
- ● Dos de las tres
- ○ Las tres

93. 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) $[\,] \leq [True] \leq [False, True]$ (ii) $[\,] \leq [False, True] \leq [True, False]$. Entonces, teniendo en cuenta que para los booleanos False $\leq$ True, se tiene:

Repasar orden

- ○ (i) es cierta pero (ii) no
- ⊗ (ii) es cierta pero (i) no
- ○ Las dos son falsas

94. 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` ??

95. 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`

96. 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`

97. 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` F
- ○ `A <= C loop loop && B <= C loop loop` se evalúa a `True` V
- ○ `C A loop == C B loop` se evalúa a `False`

98. 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` F
- ⊗ `A <= C mal mal && B <= C mal mal` se evalúa a `True`
- ○ `C A mal == C B mal` se evalúa a `True` F

99. 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`	F
(ii)	`C A mal == C B mal`	se evalúa a `True`	F
(iii)	`A <= C mal mal && B <= C mal mal`	se evalúa a `True`	V

- ⊗ Exactamente una es cierta
- ○ Exactamente dos son ciertas
- ○ Las dos anteriores son falsas.

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

T T MAL

- ○ <u>(i)</u> es cierta pero <u>(ii)</u> no
- ○ <u>(ii)</u> es cierta pero <u>(i)</u> no
- ● Las dos son ciertas

101. Considérense la declaraciones de clase e instancia

```
class C a where  f, g::  a -> Int      | instance C Bool where  g True = 0   | instance C Int where  f x = x
                 f x = g x + 1         |                        g False = 1  |                       g x = 2*x
                 g x = f x - 1         |                                     |
```

¿Qué afirmación es correcta?

- ⊗ `f False` se evalúa a 2 y `g 1` se evalúa a 2 V
- ○ La evaluación de una de las expresiones del caso anterior da un error
- ○ Las dos anteriores son falsas.

102. 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
- ⊗ {`f`} es un conjunto minimal suficiente de métodos de `C`
- ○ {`f`,`g`} es un conjunto minimal suficiente de métodos de `C` F

103. 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`

104. 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 F
- ○ 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`

105. Considérense la declaraciones de clase e instancia

```
class C a where   f, g::  Int -> a   | instance C Bool where   g x = (x == 0)   | instance C Int where   f x = x
                  f x = g (x + 1)    |                                          |
                  g x = f (x - 1)    |                                          |
```

¿Qué afirmación es correcta?

- ○ `f 0 && g 0` se evalúa a `False` y `f 1` se evalúa a `1` F
- ● La evaluación de exactamente una de las expresiones del caso anterior da un error
- ○ Las dos anteriores son falsas.

MAL

106. Considérense la declaraciones de clase e instancia

```
class C a where   f, g::  Int -> a   | instance C Bool where   g 0 = True    | instance C Int where   f x = x
                  f x = g 0          |                         g _ = False   |                        g x = 2*x
                  g x = f 1          |                                       |
```

¿Qué afirmación es **falsa**?

- ○ Al intentar evaluar `f 0` resulta un error de ambigüedad de tipos
- ○ `not (f 0)` se evalúa a `False` V
- ⊗ `f 0 == f 0` se evalúa a `True` F

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

- ○ `f xs` computa la inversa de `xs`
- ○ `f xs` computa la propia lista `xs`
- ⊗ `f` está mal tipada

108. Considérense las funciones

```
f  xs = foldr g [] xs where g x y = x:filter (/= x) y
f' xs = foldl g [] xs where g y x = x:filter (/= x) y
```

Elimina rep de xs ≡ y mismo orden

Elimina rep y da vuelta

(y ojo al orden de argumentos en `g`*).* Entonces:

- ○ `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.

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

- ○ `f xs` computa la inversa de `xs`
- ● `f xs` computa la propia lista `xs`
- ○ `f` está mal tipada

MAL

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

- ○ `[1,1,2,2,3,3]`
- ⊗ `[3,3,2,2,1,1]`
- ○ `[3,2,1,3,2,1]`

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

- ○ `[1,2,3,0]`
- ○ `[3,1,2,3]`
- ⊗ `[1,1,2,3]`

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

- ⊗ `9`
- ○ `[1,0,4]`
- ○ Una lista de funciones

113. La evaluación de `foldl (\x y -> y-x ) 0 [1,2,3]` produce como resultado
- [ ] -4
- [ ] -6
- [x] 2

114. La evaluación de `foldr (\x y -> x-y ) 0 [1,2,3]` produce como resultado
- [ ] -4
- [ ] -6
- [x] 2

115. La evaluación de `foldr (\x y -> x/y ) 1 [8,4,2]` produce como resultado
- [x] 4
- [ ] 2
- [ ] 1

116. La evaluación de `foldl (\x y -> y-x ) 1 [1,2,3]` produce como resultado
- [ ] -1
- [x] 1
- [ ] 0

117. La evaluación de `foldl (\x y -> x-(last y)) 10 [[1,2],[3,4],[5]]` produce como resultado
- [ ] `5`
- [ ] `[8,6,5`
- [ ]
- [x] -1

118. Sea $\bigoplus$`::Int -> Int -> Int` una función semánticamente asociativa, n el valor de `foldl` $\bigoplus$ `0 [1,2,3]` y m el valor de `foldr` $\bigoplus$ `0 [1,2,3]`. Entonces
- [ ] Es seguro que $n = m$
- [ ] Es seguro que $n \neq m$
- [x] Las dos anteriores son falsas.

119. Sea n el valor de `foldl (\x y -> y) 0 [1,2,3]` y m el valor de `foldr (\x y -> y) 0 [1,2,3]`. Entonces
- [x] $n > m$
- [ ] $n < m$
- [ ] $n = m$

120. La evaluación de `foldr (\x y -> y+(x!!0)) 2 [[1,2],[3,4],[5,6]]` produce como resultado
- [x] `11`
- [ ] `[2,3,4,5,6,7]`
- [ ] Nada porque la expresión está mal tipada

121. La evaluación de `foldr (\x y -> y+(x!!1)) 2 [[1,2],[3,4],[5,6]]` produce como resultado
- [ ] `[3,4,5,6,7,8]`
- [x] `14`
- [ ] Nada porque la expresión está mal tipada

122. Sea `f:: Int -> Int -> Int` una función conmutativa, y considérense las igualdades siguientes:
(i) `foldr f 0 [3] = foldl f 0 [3]` ✓

(ii) `foldr f 0 [3,5] = foldl f 0 [3,5]`. Entonces:
- ◯ Tanto (i) como (ii) son con seguridad ciertas.
- ☒ Sólo (i) es con seguridad cierta.
- ◯ Las dos anteriores son falsas.

124. Sea l una lista finita de enteros positivos. Entonces la evaluación de la expresión
`foldr (\x y -> if x > 2 then x else y) 5` l
- ◯ Produce el valor 3, con independencia de l
- ◯ Produce el valor 5, con independencia de l
- ☒ Las dos anteriores son falsas.

125. Sea l una lista finita de enteros positivos. Entonces la evaluación de la expresión
`foldl (\x y -> if x > 2 then x else y) 5` l
- ◯ Produce el valor 3, con independencia de l
- ☒ Produce el valor 5, con independencia de l
- ◯ Las dos anteriores son falsas.

126. La evaluación de la expresión
`foldr (\x y -> not x || y) False [False,True,undefined]` da como resultado
- ☒ `True`
- ◯ `False`
- ◯ Un error en tiempo de ejecución

127. La evaluación de la expresión
`foldl (\x y -> not x || y) False [False,True,undefined]` da como resultado
- ◯ `True`
- ◯ `False`
- ☒ Un error en tiempo de ejecución

128. La evaluación de la expresión
`foldr (\x y -> (not y) && x) undefined [True,True]` da como resultado
- ◯ `True`
- ◯ `False`
- ☒ Un error en tiempo de ejecución

129. La evaluación de la expresión
`foldl (\x y -> (not y) && x) undefined [True,True]` da como resultado
- ◯ `True`
- ☒ `False`
- ◯ Un error en tiempo de ejecución

130. La reducción de la expresión `(\x y -> (\z -> y (z+2)) (y x)) 3 (\x -> x+1)` producirá el resultado
- ◯ 8
- ☒ 7
- ◯ 6

131. La reducción de la expresion `(\x y -> x (x y)) (\x -> x + 3) 4` producirá el resultado
- ◯ 7
- ◯ 10
- ◯ Las dos anteriores son falsas.

132. La reducción de la expresion `(\x y -> x (x y)) (\x -> x + y) 4` producirá el resultado
- ◯ 8
- ◯ 12
- ◯ Las dos anteriores son falsas.

133. La reducción de la expresión `(\x y -> (\u -> x (u+1)) (x y)) (\x -> x+2) 3` producirá el resultado
- ◯ 8
- ◯ 7
- ◯ 6

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

135. La reducción de la expresión `(\x y -> (\u -> x (u+1)) (x y)) (\x -> x+1) 3` producirá el resultado
- ◯ 5
- ◯ 6
- ◯ 7

136. La reducción de la expresion `(\x y -> y (y x)) ((\x -> x+1) 0) (\x -> x+1)` producirá el resultado

1

- ◯ 1
- ◯ 2
- ◯ 3

6

137. La reducción de la expresion `(\x y -> x + (\x -> y+x) y) ((\x -> x+1) 5) 2` producirá el resultado

6 + (\x -> 2+x) 2

- ◯ 10
- ◯ 14
- ◯ Depende del orden en que se efectúe

138. La reducción de la expresion `(\x y -> x (y x)) (\x -> x + 2) (\x -> x 3)` producirá el resultado
- ◯ 5
- ◯ 3
- ◯ 7

3

||

139. La reducción de la expresion `(\x y z -> x z y) (\x y -> x-y) ((\x -> x+1) 2) 1` producirá el resultado
- ◯ 1
- ◯ 2
- ◯ -2

140. La reducción de la expresión `(\x y -> y (y x)) 1 (\x -> x+2)` producirá el resultado
- ◯ 3
- ◯ 5
- ◯ 7

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

- ○ 1
- ○ 2
- ☒ 3

142. Considérense las funciones:

```
f x y    = [2*i|i <- [1..x], i>y]
f' x y   = filter (> y) (map (2 *) [1..x])
f'' x y  = map (2 *) (filter (> y) [1..x])
```

- ○ f, f' y f'' computan lo mismo
- ○ f y f' computan lo mismo, pero f'' no
- ● f y f'' computan lo mismo, pero f' no

MAL

143. La evaluación de `map (zipWith (-) [3,2,1]) [[1,2,3],[4,5,6]] !! 1 !! 0` produce como resultado:

- ○ [-1]
- ☒ -1
- ○ Un error, porque esa expresión está mal tipada

144. La evaluación de `[j|j <- [i-1..i+1],i <- [1..5], i > 1]` produce como resultado

- ○ 2
- ○ 3
- ☒ Un error

145. La evaluación de `length [i|i <- [1..5], j <- [1..i],i+j<5]` produce como resultado

- ○ 3
- ☒ 4
- ○ 5

146. La evaluación de `length [i+j|i <- [1..5], i > 2, j <- [3..i]]` produce como resultado

- ○ 0
- ☒ 6
- ○ Da un error en tiempo de ejecución

147. La evaluación de `length [i+j|i <- [1..5], i > 2, j <- [i-1,i]]` produce como resultado

- ○ 9
- ☒ 6
- ○ Da un error en tiempo de ejecución

148. La evaluación de `[i+j|i <- [1..5], i > 2, j <- [i-1,i]] !! 2` produce como resultado

- ○ 9
- ☒ 7
- ○ Da un error en tiempo de ejecución

149. La evaluación de `length [i+j|i <- [1..5], i > 2, j <- [i-1,i], j < i]` produce como resultado

i <- [3,4,5]   j <- [i-1]

- ○ 6
- ☒ 3
- ○ 0

150. La evaluación de `take 2 [take j [i..2*i] | i <- [1..10], i > 2, j <- [i-1..i+1]]` produce el resultado

- [ ] `[3,4]`
- [x] `[[3,4],[3,4,5]]`
- [ ] No produce ningún valor, porque la expresión está mal tipada

151. La evaluación de `length [take j [1..i] | i <- [3..5], ~~[illegible]~~, j <- [i-1,i]]` produce como resultado

- [ ] 3
- [x] 6
- [ ] No produce ningún valor, porque la expresión está mal tipada

152. La evaluación de `map (zip [1..4]) [2,5..12]` produce el resultado

- [ ] `[(1,11),(2,11),(3,11),(4,11)]`
- [ ] `[(4,11)]`
- [x] No produce ningún valor, porque la expresión está mal tipada

153. La evaluación de `map length [take 2 ys | x <- [1..3], ys <- iterate (+ 3) x]` produce como resultado

ys<[x, x+3, x+6, x+9, ---]

- [ ] [2,2,2]
- [ ] [2]
- [x] Un error, porque la expresión está mal tipada

154. La evaluación de `(head.tail) (map ((take 3).(iterate (+ 2))) [1,3..10])` produce como resultado

- [ ] `[4,6,8,11,14]`
- [ ] `[1,4,7]`
- [x] `[3,5,7]`

155. La evaluación de `take 3 (zipWith (+) [x+y|x<-[1..4],y<-[x,x+2,x+4]] (iterate (*2) 1))` produce como resultado

[1,2,4,8,...]

- [ ] `[3,5,7]`
- [x] `[3,6,10]`
- [ ] []

segundo

156. La evaluación de `(head.tail) (map ((take 2).(iterate (+ 3))) [1,3..10])` produce como resultado

- [ ] `[4,6,8,11,14]`
- [ ] `[1,4]`
- [x] `[3,6]`

157. La evaluación de `[j|i <- [1..5], i > 1, j <- [i-1..i+1]] !! i` produce como resultado

- [ ] 2
- [ ] 3
- [x] Un error, porque hay una variable fuera de ámbito

158. La evaluación de `map (!! 2) (map (iterate (\x -> 2*x)) [0..3])` produce el resultado

- [ ] 2
- [x] `[0,4,8,12]`
- [ ] No produce ningún valor, porque la expresión está mal tipada

159. La evaluación de `map fst [(zip [0..i] [1..j]) !! i | i <- [1..3], j <- [1..(i+1)], j > i]` produce el resultado
- [ ] `(0,1)`
- [x] `[1,2,3]`
- [ ] No produce ningún valor, porque la expresión está mal tipada

160. La evaluación de `((!! 2).head) (map (iterate (+2)) [2,0,4])` produce como resultado
- [x] 6
- [ ] [2,4,6]
- [ ] Las dos anteriores son falsas.

161. La evaluación de `(head.(!! 1)) (map (zip [0..3]) [[1..4],[2..5]])` produce como resultado
- [ ] `1`
- [ ] `(1,2)`
- [x] `(0,2)`

162. ¿Cuál de las siguientes definiciones es equivalente a `f n m = [x*y | x <- [1..n], y <- [x..m]]`?
- [x] `f n m = concat (map f [1..n]) where f x = map (\y -> x*y) [x..m]`
- [ ] `f n m = concat (map f [x..m]) where f y = map (\y -> x*y) [1..n]`
- [ ] `f n m = zipWith (*) [1..n] [x..m]`

163. ¿Cuál de las siguientes definiciones es equivalente a `f n m = [x*n | x <- [1..n], x > m]`?
- [x] `f n m = map (\x -> x*n) (filter (> m) [1..n])`
- [ ] `f n m = concat (map (\x -> x*n) (filter (> m) [1..n]))`
- [ ] `f n m = filter (> m) (map (\x -> x*n) [1..n])`

164. Sea `f` una función definida previamente, y considérense las definiciones

```
g    x y    =  let z = f x y in (z,x*y,z+1)
g'   x y    =  (f x y,x*y,f x y + 1)
g''  x y    =  h x y (f x y)
h    x y z  =  (z,x*y,z+1)
```

g=g'   g mejor g'

¿Qué afirmación es correcta?
- [x] `g, g'` y `g''` computan los mismos valores, `g` y `g''` tienen una eficiencia similar, pero `g'` es menos eficiente
- [ ] `g, g'` y `g''` computan los mismos valores, `g` y `g''` tienen una eficiencia similar, pero `g'` es más eficiente
- [ ] No es verdad que `g, g'` y `g''` computen los mismos valores

165. Sea `h::Int -> Int` una función costosa de calcular, y sean `f, f', f''` definidas por

```
f x y =  (x+u,y+u)        f' x y = (x+(h x) , y+(h x))      f'' x y = g x y (h x)
         where u = h x                                       g x y z = (x+z,y+z)
```

- [ ] Tanto `f'` como `f''` son extensionalmente equivalentes a `f` y comparables en eficiencia con ella.
- [x] Tanto `f'` como `f''` son extensionalmente equivalentes a `f`, pero `f'` es menos eficiente.
- [ ] Las dos anteriores son falsas.

166. ¿Cuál de las siguientes afirmaciones es cierta?
- [x] Las listas intensionales pueden eliminarse usando `map, filter, concat`, y recíprocamente
- [ ] Las listas intensionales pueden eliminarse usando `map, filter, concat`, pero el recíproco no es cierto
- [ ] Las dos anteriores son falsas.

167. Sea $l =$ `[(x,y)|x <- [0,2,3],`b`,y <- [1..x]]`, donde b es una cierta expresión booleana, y sea $n =$`length` l.
¿Cuál de la siguientes situaciones **no** puede darse?
- ◯ $n = 0$
- ◯ $n = 5$
- ⊗ $n = 4$

168. Sea $l =$ `[(x,y)|x <- [2,4],y <- [1..x],`b`]`, donde b es una cierta expresión booleana , y sea $n =$`length` l.
¿Cuántas de la siguientes situaciones: $n = 1$ ✓ $n = 5$ ✓ $n = 7$ pueden darse?
- ◯ Solo una de ellas
- ⊗ Solo dos de ellas
- ◯ Las tres son posibles

169. Considérense las siguientes expresiones:
```
head (reverse [1..10^30])
last (reverse [1..10^30])
last (takeWhile (< 1000) ([1..10^30] ++ [1..10^30]))
```
Sí / Sí / NO

¿Cuántas de ellas nos llevará toda la vida evaluarlas?
- ◯ Exactamente una de ellas
- ⊗ Exactamente dos de ellas
- ◯ Las tres

170. Considérense las siguientes expresiones:
```
take 30 (reverse [1..10^30])
reverse (take 30 [1..10^30])
last (takeWhile (< 1000) (iterate (+ 2) 1))
```
Sí / NO / NO

¿Cuántas de ellas nos llevará toda la vida evaluarlas?
- ⊗ Exactamente una de ellas
- ◯ Exactamente dos de ellas
- ◯ Las tres

171. Considérense las siguientes expresiones:
```
head (reverse (reverse [1..10^30]))
reverse (reverse (take 30 [1..10^30]))
take 100 (iterate (+ 2) 1 ++ [1..100]))
```
Sí / NO / NO

¿Cuántas de ellas nos llevará toda la vida evaluarlas?
- ⊗ Exactamente una de ellas
- ◯ Exactamente dos de ellas
- ◯ Las tres

172. Considérense las siguientes definiciones de funciones:

`f x y z = x + y - z`   `g x y = f x x y`   `g' x = f x x`   `g'' = f x`

Entonces:
- ◯ `g, g' y g''` son equivalentes
- ● `g y g'` son equivalentes, pero la definición de `g''` contiene un error
- ◯ Las dos anteriores son falsas.

MAL

173. Considérense las siguientes definiciones de funciones:

`f x y z = x + y - z`   `g x y = f x y y`   `g' x = f x`   `g'' x = f x y`

Entonces:

○ `g, g’` y `g’’` son equivalentes
○ `g` y `g’` son equivalentes, pero la definición de `g’’` contiene un error
○ Las dos anteriores son falsas.

174. Considérense las siguientes definiciones de funciones:

`f x y z = x + y - z`   `g x y z = f (x+1) y z`   `g’ x y = f (x+1) y`   `g’’ x = f (x+1)`

○ `g, g’` y `g’’` son equivalentes
○ `g` y `g’` son equivalentes entre sí, pero no equivalentes a `g’’`
○ La definición de `g’’` contiene un error

175. Sean las definiciones `f g h x y = g (h x (g y))` `f’ g h x = g.(h x).g` `f’’ g h = g.(\x -> h x).g`

○ `f`, `f’` y `f’’` son extensionalmente equivalentes
○ `f` y `f’` son extensionalmente equivalentes, pero `f’’` no
○ `f` y `f’’` son extensionalmente equivalentes, pero `f’` no

176. Sean las definiciones `f g h x = g (h (g x))` `f’ g h x = g.h.(\x -> g x)` `f’’ g h = g.h.g`

○ `f`, `f’` y `f’’` son extensionalmente equivalentes
○ `f` y `f’` son extensionalmente equivalentes, pero `f’’` no
○ `f` y `f’’` son extensionalmente equivalentes, pero `f’` no

177. Considérense las siguientes definiciones de funciones:

`f1 x y z = x . y`   `f2 x y z = (x . y) z`   `f3 x y z = x (y z)`   `f4 x y = x . y`

○ `f1, f2, f3, f4` computan la misma función
○ `f1, f2, f3` computan la misma función, pero `f4` no
○ `f2, f3, f4` computan la misma función, pero `f1` no

178. Considérense las igualdades
`(take m).(take n) = take (min n m)` | `(drop m).(drop n) = drop (n+m)` | `(take m).(drop n) = (drop n).(take (m+n))`
donde `m,n` son ≥ 0. Se tiene:

○ Las tres son correctas
○ Solo dos son correctas
○ Solo una es correcta

179. ¿Cuántas de las siguientes igualdades son correctas?
`map id = id` | `map f (xs ++ ys) = (map f xs) ++ (map f ys)` | `map (f.g) xs = (map f . map g) xs`
Nota: en la primera de ellas, se entiende que las funciones de ambos lados se aplican solo a listas

○ Las tres
○ Solo dos
○ Solo una

180. Sea un programa funcional con dos funciones `f,g::Int -> Int` y la función `h = h`. ¿Cuál de las siguientes situaciones es posible?

○ $[\![\texttt{g (f h)}]\!] = 2$ y $[\![\texttt{g (f 0)}]\!] = \perp$
○ $[\![\texttt{g (f h)}]\!] = \perp$ y $[\![\texttt{g (f 0)}]\!] = 2$
○ $[\![\texttt{g (f h)}]\!] = 0$ y $[\![\texttt{g (f 0)}]\!] = 2$

181. Sea un programa funcional con dos funciones `f,g::Int -> Int` y la función `h = h`. ¿Cuál de las siguientes situaciones no es posible?
○ $\llbracket \texttt{g (f h)} \rrbracket = 2$ y $\llbracket \texttt{g (f 0)} \rrbracket = \bot$
○ $\llbracket \texttt{g (f h)} \rrbracket = \bot$ y $\llbracket \texttt{g (f 0)} \rrbracket = 2$
○ $\llbracket \texttt{g (f h)} \rrbracket = \bot$ y $\llbracket \texttt{g (f 0)} \rrbracket = \bot$

182. Sea un programa funcional con dos funciones `f,g::Int -> Int` y la función `loop = loop`. ¿Cuál de las siguientes situaciones es posible?
○ $\llbracket \texttt{f (g loop)} \rrbracket = 1$ pero $\llbracket \texttt{f (g 0)} \rrbracket = \bot$
○ $\llbracket \texttt{f (g loop)} \rrbracket = \bot$ pero $\llbracket \texttt{f (g 0)} \rrbracket = 1$
○ $\llbracket \texttt{f (g loop)} \rrbracket = 0$ pero $\llbracket \texttt{f (g 0)} \rrbracket = 1$

183. De una función `f` solo sabemos que su tipo es `∀a.[a] -> Int`. ¿Cuál de las siguientes situaciones es posible?
○ $\llbracket \texttt{f [1,2]} \rrbracket = 2$ y $\llbracket \texttt{f [True,True]} \rrbracket = 1$
○ $\llbracket \texttt{f [1,2]} \rrbracket = 1$ y $\llbracket \texttt{f [True,True]} \rrbracket = 1$
○ $\llbracket \texttt{f [1,2]} \rrbracket = 1$ y $\llbracket \texttt{f [True,True]} \rrbracket = 2$

184. De una función `f` solo sabemos que su tipo es `∀a.Eq a => [a] -> Int`. ¿Cuántas de las siguientes situaciones son posibles?
- $\llbracket \texttt{f [1,2]} \rrbracket = 2$ y $\llbracket \texttt{f [True,True]} \rrbracket = 1$
- $\llbracket \texttt{f [1,2]} \rrbracket = 1$ y $\llbracket \texttt{f [True,True]} \rrbracket = 1$
- $\llbracket \texttt{f [1,2]} \rrbracket = 1$ y $\llbracket \texttt{f [True,True]} \rrbracket = 2$

○ Las tres
○ Solo dos
○ Solo una

185. La función `f` definida por las siguientes ecuaciones:
```
f      x   False  = True
f  False       y  = True
f   True    True  = False
```
MAL
○ Es estricta en el segundo argumento pero no en el primero
○ No es estricta en ninguno de sus argumentos
○ Es estricta en sus dos argumentos

186. Sean las funciones
```
f x y = if x > 0 then 1 else y
g x = if x > 0 then 1 else 0
h x y = (g.(f x)) y
```
○ La función `h` es estricta en el primer argumento pero no en el segundo
○ La función `h` no es estricta en ninguno de sus argumentos
○ La función `h` es estricta en sus dos argumentos

187. Considérese el programa
```
f y 0 = g y
f 0 x = h x (x*x)
g x = if x > 0 then 1 else 0
h x y = if x > y then 1 else 0
```
○ La función `f` es estricta en el segundo argumento pero no en el primero
○ La función `f` no es estricta en ninguno de sus argumentos
○ La función `f` es estricta en sus dos argumentos

188. Considérese el programa

```
f 0 y = g y
f x 0 = h x (x*x)
g x = if x > 0 then 1 else 0
h x y = if x > y then 1 else 0
```

- [ ] La función `f` es estricta en el primer argumento pero no en el segundo
- [ ] La función `f` no es estricta en ninguno de sus argumentos
- [x] La función `f` es estricta en sus dos argumentos

189. Considérese el programa

```
f 0 y = g y
f x y = h y
g x = if x > 0 then 1 else 0
h x = 0
```

- [ ] La función `f` no es estricta en ninguno de sus argumentos
- [ ] La función `f` es estricta en sus dos argumentos
- [x] Las dos anteriores son falsas.

190. La función `f` definida por las siguientes ecuaciones:

```
f  False      y   = True
f      x  False   = True
f   True   True   = False
```

- [ ] No es estricta en ninguno de sus argumentos
- [x] Es estricta en el primer argumento pero no en el segundo
- [ ] Las dos anteriores son falsas.

191. Sea la función `f` definida por las siguientes ecuaciones:

```
f      x  False   = True
f  False      y   = True
f   True   True   = False
```

y considérense las siguientes afirmaciones: *(a)* $[\![\texttt{f e } \bot]\!] = \bot$, para toda expresión `e` V
(b) $[\![\texttt{f } \bot \texttt{ e}]\!] = \bot$, para toda expresión `e` F

- [ ] *(a)* y *(b)* son ciertas
- [x] *(a)* es cierta y *(b)* es falsa
- [ ] Las dos anteriores son falsas.

192. Considérese el programa

```
f 0 y = g y        g x = if x > 0 then 1 else 0
f x y = h y        h x = 0
mal = head []
```

- [ ] Existen e, e' tales que ni la evaluación de (`f` e `mal`) ni la de (`f mal` e') dan error
- [ ] Para todo e, e' `f` e `mal` y `f mal` e' dan error
- [x] Las dos anteriores son falsas.

193. Sea `f` definida por las siguientes ecuaciones: y sea `mal = head []`.

```
f      x  False   = True
f  False      y   = True
f      _      _   = False
```

¿Cuál de las siguientes afirmaciones es cierta?

- ○ `f mal` e da error de ejecución, para cualquier expresión e F
- ⊗ `f` e `mal` da error de ejecución, para cualquier expresión e V
- ○ `f mal True` se evalúa a `False` F

194. Sea `f` definida por las siguientes ecuaciones:

```
f     x  False  = True
f False      y  = True
f  True   True  = False
```

y sea `mal = head []`.

¿Cuál de las siguientes afirmaciones es cierta?

- ○ La evaluación de `f mal` e da error, para cualquier expresión e F
- ⊗ La evaluación de `f` e `mal` da error, para cualquier expresión e V
- ○ `f mal True` se evalúa a `False` F

195. Sea `f` definida por las siguientes ecuaciones:

```
f False      y  = y
f     x  False  = True
f  True   True  = False
```

y sea `mal = head []`. ¿Cuál de las siguientes afirmaciones es **falsa**?

- ○ La evaluación de `f mal` e da error, para cualquier expresión e V
- ○ La evaluación de `f` e `mal` da error, para cualquier expresión e cuya evaluaci V
’on termine
- ⊗ Las dos anteriores son falsas.

196. Considérense las definiciones:

```
f x y  | x == 0     = 1        .
       | x < y      = g (x-y)
       | otherwise  = 2
g x    | x >= 0     = x
```

MAL

¿Qué afirmación es correcta?

- ○ $f\ x\ y$ tiene un valor definido para valores cualesquiera de x e y
- ● $f\ x\ y$ tiene un valor definido $\Leftrightarrow x = 0 \vee x \geq y$
- ○ La definición es errónea y será rechazada por el sistema

197. Considérese el programa

```
f x 0 = g x
f x y = h x
g x = if x > 0 then 1 else 0
h x = 0
```

- ○ La función `f` no es estricta en ninguno de sus argumentos
- ○ La función `f` es estricta en sus dos argumentos
- ⊗ Las dos anteriores son falsas.

198. Sea `f` definida por las siguientes ecuaciones:

```
f  x  False  = False
f  y   True  = y
```

¿Cuál de las siguientes afirmaciones es cierta?

- ○ La función es estricta en sus dos argumentos F
- ⊗ La función es estricta en el segundo pero no en el primer argumento
- ○ Las dos anteriores son falsas.

199. Sea `f` definida por las siguientes ecuaciones:

```
f  x  False  = x
f  y   True  = not y
```

¿Cuál de las siguientes afirmaciones es cierta?

- ◯ La función es estricta en sus dos argumentos
- ◯ La función es estricta en el segundo pero no en el primer argumento
- ◯ Las dos anteriores son falsas.

200. Sea `f` definida por las siguientes ecuaciones:

```
f     x   False  = True
f  True    True  = False
```

¿Cuál de las siguientes afirmaciones es cierta?
- ◯ La función es estricta en sus dos argumentos
- ◯ La función es estricta en el segundo pero no en el primer argumento
- ◯ Las dos anteriores son falsas.

201. Dada la definición del tipo `data T = A | B | C T T`, considérese su dominio $\mathcal{D}_T$, y el orden de aproximación sobre él $\sqsubseteq$.
¿Cuál de las siguientes afirmaciones es cierta?
- ◯ $\texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{C A B}$
- ◯ $\mathcal{D}_T$ es finito y $\perp$ es el elemento mínimo para $\sqsubseteq$
- ◯ Las dos anteriores son falsas.

202. Dada la definición del tipo `data T = A | B | C T T`, considérese su dominio $\mathcal{D}_T$, y el orden de aproximación sobre él $\sqsubseteq$.
¿Cuál de las siguientes afirmaciones es cierta?
- ◯ `A` y `B` son los únicos maximales para $\sqsubseteq$
- ◯ `A` y `B` son maximales para $\sqsubseteq$ pero no son los únicos
- ◯ Las dos anteriores son falsas.

203. Considérese la definición del tipo `data T = A | B | C T T`, y sea $\sqsubseteq$ el orden de información o de aproximación sobre los valores de `T`. ¿Cuál de las siguientes afirmaciones es cierta?
- ◯ $\perp \sqsubseteq \texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{C} \perp \perp$
- ◯ `A` y `B` son los únicos valores maximales para $\sqsubseteq$
- ◯ $\perp \sqsubseteq \texttt{C} \perp \perp \sqsubseteq \texttt{C} \perp \texttt{B} \sqsubseteq \texttt{C A B}$

204. Considérese la definición del tipo `data T = A | B | C T T`, y sea $\sqsubseteq$ el orden de información o de aproximación sobre los valores de `T`. ¿Cuál de las siguientes afirmaciones es cierta?
- ◯ $\perp \sqsubseteq \texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{C} \perp \perp$
- ◯ `A` y `B` son valores maximales para $\sqsubseteq$
- ◯ $\perp \sqsubseteq \texttt{C A} \perp \sqsubseteq \texttt{C} \perp \texttt{B} \sqsubseteq \texttt{C A B}$

205. Considérese la definición del tipo `data T = A | B | C T T`, y sea $\sqsubseteq$ el orden de información o de aproximación sobre los valores de `T`. ¿Cuál de las siguientes afirmaciones es **falsa**?
- ◯ $\perp \sqsubseteq \texttt{C} \perp \perp \sqsubseteq \texttt{C A A}$
- ◯ `C A A` es un valor maximal para $\sqsubseteq$
- ◯ $\perp \sqsubseteq \texttt{C} \perp \texttt{B} \sqsubseteq \texttt{C B} \perp \sqsubseteq \texttt{C A B}$

206. En lo que sigue, $\sqsubseteq$ indica el orden de aproximación entre valores semánticos. ¿Cuál de las siguientes afirmaciones es cierta?
- ◯ $[\perp] \sqsubseteq [0] \sqsubseteq [0,\perp]$
- ◯ $[\perp] \sqsubseteq [\perp,\perp] \sqsubseteq [0,\perp]$
- ◯ $\perp \sqsubseteq [\perp,\perp] \sqsubseteq [0,\perp]$

207. En lo que sigue, $\sqsubseteq$ indica el orden de aproximación entre valores semánticos. ¿Cuál de las siguientes afirmaciones es cierta?
   - ◯ $[\ ] \sqsubseteq [\bot] \sqsubseteq [\bot,\bot]$
   - ◯ $[\bot] \sqsubseteq [\bot,\bot] \sqsubseteq [0,\bot]$
   - ◯ Las dos anteriores son falsas.

208. En lo que sigue, $\sqsubseteq$ indica el orden de aproximación entre valores semánticos del tipo `[Int]`. ¿Cuál de las siguientes afirmaciones es falsa?
   - ◯ $\bot \sqsubseteq [\bot] \sqsubseteq [0]$
   - ◯ $[\bot] \sqsubseteq [\bot,\bot] \sqsubseteq [0,\bot]$
   - ◯ $\bot \sqsubseteq [\bot,\bot] \sqsubseteq [0,\bot]$

209. El valor de la expresión `let x=1:3:x in take 3 x` es:
   - ◯ [1,3,1]
   - ◯ ⊥, porque la evaluación no termina
   - ◯ Hay un error sintáctico porque `x` no puede aparecer en el lado derecho de `let x= ...`

210. El valor de la expresión `let x=1:3:x in head (x ++ x)` es:
   - ◯ 1
   - ◯ ⊥, porque la evaluación no termina
   - ◯ Hay un error sintáctico porque `x` no puede aparecer en el lado derecho de `let x= ...`

211. El valor de la expresión `let x= reverse (x++[1]) in head x` es:
   - ◯ 1
   - ◯ La evaluación no termina
   - ◯ La expresión es incorrecta, porque `x` no puede aparecer a la derecha de `let x = ...`

212. El valor de la expresión `let x=x++x in head (2:x)` es:
   - ◯ 2
   - ◯ ⊥, porque la evaluación no termina
   - ◯ La expresión está mal construida o mal tipada

213. El valor de la expresión `let x=x++[1] in last x` es:
   - ◯ 1
   - ◯ ⊥, porque la evaluación no termina
   - ◯ La expresión está mal construida o mal tipada

214. Considére la evaluación de las expresiones

```
let x=2:filter (/=2) x in head x
let x=2:filter (/=2) x in head (tail x)
```

MAL

   - ◯ Ninguna de las dos termina
   - ◯ Una de las dos termina
   - ◯ Las dos terminan

215. El valor de la expresión `let x=x++[1] in x!!1` es:
   - ◯ 1
   - ◯ ⊥, porque la evaluación no termina
   - ◯ La expresión está mal construida o mal tipada

MAL

216. El valor de la expresión `let x=[1]++x in head (tail x)` es:
◯ 1
◯ La evaluación no termina
◯ La expresión está mal construida o mal tipada

217. El valor de la expresión `let x=1:y in head x` es:
◯ 1
◯ $\perp$, porque la evaluación no termina
◯ La expresión está mal construida o mal tipada

218. La evaluación de la expresión `let x= 1:map (+ 2) x in take 3 x` produce como resultado:
◯ [1,3,5]
◯ Una lista que empieza por `1` y luego da un error
◯ Un error sintáctico o de tipos

219. La evaluación de la expresión `let x= 1:3:map (+ 1) x in last (take 3 x)` produce como resultado:
◯ `2`
◯ Un error en ejecución
◯ Un error sintáctico o de tipos

220. La evaluación de la expresión `let x= 1:map (+ 1) x in length (take 4 x)` produce como resultado:
◯ `4`
◯ Un cómputo no terminante
◯ Un error

221. Dadas las expresiones: `length [1..]` y `let x=length [1..] in fst (1,x)`
◯ La evaluación de la segunda termina pero la de la primera no
◯ La evaluación de ninguna de las dos termina
◯ Las dos anteriores son falsas.

222. Considérense las definiciones

```
f x = 1 + f (x+1)
g x = if x >= 1 then 1 else 0
h x = 3
```

y las expresiones $e \equiv$ `g (f 1)` y $e' \equiv$ `h (g (f 1))`. Entonces:
◯ Ni la evaluación de e ni la de e' terminan, tanto al usar evaluación impaciente como perezosa.
◯ Ni la evaluación de e ni la de e' terminan al usar evaluación impaciente, pero sí lo hacen (ambas) al usar evaluación perezosa.
◯ Las dos anteriores son falsas.

223. Sea e una expresión. ¿Cuál de las siguientes situaciones es posible?
◯ Al evaluar e por evaluación impaciente se obtiene el valor 3, y por evaluación perezosa el valor 2
◯ Al evaluar e por evaluación impaciente resulta el valor 3, y por evaluación perezosa el cómputo no termina
◯ Al evaluar e por evaluación perezosa resulta el valor 3, y por evaluación impaciente el cómputo no termina

224. Sea e una expresión. ¿Cuántas de las siguientes situaciones son posibles al evaluar e?
• Por evaluación impaciente se obtiene el valor 3, y por evaluación perezosa el valor 2. F
• Se obtiene el valor 3 tanto por evaluación impaciente como por evaluación perezosa. V

- Por evaluación perezosa el cómputo no termina, y por evaluación impaciente el cómputo termina. F
- Por evaluación perezosa el cómputo termina, y por evaluación impaciente el cómputo no termina. V
- El cómputo no termina ni por evaluación impaciente ni por evaluación perezosa. V

◯ Exactamente dos
⊗ Exactamente tres
◯ Las dos anteriores son falsas.

225. Sean las funciones `f` y `g` definidas por `f x = f x` y por `g f x = if x==0 then 0 else f (x-1)`.
Entonces, al evaluar la expresión `g f 1`:
◯ Con evaluación perezosa se obtiene el valor 0, y con evaluación impaciente el cómputo no termina.
◯ Tanto con evaluación perezosa como con evaluación impaciente se obtiene el valor 0.
⊗ Tanto con evaluación perezosa como con evaluación impaciente el cómputo no termina.

226. Sean las funciones `f` y `g` definidas por `f g = g (f g)` y por `g f x = if x==0 then 0 else f (x-1)`. MAL
Entonces, al evaluar la expresión `f g 1`:
◯ Con evaluación perezosa se obtiene el valor 0, y con evaluación impaciente el cómputo no termina.
⊗ Tanto con evaluación perezosa como con evaluación impaciente se obtiene el valor 0. ??
◯ Tanto con evaluación perezosa como con evaluación impaciente el cómputo no termina.

227. Sean las funciones `f` y `g` definidas por `f g = g (f g)` y por `g f x = if x==[] then 0 else 1 + f (tail x)`.
Entonces, al evaluar la expresión `f g [2]`:
◯ Con evaluación perezosa se obtiene el valor 2, y con evaluación impaciente el cómputo no termina.
⊗ Con evaluación perezosa se obtiene el valor 1.
◯ Tanto con evaluación perezosa como con evaluación impaciente el cómputo no termina.

228. ¿Cuáles de las dos siguientes expresiones representa correctamente una acción de I/0?

```
do  x <- getChar              do  x <- getChar
    x                             return x
```

◯ Las dos
◯ Solo la primera
⊗ Las dos anteriores son falsas.

229. ¿Cuál de las siguientes expresiones denota correctamente la acción de leer un carácter y escribirlo dos veces?

◯ `let x=getChar in [x,x]` F
◯
```
do  x <- getChar
    return x
    return x
```
⊗
```
do  x <- getChar
    putStr [x,x]
```

230. Considérense las definiciones:

```
f::  IO ()                         g = do  x <- f
f = do  x <- getChar                       print x
        print x
```

Entonces, la evaluación de `g` tiene por efecto:
◯ Leer un carácter y escribirlo dos veces
◯ `g` no llegará a evaluarse pues hay errores de tipo en ese código
⊗ Las dos anteriores son falsas.