PROGRAMACIÓN DECLARATIVA CURSO 2018-19 16 de enero de 2019

APELLIDOS, NOMBRE:

La puntuación total del examen es de **7,5 puntos**

De ellos, **2,5 puntos** corresponden a este test

PREGUNTAS DE TEST

• Cada pregunta tiene (espero) una y solo una respuesta correcta. Marcad con un aspa la opción elegida.

• **Cada respuesta correcta suma un punto; cada respuesta incorrecta resta medio punto**; las respuestas en blanco ni suman ni restan. Estad ojo avizor y suerte. Está prohibidísimo copiar.

---

1. ¿Cuántas de las siguientes expresiones son sintácticamente equivalentes a `[[1],[2,3]]`?

`[1]:[2,3]`   `[[1]]:[[2,3]]`   `[1]:([2,3]:[])`   `([1]:[2,3]):[]`

◯ Ninguna
⊗ Exactamente una
◯ Exactamente dos

---

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

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

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

---

3. Considérese el operador `infixr 4 ?` y las expresiones:

$e_1 =$ `f x y ? g y ? x`
$e_2 =$ `(?) ((f x) y) (g y ? x)`
$e_3 =$ `((f x y) ?) ((?) (g y) x)`

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

---

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

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

¿Cuál de las siguientes afirmaciones es cierta?

◯ La función es estricta en sus tres argumentos
⊗ La función es estricta en el segundo argumento y en ninguno otro más
◯ Las dos anteriores son falsas.

---

5. La evaluación de la expresión `length (foldr (\x y -> x:y) [1] [undefined,1])` da como resultado

⊗ `3`
◯ Un error de tipos
◯ Un error en tiempo de ejecución

---

6. La evaluación de la expresión `(\x -> (\y -> y x) (\x -> x+1)) 4` da como resultado

⊗ `5`

◯ `6`

◯ Un error de tipos

---

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

◯ `(a -> b) -> a -> (c -> b) -> b`

◯ `(a -> b) -> a -> (b -> b) -> b`

⊗ `(a -> b) -> a -> (b -> c) -> c`

---

8. La evaluación de la expresión `[take 3 (iterate (*i) i) !! 1 | i <- [1..10]]` produce como resultado:

◯ Una lista formada por diez listas de longitud 3

⊗ Una lista formada por diez números

◯ Un cómputo no terminante

---

9. Considérense el programa lógico:

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

y los objetivos:

i) `p(X,f(a)).`

ii) `p(f(a),Y).`

⊗ El árbol de resolución de *i)* tiene alguna rama infinita y *ii)* solo tiene como solución `Y = f(a)`.

◯ *i)* solo tiene como solución `X = a` y *ii)* solo tiene como solución `Y = f(a)`.

◯ *i)* y *ii)* tienen infinitas soluciones.

---

10. Considérense los siguientes tres objetivos Prolog:

```
var(X),X is 1+1,X == 1+1.
X = 1+1,var(X),X is 1+1.
var(X),X = 1+1,X == 1+1.
```

◯ Ninguno de ellos tiene éxito.

⊗ Uno de ellos tiene éxito y los otros dos no.

◯ Dos de ellos tienen éxito y el otro no.

EXAMEN FINAL de PROGRAMACIÓN DECLARATIVA

PREGUNTAS DE DESARROLLO

1. **[0,5 puntos]**
   Escribe una expresión Haskell cuya evaluación produzca la lista infinita
   `[(0,0),(1,2),(3,6),(7,14),(15,30),...,...`

```
[(2 ^ i - 1, 2 * (2 ^ i - 1)) | i<-[0..]]
```

---

2. **[0,5 puntos]**
   Razona brevemente cuál es el tipo de la función definida por la ecuación
   `f x y z = y (y x z) z`

La función `f` recibe tres parámetros luego su tipo ha de ser:
`f :: A ->B ->C ->D`
Siendo `A, B, C` el tipo de los parámetros `x, y, z`, respectivamente, y `D` el tipo del resultado. Es decir:
`x :: A`, `y :: B`, `z :: C`,
`y (y x z) z :: D`
A su vez, si `y (y x z) z :: D`, `y` será una función que aplicada a dos parámetros da como resultado una expresión de tipo `D`. Podemos asegurar que el segundo de estos parámetros es de tipo `C`, pues `z` figura como segundo parámetro de `y` tanto en la expresión `y (y x z) z`, como en `(y x z)`. En esta segunda expresión el primer parámtetro de `y` es `x`, a cuyo tipo hemos llamado `A`, por lo que:
`y :: A ->C ->D`. Y por tanto, la expresión `(y x z)` tendrá el tipo resultante de aplicar la función `y` a dos argumentos es decir:
`(y x z) :: D`
Pero como `(y x z)` es el primer parámetro de `y` en la expresión `y (y x z) z`, a la fuerza `D` = `A`.
Recopilando, `A` es sencillamente una variable de tipo, llamésmosla `a`, y `C` una variable de tipo, llamésmosla `b`, mientras que `B` (el tipo supuesto para `y`) debe ser `a ->b ->a` y concluimos:
`f :: a ->(a ->b ->a) ->b ->a`.

---

3. **[1 punto]**
   - Define un tipo de datos polimórfico `ArbolT a` para representar árboles ternarios, en los que cada nodo tiene una información de tipo `a` y, salvo que sea una hoja, tiene tres hijos, que de nuevo son árboles ternarios.
   - Programa la función `sumaCentral t` que, dado un árbol ternario de números, obtiene la suma de los elementos de la rama central del árbol.

```
data ArbolT a = Hoja a | Nodo a (ArbolT a) (ArbolT a) (ArbolT a) deriving (Show, Eq, Read)

  sumaCentral :: Num a => ArbolT a -> a
  sumaCentral (Hoja x) = x
  sumaCentral (Nodo x hi hc hd) = x + sumaCentral hc
```

---

4. Programa en Haskell las siguientes funciones, indicando sus tipos:

   a) **[0,5 puntos]**
   `sublista xs ys` = `True` si los elementos de la lista `xs` aparecen consecutivamente en `ys` (`False` e.o.c).

   b) **[0,5 puntos]**
   `maxterna xs` = (i, j, k), siendo (i, j, k) las tres posiciones consecutivas en `xs` tales que la suma de los elementos en esas posiciones es máxima. Se supone que las posiciones se numeran desde 0.
   Ejemplo: maxterna [7,3,10,15,2] = (1,2,3)

Una forma corta, sin preocuparse de la eficiencia (se obtienen muchas repeticiones).

```
sublista :: [a] -> [a] -> Bool
sublista xs ys = elem xs [take n (drop m ys) | n <- [0..length ys], m <- [0..length ys]]

maxterna :: (Ord a, Num a) => [a] -> (Int,Int,Int)
maxterna xs
  | (length xs) >= 3 = let i = maxIndex  (sumaternas xs) in (i,i+1,i+2)
  | otherwise      = error "lista con menos de 3 elementos"

sumaternas :: Num a => [a] -> [(Int,a)]
sumaternas xs = [(i, (xs !! i)+(xs !! (i+1))+(xs !! (i+2))) | i<- [0..(length xs)-3 ]]

maxIndex :: Ord a => [(Int,a)] -> Int
maxIndex xs = fst $ foldl1 (\ (i,x) (j,y) -> (if x < y then (j,y) else (i,x))) xs
```

---

5. Programa en Prolog los siguientes predicados:

   a) **[0,25 puntos]**
   `ordenada(Xs)` ↔ los elementos de la lista `Xs` están ordenados de menor a mayor.

   b) **[0,75 punto]**
   `max_aparece(Xs,X)` ↔ la lista `Xs` está ordenada y `X` es el elemento que aparece un número mayor de veces en `Xs`, o bien es la constante `ninguno` si `Xs` es vacía.

```
ordenada([]) :- !.
ordenada([_]) :- !.
ordenada([X,Y|Xs] ) :- X @=< Y, ordenada([Y|Xs]).

maxaparece([],ninguno) :- !.
maxaparece([X|Xs],Y) :- ordenada([X|Xs]), maxaparece(|Xs,X,1,X,1,Y).

maxaparece([],_,_,Y,_,Y) :- !.
maxaparece([X|Xs],X,N,Z,M,Y) :- !,
    N1 is N + 1,
    maxaparece(Xs,X,N1,Z,M,Y).
maxaparece([X|Xs],X1,N,_,M,Y) :-
    N > M, !,
    maxaparece(Xs,X,1,X1,N,Y).
maxaparece([X|Xs],_,_,Z,M,Y) :-
    maxaparece(Xs,X,1,Z,M,Y).
```

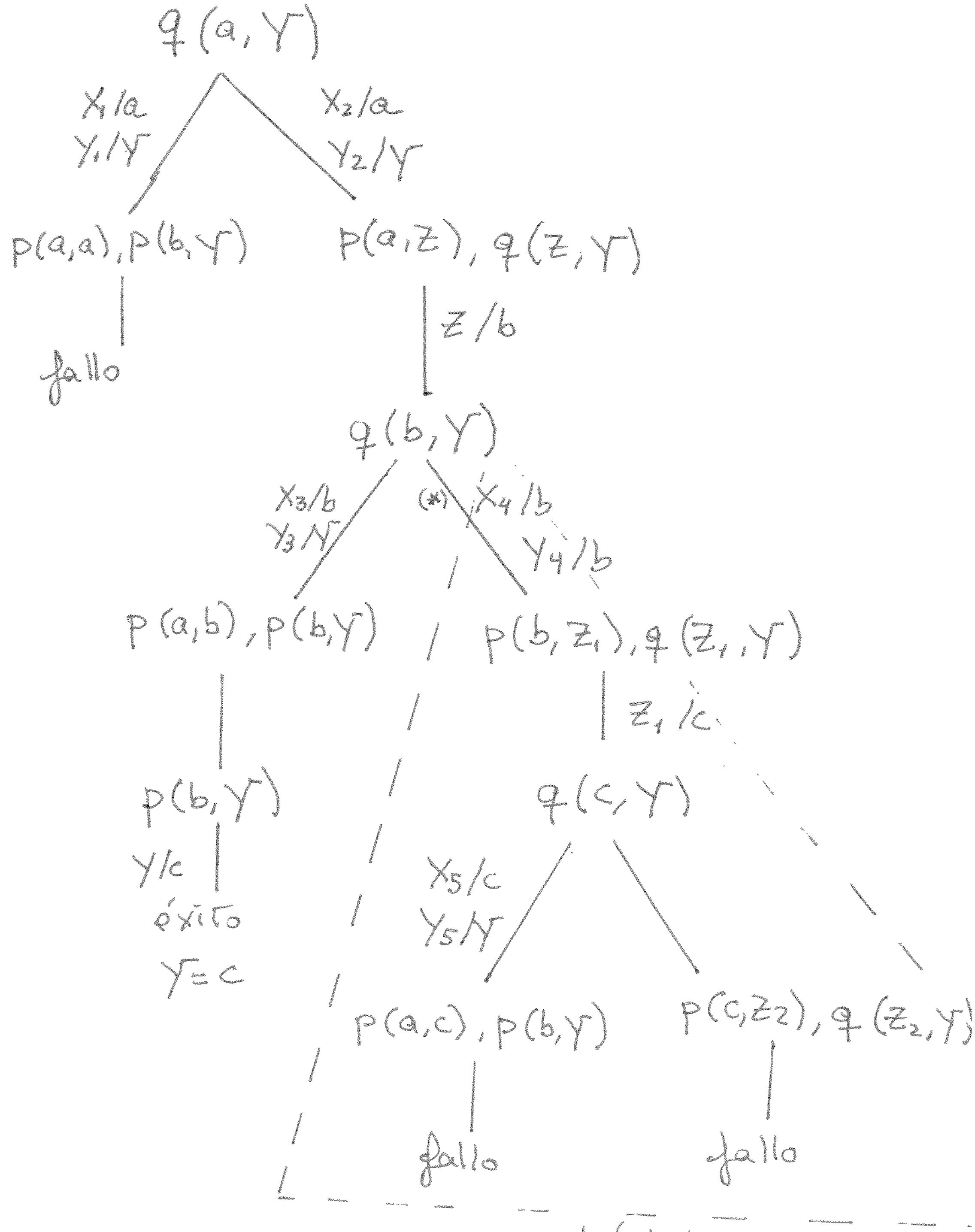
---

6. Dado el programa lógico

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

   a) **[0,75 puntos]**
   Construye el árbol de resolución del objetivo `q(a,Y)`.

   b) **[0,25 puntos]**
   Señala en el árbol anterior las ramas que se podan si la primera cláusula del predicado `q` se sustituye por `q(X,Y) :- p(a,X), !, p(b,Y).`



subárbol
podado en apartado b)

Desaparecen todas las ramas
a partir de (*)