PROGRAMACIÓN DECLARATIVA CURSO 2018-19 20 de junio 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. La nota mínima del test es 0.

---

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

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

⊗ Exactamente una
◯ Exactamente dos
◯ Exactamente tres

---

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

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

---

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

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

---

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

```
f x 0 z = x == z
f x y z = x
```

¿Cuál de las siguientes afirmaciones es cierta?

◯ La función es estricta en sus tres argumentos.
◯ La función es estricta únicamente en el segundo argumento.
⊗ La función es estricta en los argumentos primero y segundo.

---

5. 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

---

6. La evaluación de la expresión `(\x y -> y (x + 2)) (\y -> y / 2) 2` da como resultado

◯ `2`
◯ `3`
⊗ Un error de tipos.

---

7. Sea `f` definida por `f g h = (g h).(g h)`. El tipo de `f` es:

◯ `(a -> a) -> (a -> a)`
◯ `(a -> b) -> (a -> b) -> (a -> b)`
⊗ `(a -> b -> b) -> a -> b -> b`

---

8. La evaluación de `foldl (\xs x -> (x:x:xs)) [] ['a','b','c']` produce como resultado

◯ `['a','a','b','b','c','c']`
⊗ `'ccbbaa'`
◯ Ninguno de los anteriores.

---

9. Dados los términos $t_1 = p(g(Y), f(W, Z, Y))$, $t_2 = p(X, f(g(Z), h(Y), a))$
◯ No son unificables por un error del occur check.
⊗ $[X/g(Y), Y/a, W/g(h(a)), Z/h(Y)]$ es un unificador de t_1, t_2.
◯ El unificador más general de t_1, t_2 es $[X/g(a), Y/a, W/g(Z), Z/h(a)]$.

---

10. Considérense los siguientes tres objetivos Prolog:

```
X is 1+1, X = 1+1.
X = 1+1, X =:= 1+1.
X is 1+1, X == 1+1.
```

◯ Solo uno tiene éxito y da como solución `X = 2`.
⊗ Solo uno tiene éxito y da como solución `X = 1+1`.
◯ Las dos anteriores son falsas.

PREGUNTAS DE DESARROLLO

1. **[0,5 puntos]** Elimina la lista extensional de la siguiente definición utilizando las funciones del preludio de Haskell `map`, `filter` y `concat`, pero no otras funciones auxiliares.

```
f q n = [(x,y) | x <- [1..n], y <- [x..100], q y]
```

```
concat (map g [1..n] where g x = map (\y ->(x,y)) (filter q [x..100])
```

2. **[0,5 puntos]** Determina de manera razonada cuál es el tipo de la función `f` definida por la ecuación
`f x y z = if (\x -> y x) z then x (y z) else y x`

```
f :: Tx -> Ty -> Tz -> T
if :: Bool -> T -> T -> T ⇒ (\x -> y x) z :: Bool,  x (y z) :: T,  y x :: T
(\x -> y x) z :: Bool ⇒ y z :: Bool ⇒ Ty = Tz -> Bool
y x :: T ,  Ty = Tz -> Bool ⇒ T = Bool,  Tx = Tz
x (y z) :: T ,  y z :: Bool,  T = Bool ⇒ Tx = Bool -> Bool
Tx = Tz, Tx = Bool ->Bool ⇒ Tz = Bool -> Bool
Tz = Bool -> Bool, Ty = Tz -> Bool ⇒ Ty = (Bool -> Bool) -> Bool
```

Por tanto: `f :: (Bool -> Bool) -> ((Bool -> Bool) -> Bool) -> (Bool -> Bool) -> Bool`

3. Programa en Haskell sin olvidar las anotaciones de tipo en todas las funciones que definas.

   - **[0,25 puntos]** Define un tipo polimórfico `Par a b` como un alias del tipo par de elementos, con primera componente de tipo `a` y lsegunda de tipo `b`. Utilizando este tipo, define un tipo de datos polimórfico `ConjPar a b` para representar conjuntos cuyos elementos son de tipo `Par a b` (puedes apoyarte en el tipo lista).

     ```
     type Par a b = (a,b)
     data ConjPar a b = Cj [Par a b]
     ```

   - **[0,5 puntos]** Programa en Haskell la función `eqConj` que, dados dos argumentos de tipo `ConjPar a b` compruebe que efectivamente representan conjuntos y que determine si son iguales. Recuerda que un conjunto no puede tener elementos repetidos y que el orden de sus elementos es irrelevante.

     ```
     eqConj :: (Eq a,Eq b) => (ConjPar a b) -> (ConjPar a b) -> Bool
     eqConj c1 c2 = esConjPar c1 && esConjPar c2 && iguales c1 c2

     esConjPar :: (Eq a,Eq b) => (ConjPar a b) -> Bool
     esConjPar (Cj []) = True
     esConjPar (Cj (x:xs)) = not (elem x xs) && esConjPar (Cj xs)

     iguales :: (Eq a,Eq b) => (ConjPar a b) -> (ConjPar a b) -> Bool
     iguales c1 c2 = contenido c1 c2 && contenido c2 c1

     contenido :: (Eq a,Eq b) => (ConjPar a b) -> (ConjPar a b) -> Bool
     contenido (Cj []) _ = True
     contenido (Cj (x:xs)) (Cj ys) = elem x ys && contenido (Cj xs) (Cj ys)
     ```

   - **[0,25 puntos]** Declara `ConjPar a b` como instancia de la clase `Eq` de modo que `==` coincida con la igualdad de conjuntos.

     ```
     instance (Eq a,Eq b) => Eq (ConjPar a b) where c1 == c2 = eqConj c1 c2
     ```

4. **[0,5 puntos]** Programa en Haskell la siguiente función, anotando su tipo y sin utilizar recursión explícita, sino alguna función de la familia `fold`.

   `multiplos3 xs = (n, ys)`, donde `xs` es una lista de enteros, `n` es el número de elementos de `xs` que son múltiplos de 3 y `ys` es la lista de las posiciones en las que aparecen dichos números dentro de `xs`.
   Ejemplo: `multiplos3 [-2,10,15,9] = (2,[3,2])`.

```
multiplos3 :: [Int] -> (Int,[Int])
multiplos3 xs = snd (foldl f (0,(0,[])) xs)
   where f (i,(n,ys)) x
           | (x `mod` 3) == 0 = (i+1,(n+1,i:ys))
           | otherwise        = (i+1,(n,ys))
```

5. **[0,5 puntos]** Programa en Haskell la siguiente función, indicando los tipos de todas las funciones que definas:

   `digitParImpar x = (ps,is)`, donde `x` es un entero positivo, `ps` es la lista de los dígitos pares de `x` e `is` es la lista de los dígitos impares de `x`.

```
digitos :: Integer -> [Integer]
digitos x
 | abs x < 10 = [x]
 | otherwise = (x `mod` 10) : (digitos (div x 10))

par :: Integer -> Bool
par x = (x `mod` 2) == 0

digitParImpar :: Integer -> ([Integer],[Integer])
digitParImpar x = (filter par xs,filter (not.par) xs)
     where xs = digitos x
```

6. **[1 punto]** Programa en Prolog los siguientes predicados:

   a) `nobasico(T)` $\leftrightarrow$ `T` es un término no básico, es decir, contiene variables. Puedes utilizar los predicados predefinidos de Prolog `atomic/1`, `var/1`, `=../2`.

```
nobasico(T) :- var(T),!.
nobasico(T) :- atomic(T), !, fail.
nobasico(T) :- T =.. L, nobasiclist(L).

nobasiclist([T|_]) :- nobasico(T), !.
nobasiclist([_|L]) :- nobasiclist(L).
```

   b) `simetrico T` $\leftrightarrow$ `T` es un árbol binario simétrico.

```
simetrico(void) :- !.
simetrico(arbol(_,HI,HD) :- simetricos(HI,HD).

simetricos(void,void) :- !.
simetricos(arbol(X,HI1,HD1),arbol(X,HI2,HD2)) :-
    simetricos(HI1,HD2),
    simetricos(HD1,HI2).
```

7. **[1 punto]** Dado el programa lógico:

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

   Construye el árbol de resolución del objetivo `q(X,Y)`, `r(f(X))`.