Programación Funcional

Curso 2016/2017. Ejercicios – Sesión práctica (Lote 1)

Esto es un repertorio de actividades sugeridas para la sesión de prácticas

1. - El ordenador más potente del mundo, el chino Sunway TaihuLight, exhibe una potencia de cómputo de 93 petaFlops (1 petaFlop = 10^{15} Flop). Calcula cuántas operaciones en coma flotante habría realizado hasta el momento actual dicho ordenador si hubiese estado operando desde el principio de los tiempos, sabiendo que la edad del universo viene a ser de unos 13700 millones de años. Obtén el resultado como `Int`, `Integer`, `Double`.
   - Calcula cuántos años hay en 10^{10} segundos (*supón que todos los años tienen 365 días; en otro momento puedes hacerlo teniendo en cuenta bisiestos*).
   - Calcula cuántos años enteros, días restantes enteros, horas restantes enteras, minutos restantes enteros y segundos restantes hay en 10^{10} segundos.
   - Generaliza alguno de los apartados anteriores (por ejemplo, los dos anteriores) convirtiendo el número de años (o número de segundos, según el caso) en parámetro de una función. Indica en el programa los tipos de las funciones.

2. Escribe un programa Haskell en el que definas **usando notación currificada** las siguientes funciones que vienen escritas en notación matemática clásica:

$$
\begin{aligned}
&f(x,y) = 2x - y * x \\
&g(x) = f(f(2,x), f(x,1)) \\
&h(x,y,z) = f(f(x+2y, g(3)), 5 - g(z) - y) \\
&i(x,y) = \begin{cases} x-y & \text{si } x \geq y > 0 \\ 0 & \text{si } 0 < x < y \\ y-x & \text{en otro caso} \end{cases}
\end{aligned}
$$

   Pon en el intérprete expresiones para calcular lo que en notación no currificada serían las expresiones: $f(7,4)$, $f(f(1,1), g(-2))$, $i(2,2) - i(g(1),0)$. Procura no poner paréntesis de más.

   Haz algunas variantes de las definiciones de arriba:

   - En el caso de la función i, usando `if _ then _ else` o usando guardas
   - En lugar de usar de modo infijo los operadores aritméticos `+,-,>,...` (que es lo usual), escribirlos en forma prefija `(+),(-),(>),...`

3. Programa las siguientes funciones:

   - `tresIguales x y z` $=_{def}$ `True` si `x, y ,z` son iguales, `False` en otro caso.
   - `distintos x y z` $=_{def}$ `True` si `x, y ,z` son distintos dos a dos, `False` en otro caso.

   Descubre (con el comando `:t` del intérprete) el tipo que infiere Haskell para ellas. Descubre (con el comando `:info` del intérprete) la prioridad y asociatividad de los operadores infijos que estés usando, y tenlo en cuenta para no poner paréntesis innecesarios. Reescribe las definiciones usando notación prefija para los operadores infijos *(recuerda, esto se consigue rodeándo los operadores por paréntesis)*.

4. Programa de manera recursiva (usando `div` y `mod`) las siguientes funciones:

   - `digitos x` $=_{def}$ número de dígitos del número entero `x`
   - `reduccion x`$=_{def}$ resultado del proceso de sumar los dígitos del entero `x`, sumar los dígitos del resultado obtenido, y así sucesivamente hasta obtener un número menor que 10. La reducción de un entero negativo es la de su valor absoluto.

5. Programa las siguientes funciones, declarando sus tipos:

   - `perm n` = número de permutaciones de `n` elementos
   - `var n m` = número de variaciones de `n` elementos tomados de `m` en `m`

- `comb n m` = número de combinaciones de `n` elementos tomados de `m` en `m`

Hazlo de dos modos: *(i)* usando la función factorial (que has de definir) para las tres funciones; *(ii)* usando recursión para `var` y lo que quieras para las demás.

6. Programa como función de aridad uno la sucesión de Fibonacci, definida por $a_0 = 1, a_1 = 1, a_{n+2} = a_n + a_{n+1}$. Comprueba empíricamente su complejidad y, si te sale muy alta, arréglalo programándola de otro modo. Para la comprobación empírica utiliza el comando `:set +s` en el intérprete de Haskell, que muestra indicadores de tiempo y memoria de la evaluación de las expresiones que se sometan al intérprete.

7. Programa, mediante ecuaciones con ajuste de patrones, todos los conectivos booleanos habituales (conjunción, disyunción, negación, implicación, equivalencia, disyunción exclusiva). Hazlo en varias variantes, como se va indicando en los siguientes apartados:

   - Variantes sintácticas:
     - Sin usar operadores infijos para los nombres de las funciones (por ejemplo, llamándolas `no, y, o`, etc).
     - Declarando tus propios operadores infijos (para las funciones de aridad 2), como pueden ser `&&&` o `|||`.

     Evalúa, en ambas versiones, la expresión booleana que en notación matemática usual escribiríamos $(True \vee False \vee \neg True) \wedge (\neg(True \wedge False) \rightarrow True)$
   - Usando la negación y la conjunción para definir el resto de conectivos.

8. Más variaciones sobre la conjunción booleana (y lo mismo se puede hacer para la disyunción).

   - Define la conjunción booleana por ajuste de patrones, pero de cuatro o cinco formas diferentes, cambiando el número de ecuaciones, o las combinaciones de patrones `True, False, x, ...` en cada ecuación, o el orden de ecuaciones, etc. Para que coexistan todas definiciones en el mismo programa, dales nombres (o usa operadores) diferentes.
   - Comprueba que todas tus definiciones del primer apartado dan el resultado correcto para todas las combinaciones posibles de `True, False` como argumentos.
   - Piensa lo que ocurre (y haz alguna comprobación) cuando alguno de los argumentos está indefinido (o sea, su evaluación da error o no termina). ¿Se comportan igual todas las definiciones que has puesto? Para obtener expresiones indefinidas polimórficas puedes usar cualquiera de las siguientes expresiones (en ellas uso `bot` como abreviatura de *bottom*, cuyo símbolo habitual es $\perp$, que se suele usar para representar el valor indefinido en dominios semánticos).
     - `bot = undefined` (`undefined` *es una función de aridad cero del Prelude cuya ejecución da error*)
     - `bot' | False = bot'`
     - `bot'' = error "Valor indefinido"`
     - `bot''' = head []`
     - `bot'''' = bot''''`

     Comprueba que, en efecto, todas estas funciones de aridad 0 tienen tipo polimórfico `a` y que su evaluación da error o no termina (es decir, que en términos abstractos $[\![e]\!] = \perp$ para cada una de esas expresiones e).
   - Indica en qué argumentos son estrictas las distintas variantes de la conjunción que has programado. Si resulta que todas ellas son estrictas en el primer argumento, programa una variante más que no lo sea.

9. Programa una función f de tres argumentos que sea:

   (i) Estricta en el primer argumento.

   (ii) No estricta ni en el segundo ni en el tercer argumento.

   (iii) *Conjuntamente estricta* en el segundo y tercer argumento, es decir, tal que $[\![f\ x\ \perp\ \perp]\!] = \perp$, para todo valor x.