Programación Funcional

Curso 2023/2024. Ejercicios – Sesiones de laboratorio (Lote 1)

Esto es un repertorio de actividades sugeridas para las primeras sesiones de laboratorio

1. - Desde junio de 2022, el ordenador más potente del mundo es el Frontier del Oak Ridge National Laboratory (Tennessee, EEUU), con una potencia de cómputo de 1194 *petaFlops* (1 petaFlop $= 10^{15}$ Flop). Calcula cuántas operaciones en coma flotante habría realizado hasta el momento actual *Frontier* 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, declarando sus tipos:

   - `digitos x` = número de dígitos del número entero `x`.
   - `reduccion` x= 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.
   - `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`

4. Programa la siguiente propiedad (es decir, función que devuelve `True` o `False`)

   `primo n` $\leftrightarrow$ n es un número primo

   Calcula el primer número primo mayor que 10000. Calcula el centésimo primo. Calcula la lista de los 100 primeros primos. Escribe una expresión que nos dé la lista de los infinitos primos.
   Nota: revisa este ejercicio según vayas aprendiendo más cosas, como por ejemplo funciones de orden superior, listas, listas intensionales.

5. 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 (por ejemplo, si tu programa no es capaz de calcular a_100) 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. Entre las formas de calcular números de Fibonacci considera la posibilidad de obtener directamente la lista de los números de Fibonacci.

6. 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.

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

   - Define la conjunción booleana por ajuste de patrones, como antes, 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.

8. Programa una función f de tres argumentos que verifique las tres condiciones siguientes:

   (i) Ser estricta en el primer argumento.

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

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

9. Programa la función `media` que calcula la media aritmética de una lista de números, usando para ello la función `length` que calcula el número de elementos de una lista. ¿Tienes algunos problemas con los tipos? Si es así, resuélvelos bien definiendo tu propia versión de `length` con un tipo más flexible, bien usando la función de conversión `fromIntegral`.

10. Programa las siguientes funciones, indicando sus tipos:
    `last xs` = último elemento de la lista no vacía `xs`
    `init xs` = todos menos el último elemento de la lista no vacía `xs`
    `initLast xs` = `(init xs,last xs)`
    `concat xss` = resultado de concatenar los elementos de la lista de listas `xss`
    `take n xs` = lista de los `n` primeros elementos de `xs`
    `drop n xs` = resultado de eliminar los `n` primeros elementos de `xs`
    `splitAt n xs` = `(take n xs, drop n xs)`
    `nub xs` = resultado de eliminar los elementos repetidos de la lista `xs`
    `sort xs` = resultado de ordenar la lista `xs` (usa diferentes métodos)
    `and bs` = resultado de hacer la conjunción de todos los elementos de `bs`
    `or bs` = resultado de hacer la disyunción de todos los elementos de `bs`
    `sum xs` = resultado de sumar todos los elementos de `xs`
    `product xs` = resultado de multiplicar todos los elementos de `xs`
    `lmedia xss` = longitud media de los elementos de la lista de listas `xss`

11. Programa la función `reverse`, que devuelve la inversa de una lista, de estas dos maneras:

    a) Por la recursión más obvia, usando la concatenación de listas `++`. Comprueba empíricamente que la complejidad es cuadrática en la longitud de la lista. *(Para obtener datos sobre tiempo y memoria usa el comando* `:set +s` *del intérprete)*

    b) Utilizando un parámetro auxiliar acumulador, consiguiendo complejidad lineal. Haz comprobaciones empíricas.

12. Piensa y/o prueba en el intérprete cuáles de las siguientes expresiones tardarán poco (digamos centésimas o milésimas de segundos), regular (digamos décimas o segundos) o mucho (digamos toda una vida) en ser evaluadas. En el tercer caso, no te esperes toda tu vida, sino que estima cuánto va a tardar la evaluación o, al menos, interrumpe el cómputo.

    - `last [1..10^5]`
    - `last [1..10^7]`
    - `last [1..10^20]`
    - `head [1..10^20]`
    - `last [10^20..1]`
    - `head (tail [1..10^20])`
    - `length [1..10^20]`
    - `last (take (10^7) [1..10^20])`
    - `head (take (10^7) ([1..100] ++ [1..10^20]))`
    - `last (take 100 ([1..10^20] ++ [1..100]))`
    - `last (drop 100 ([1..10^20] ++ [1..100]))`
    - `head (drop (10^7) ([1..10^20] ++ [1..100]))`
    - `[1..10^7]==[1..10^7]`
    - `[1..10^20]==[1..10^20]`
    - `[1..10^20]==[1..10^20+1]`
    - `[1..10^20]==[2..10^20]`
    - `head (reverse [1..10^7])`
    - `last (reverse [1..10^7])`
    - `reverse [1..10^20] == reverse [1..10^20+1]`