Programácion Funcional

Trabajo Práctico Nro. 1

"Lo que debemos aprender a hacer lo aprendemos haciéndolo"

Aristóteles, Ética Nicomaquea II (325 A.C.)

Para realizar las prácticas de la materia, usted debe considerar las siguientes indicaciones:

- **Ejercicios complementarios**
  Los ejercicios que estén en esta sección de cada práctica son complementarios y la realización de los mismos no es esencial durante la cursada. Se recomienda intentar pensarlos y realizarlos para la EPI.
- ⊛
  Indica que es un ejercicio de complejidad mayor a los demás ejercicios. No debe preocuparse si no puede resolverlo, aunque es recomendable que trate de buscar una solución y eventualmente consulte con el ayudante de la comisión.
- Reutilizar código es una forma eficiente de disminuir el tiempo necesario para programar, por eso siempre que lo crea adecuado reutilice las funciones que haya definido anteriormente.

Temas: Introducción a la sintaxis de Haskell y al ambiente Hugs.

Bibliografía relacionada:

- John Hughes. Why Functional Programming Matters? Computer Journal 32 (2), 1989.
- Simon Thompson. The craft of Functional Programming. Addison Wesley, 1996. Cap. 1.
- L.C. Paulson. ML for the working programmer. Cambridge University Press, 1996. Cap. 1.
- Bird, Richard. Introduction to funtional programming using Haskell. Prentice Hall, 1998 (Second Edition). Cap. 1 y 2.

1. Definir las siguientes funciones:

`seven,`	que dado cualquier valor, devuelve 7.
`sign,`	que dado un entero devuelve 1 si es positivo, -1 si es negativo y 0 si es cero (usando guardas y sin usarlas).
`absolute,`	la función valor absoluto (usando `sign` y sin usarla).
`and',or',not',xor',`	las operaciones booleanas estándar, sin utilizar las funciones predefinidas en el Hugs.
`dividesTo,`	toma dos números y dice si el primero divide al segundo (sugerencia: utilizar la función `mod`, que devuelve el resto de la división entera).
`isMultiple,`	toma dos números y dice si el primero es múltiplo del segundo.
`isCommonDivisor,`	toma un número y **un par de números**, y verifica si el primero es divisor común de los otros dos.
`isCommonMult,`	toma un número y **un par de números**, y verifica si el primero es múltiplo común de los otros dos.
`swap,`	toma un par ordenado, y devuelve un par ordenado con sus componentes en orden inverso.

2. Reescribir las siguientes funciones sin el uso de `let`, `where` o `if then else` cuando sea posible.

   a) `f x = let (y,z) = (x,x) in y`

   b) `greaterThan (x,y) = if x > y then True else False`

   c) `f (x,y) = let z = x + y in g (z,y) where g (a,b) = a - b`

3. Redefinir la función `power4` de dos formas diferentes

```
power4 x = let sqr y = y * y
           in sqr (sqr x)
```

4. Definir la función `fib` que calcula el n-ésimo término de la sucesión de Fibonacci, definida esta última por:
   $T(0) = T(1) = 1$
   $T(n) = T(n-1) + T(n-2)$

5. Enumere algunas propiedades deseables en los programas.

6. Según su apreciación, ¿qué caracteriza al paradigma Funcional?

7. Utilizar el intérprete Hugs para implementar y probar las funciones definidas.

Ejercicios complementarios

9. Un año es saurónico si sus últimas dos cifras son divisibles por 4 y distintas a 00; en caso contrario sólo se considera saurónico si es divisible por 400. Ejemplos:

   - 1996 es saurónico porque 96 es divisible por 4 y es distinto a 00
   - 2003 no es saurónico porque 03 no es divisible por 4
   - 1900 no es saurónico porque termina en 00 y no es divisible por 400
   - 2000 es saurónico porque termina en 00 y es divisible por 400

   Definir una función que determine si un año es saurónico o no.

10. Definir la función `sort3`, que recibe tres números y los retorna ordenados (el primero menor o igual que los demás; el segundo menor o igual que el tercero). Implemente al menos tres soluciones distintas.