Informática – Haskell – Matemáticas – Curso 2004-2005
Pepe Gallardo – Universidad de Málaga

Práctica 1

1.1 Define un operador (|||) que devuelva *True* si y solo si tan solo uno de sus argumentos es *True*. Por ejemplo:

? $(3 > 2)$ ||| $(2 > 5)$
True :: *Bool*

? $(3 > 2)$ ||| $(5 > 2)$
False :: *Bool*

1.2 Define el operador anterior con tan solo una ecuación. (AYUDA: Puedes usar la función predefinida *not*).

1.3 Sea la siguiente función para calcular el máximo de sus tres argumentos:

$$
\begin{aligned}
&máximo3 :: Integer \to Integer \to Integer \to Integer\\
&máximo3\ x\ y\ z\\
&\quad |\ x > y \;\&\&\; x > z = x\\
&\quad |\ y > x \;\&\&\; y > z = y\\
&\quad |\ otherwise \qquad\quad\; = z
\end{aligned}
$$

Pruébala con los siguientes valores y comprueba que devuelve el resultado adecuado:

```
6 4 1
6 6 6
2 6 6
2 2 6
```

Encuentra un contraejemplo que demuestre que la definición no es correcta, es decir, tres valores de los argumentos para los cuales no se devuelva el máximo.

Redefine la función de modo que sea correcta, es decir, que devuelbva el resultado adecuado para cualquier combinación de argumentos.

1.4 Define tres funciones ($max2$, $max3$ y $max4$) para calcular el máximo de dos, tres y cuatro enteros. (AYUDA: usa $max2$ para definir $max3$ y $max4$).

? $max2\ 5\ 4$
5 :: *Integer*

? $max3\ 5\ 7\ 4$
7 :: *Integer*

? $max4\ 5\ 4\ 9\ 5$
9 :: *Integer*

1.5 Define una función *tresDiferentes* que devuelva *True* si y solo si sus tres argumentos enteros son distintos. Por ejemplo:

? *tresDiferentes* 1 2 3
True :: *Bool*

? *tresDiferentes* 3 4 3
False :: *Bool*

1.6 Define una función *cuatroIguales* que devuelva *True* si y solo si sus cuatro argumentos enteros son todos iguales.

1.7 Define una función *media*3 que devuelva la media de los tres reales que toma como argumento. Usando esta función, define la función *cuántosSobreMedia*3 que tome tres reales y devuelva cuántos de estos son estrictamente mayores a su media.

1.8 Usando la suma, define una función producto que calcule el producto de dos enteros.

? *producto* 5 4
20 :: *Integer*

? *producto* (-5) 4
-20 :: *Integer*

1.9 La raíz cuadrada entera de un número entero y positivo n es el mayor entero cuyo cuadrado es menor o igual que n. Por ejemplo, las raíces cuadradas enteras de 15 y 16 son 3 y 4, respectivamente. Define una función recursiva que calcule la raíz cuadrada entera de su argumento (sin usar *sqrt*).

? *raíz* 15
3 :: *Integer*

AYUDA: Define primero una función *buscarDesde* tal que *buscarDesde* i n devuelva el primer entero j mayor o igual que i tal que $j^2 \leq n$ y $(j+1)^2 > n$.

1.10 Supón que queremos calcular el valor de la potencia 2^n. Si n es par, es decir si $n = 2m$, entonces $2^n = 2^{2m} = (2^m)^2$. Si n es impar, es decir si $n = 2m+1$, entonces $2^n = 2^{2m+1} = 2\,(2^m)^2$

Define una función recursiva *pot*2 para calcular potencias de 2 usando las propiedades anteriores.

? *pot*2 4
16 :: *Integer*

? *pot*2 3
8 :: *Integer*

Práctica 2

2.1 Define una función de orden superior *opera* que tome como parámetros un operador y una lista y devuelva el resultado de operar de derecha a izquierda todos los elementos de la lista con dicho operador, es decir,

$$opera\ (\otimes)\ [x_1,\ x_2,\ \ldots,\ x_{n-1},\ x_n] \Longrightarrow \qquad x_1 \ \otimes\ (x_2\ \otimes\ (\ldots\ x_{n-1}\ \otimes\ x_n))$$

Por ejemplo,

? *opera* $(+)$ $[1,2,3,4]$
10 :: *Integer*

? *opera* $(*)$ $[1,2,3,4]$
24 :: *Integer*

¿Cuál es el tipo polimórfico de esta función?.

2.2 Define una función de orden superior *opera*2 que tome como parámetros un operador y una lista y devuelva una lista con el resultado de operar de dos en dos los elementos de la lista con dicho operador, es decir,

$$opera2\ (\otimes)\ [x_1,\ x_2,\ \ldots,\ x_{n-1},\ x_n] \Longrightarrow \qquad [\ x_1\ \otimes\ x_2,\ \ldots\ x_{n-1}\ \otimes\ x_n]$$

Por ejemplo,

? *opera*2 $(+)$ $[1,2,3,4,5,6]$
$[3,7,11]$ :: *Integer*

? *opera*2 $(*)$ $[1,2,3,4]$
$[2,12]$ :: *Integer*

¿Cuál es el tipo polimórfico de esta función?. Considera en tu solución el caso de que la lista argumento tenga longitud impar.

2.3 Escribe una función de orden superior *filtra* con tipo :: (*Integer* → *Bool*) → [*Integer*] → [*Integer*] que tome una función y una lista de enteros, y devuelva otra lista con los elementos de la lista original para los que el resultado de la función sea *True*:

? *filtra even* $[0,1,2,0,3,4]$
$[0,2,0,4]$ :: [*Integer*]

? *filtra* (>1) $[0,1,2,0,3,4]$
$[2,3,4]$ :: [*Integer*]

2.4 Escribe otra función de orde superior *rechaza* con tipo :: (*Integer* → *Bool*) → [*Integer*] → [*Integer*] que tome una función y una lista de enteros, y de-

vuelva otra lista con los elementos de la lista original para los que el resultado de la función sea *False*:

> ? *rechaza even* $[0, 1, 2, 0, 3, 4]$
> $[1, 3]$ :: $[Integer]$

Además de una definición recursiva, da otra definición de *rechaza* usando *filtra*, *not* y el operador de composición (.).

2.5 Escribe una función *divideA* con tipo :: *Integer* → *Integer* → *Bool*, de modo que *divideA x y* sea *True* si y es un divisor de x:

> ? *divideA* 10 2
> *True* :: *Bool*
>
> ? *divideA* 10 3
> *False* :: *Bool*

2.6 Escribe una función *divisores* con tipo :: *Integer* → [*Integer*], de modo que *divisores x* devuelva una lista con los divisores del número x:

> ? *divisores* 10
> $[1, 2, 5, 10]$ :: $[Integer]$

Para definir esta función usa las funciones *filtra* y *divideA*. Puedes definir una lista que contenga todos los enteros desde 1 hasta x con $[1..x]$. Por ejemplo:

> ? *filtra even* $[1..5]$
> $[2, 4]$ :: $[Integer]$

2.7 Escribe una función *mcd* que calcule el máximo común divisor dos números naturales. Para ello puedes utilizar el algoritmo de Euclides, que consiste en restar al mayor de los dos números el menor, y repetir este proceso hasta que uno de los números sea cero. En este momento, el otro número (el que no es cero) es el máximo común divisor. Por ejemplo, para calcular *mcd*10 8 procederíamos del siguiente modo:

  mcd 10 8
$\Longrightarrow$ {restando 8 a 10}
  mcd 2 8
$\Longrightarrow$ {restando 2 a 8}
  mcd 2 6
$\Longrightarrow$ {restando 2 a 6}
  mcd 2 4
$\Longrightarrow$ {restando 2 a 4}
  mcd 2 2
$\Longrightarrow$ {restando 2 a 2}
  mcd 0 2
$\Longrightarrow$ {un argumento es cero}
  2

Práctica 3

Sea el siguiente tipo para representar números naturales:

data *Nat* = *Cero* | *Suc Nat* **deriving** *Show*

3.1 Haz este tipo instancia de la clase *Eq*, es decir, define la igualdad para el tipo *Nat* (no derives la instancia). Para probar tu definición evalúa algunos ejemplos como

```
? Cero == Cero
True :: Bool

? Suc (Suc Cero) == Cero
False :: Bool

? Cero /= Suc (Suc Cero)
True :: Bool

? Suc Cero == Suc (Suc Cero)
False :: Bool
```

3.2 Haz el tipo *Nat* instancia de la clase *Ord*, es decir, define un orden para el tipo (no derives la instancia). Prueba distintos ejemplo, intentado abarcar todos los casos posibles, por ejemplo,

```
? Cero < Cero
False :: Bool

? Suc Cero <= Suc (Suc Cero)
True :: Bool

? max (Suc Cero) (Suc (Suc Cero))
Suc (Suc Cero) :: Nat
```

3.3 Haz el tipo *Nat* instancia de la clase *Num*, es decir, define las operaciones de esta clase para el tipo. Prueba distintos ejemplos, intentado abarcar todos los casos posibles, por ejemplo,

```
? Cero + Cero
Cero :: Nat

? Suc (Suc Cero) * Suc (Suc Cero)
Suc (Suc (Suc (Suc Cero))) :: Nat

? fromInteger 3 + Suc Cero
Suc (Suc (Suc (Suc Cero))) :: Nat
```

3.4 La clase predefinida *Enum* es la clase de los tipos enumerables, es decir tipos secuencialmente ordenados. Esta clase se encuentra predefinida como

```
class Enum a where
  succ, pred :: a → a
  toEnum     :: Int → a
  fromEnum   :: a → Int
  ...
```

Las funciones *succ* y *pred* devuelven el predecesor y el sucesor de un valor respectivamente; la función *toEnum* permite convertir un entero en un valor del tipo con el que se va a realizar la instancia, mientras que *fromEnum* es la recíproca de *toEnum*. Para realizar una instancia de esta clase, basta con definir las funciones *toEnum* y *fromEnum* ya que las demás están definidas por defecto. Realiza una instancia de esta clase con el tipo *Nat*, es decir, define en la instancia las funciones *toEnum* :: *Int* → *Nat* y *fromEnum* :: *Nat* → *Int*. Prueba distintos ejemplos

```
? pred (Suc Cero)
Cero :: Nat

? fromEnum (Suc Cero)
1 :: Int
```

3.5 La clase predefinida *Integral* es la clase de los tipos que definen las funciones *div* y *mod*. Realiza una instancia de la clase Integral con el tipo *Nat* en la que definas las funciones *div* y *mod* para naturales.

Nota. Debido a ciertas restricciones que no hemos estudiado en el curso, para poder realizar una instancia de la clase *Integral* es necesario realizar otra de la clase *Real*; para esto basta con que escribas la siguiente línea en el programa:

```
instance Real Nat
```

Prueba las funciones que has definido, por ejemplo,

```
? div (toEnum 4) (Suc (Suc Cero))
(Suc (Suc Cero)) :: Nat

? mod (toEnum 4) (Suc (Suc Cero))
Cero :: Nat
```

Práctica 4

4.1 Define una función *esNeutroDer* que tome una función (u operador) de dos argumentos, un dato y una lista que representará un conjunto, y compruebe si el dato dado es elemento neutro por la derecha de la operación dada en el conjunto (lista) indicado. Por ejemplo,

```
? esNeutroDer (+) 0 [1..10]
True :: Bool
```

? *esNeutroDer* (+) 4 [1..10]
False :: *Bool*

? *esNeutroDer div* 1 [1..10]
True :: *Bool*

ya que para la suma, el 0 es elemento neutro por la derecha en el conjunto [1..10] y para la división el 1 es elemento neutro por la derecha en el conjunto [1..10]. Indica además el tipo polimórfico de la función *esNeutroDer*.
Sugerencia. Para calcular *esNeutroDer f e xs* tendrás que comprobar que $\forall\, x \in xs$ **.** $f\ x\ e == x$. Para que x recorra toda la lista *xs* puedes usar una lista por comprensión.

4.2 Define una función *esNeutroIzq*, dando su tipo polimórfico, que tome una función (u operador) de dos argumentos, un dato y una lista que representará un conjunto, y compruebe si el dato dado es elemento neutro por la izquierda de la operación dada en el conjunto (lista) indicado. Por ejemplo,

? *esNeutroIzq* (+) 0 [1..10]
True :: *Bool*

? *esNeutroIzq* (+) 4 [1..10]
False :: *Bool*

4.3 Define una función *esNeutro*, dando su tipo polimórfico, que tome una función (u operador) de dos argumentos, un dato y una lista que representará un conjunto, y compruebe si el dato dado es elemento neutro de la operación dada en el conjunto (lista) indicado. Por ejemplo,

? *esNeutro* (+) 0 [1..10]
True :: *Bool*

? *esNeutro div* 1 [1..10]
False :: *Bool*

4.4 Define una función *esConmutativa*, dando su tipo polimórfico, que tome una función (u operador) de dos argumentos y una lista que representará un conjunto, y compruebe si la función dada es conmutativa en el conjunto dado. Por ejemplo,

? *esConmutativa* (+) [1..10]
True :: *Bool*

? *esConmutativa* (−) [1..10]
False :: *Bool*

Sugerencia. Para calcular *esConmutativa f xs* tendrás que comprobar que $\forall\, x \in xs$ **.** $\forall\, y \in xs$ **.** $f\ x\ y == f\ y\ x$. Para que x e y recorran toda la lista *xs* puedes usar una lista por comprensión con dos generadores.

4.5 Define una función *esAsociativa*, dando su tipo polimórfico, que tome una función (u operador) de dos argumentos y una lista que representará un conjunto, y compruebe si la función dada es asociativa en el conjunto dado. Por ejemplo,

? *esAsociativa* (+) [1..10]
True :: *Bool*

? *esAsociativa* (−) [1..10]
False :: *Bool*

sta *xs* puedes usar una lista por comprensión con tres generadores.

4.6 Define una función *neutro*, dando su tipo polimórfico, que tome una función (u operador) de dos argumentos y una lista que representará un conjunto, y devuelva una lista con los elementos neutros de la función para el conjunto dado. Por ejemplo,

? *neutro* (+) [−10..10]
[0] :: [*Integer*]

? *neutro* (∗) [−10..10]
[1] :: [*Integer*]

? *neutro* (+) [1..10]
[ ] :: [*Integer*]

Práctica 5

Se define el siguiente tipo para representar matrices en Haskell,

type *Matriz* = [[*Float*]]

de modo que una matriz es una lista de listas, donde cada sublista representa una fila de la matriz. Por ejemplo, la matriz

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

se representaría en Haskell como [[1, 2, 3], [4, 5, 6]] .

5.1 Define dos funciones *filas* y *columnas* que dada una matriz devuelvan cuántas filas y columnas tiene:

? *filas* [[1, 2, 3], [4, 5, 6]]
2 :: *Integer*

? *columnas* [[1, 2, 3], [4, 5, 6]]
3 :: *Integer*

5.2 Define un operador $(<+>)$ que tome dos matrices y devuelva la matriz suma:

```
? [ [1,2], [3,4] ] <+> [ [10,20], [30,40] ]
[ [11,22], [33,44] ]  :: Matriz
```

5.3 Define una función traspuesta que dada una matriz devuelva su traspuesta (la que se obtiene al cambiar filas por columnas):

```
? trasp [ [1,2,3], [4,5,6] ]
[[1,4],[2,5],[3,6]]  :: Matriz
```

5.4 **(Difícil)** Define un operador $(<*>)$ que calcule el producto de dos matrices.

Práctica 6

El matemático George Boole (1815-1864) pretendía determinar las leyes que rigen el pensamiento humano. Para ello, diseñó a mediados del siglo XIX el álgebra de Boole. Se trata de una estructura algebraica que captura la esencia de las operaciones lógicas Y, O y No.

Una proposición en este álgebra consiste en una expresión en la que pueden aparecer las constantes *True* y *False* (para representar la noción de verdadero y falso), variables con dominio $\{True, False\}$, la conjunción de dos proposiciones (representada como $p_1 \wedge p_2$), que será cierta si ambas proposiciones son ciertas, la disyunción de dos proposiciones (representada como $p_1 \vee p_2$), que será cierta si al menos una de las proposiciones es cierta y la negación de una proposición (representada como *no p*) que será cierta si la proposición p es falsa.

Algunos ejemplos de proposiciones son $True \wedge a$, $a \wedge no\ a$ y $(a \wedge b) \vee no\ c$. Con objeto de ahorrar paréntesis a la hora de escribir proposiciones, se considerará que *no* tiene la máxima prioridad, seguida de la conjunción y por último la disyunción.

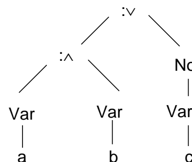
Las siguientes definiciones de tipo pueden ser usadas para representar proposiciones de álgebra de Boole en Haskell:

```haskell
type Nombre = String
infixl :/\ 7
infixl :\/ 6
data Prop = Const Bool
          | Var Nombre
          | No Prop
          | Prop :/\ Prop
          | Prop :\/ Prop  deriving Show
```

con lo que los ejemplos anteriores se definirían como:

$e1, e2, e3 \ :: \ Prop$
$e1 \ = \ Const\ True\ :/\backslash\ Var$ "a"
$e2 \ = \ Var$ "a" $:/\backslash\ No\ (Var$ "a"$)$
$e3 \ = \ (Var$ "a" $:/\backslash\ Var$ "b"$)\ :\backslash/\ No\ (Var$ "c"$)$

Observa que el tipo anterior permite representar proposiciones como árboles con nodos binarios (los correspondientes as :/\ y :\/) o unarios (los demás). Por ejemplo, el árbol correspondiente a $e3$ es:



6.1 Copia en un fichero las definiciones de tipos y ejemplos anteriores.

6.2 Para evaluar una proposición en la que aparecen variables es necesario asignar a cada variable un valor en el dominio $\{True, False\}$. Por ejemplo, si asignamos el valor *True* a la variable a, el valor *False* a la variable b y el valor *True* a la variable c (lo cual denotaremos con $[a \mapsto True,\ b \mapsto False,\ c \mapsto True]$), la evaluación de la proposición $(a \wedge b) \vee (no\ c)$ sería *False*, ya que éste es el valor que so obtiene al sustituir en la proposición cada variable por su valor asignado.

En Haskell podemos representar la asignación de un valor a una variable con el tipo:

data *Asig* = *Nombre* :–> *Bool* **deriving** *Show*

y una asignación a varias variables como una lista de éstas:

type *Asignación* = [*Asig*]

Así, la asignación de variables anterior se escribiría ["a" :–> *True*, "b" :–> *False*, "c" :–> *True*] en Haskell.

Define una función *evaluar* :: *Prop* → *Asignación* → *Bool*, de modo que *evaluar p s* devuelva la evaluación de la proposición p con la asignación de variables dada por s. Por ejemplo:

? *evaluar e3* ["a" :–> *True*, "b" :–> *False* "c" :–> *True*]
False :: *Bool*

? *evaluar e3* ["a" :–> *True*, "b" :–> *True* "c" :–> *True*]
True :: *Bool*

Sugerencia: puede resultar útil definir primero una función *valorDe* :: *Nombre* → *Asignación* → *Bool*, que devuelva en valor de una variable dado por una asignación:

? *valorDe* "b" ["a" :–> *True*, "b" :–> *False* "c" :–> *True*]
False :: *Bool*

6.3 Define una función *variables* :: *Prop* → [*Nombre*], que devuelva una lista con todos los nombres de variable (sin repetir) que aparecen en una proposición. Por ejemplo:

? *variables* $e3$
["a", "b", "c"] :: [*Nombre*]

? *variables* $e2$
["a"] :: [*Nombre*]

6.4 Para una proposición con n variables distintas existen 2^n asignaciones de valores posibles, ya que cada variable puede tomar dos valores (*True* o *False*). Por ejemplo, para la proposición *Var* "a" :\/ *Var* "b" existen 4 asignaciones posibles: [a ↦ *True*, b ↦ *True*], [a ↦ *True*, b ↦ *False*], [a ↦ *False*, b ↦ *True*] y [a ↦ *False*, b ↦ *False*]. Define una función *posiblesAsignaciones* :: [*Nombre*] → [*Asignación*], que devuelva todas las asignaciones posibles para una lista de variables. Por ejemplo:

? *posiblesAsignaciones* ["a"]
[["a" :–> *True*], ["a" :–> *False*]]

? *posiblesAsignaciones* ["a", "b"]
[["a" :–> *True*, "b" :–> *True*], ["a" :–> *True*, "b" :–> *False*],
["a" :–> *False*, "b" :–> *True*], ["a" :–> *False*, "b" :–> *False*]]

Nota: observa que todas las asignaciones para n variables pueden ser obtenidas a partir de las de $n-1$ variables, si añadimos a cada una de éstas una asignación v :–> *True* y otra v :–> *False*, donde v representa la nueva variable.

6.5 Se dice que una proposición es satisfacible si es posible encontrar una asignación de variables para la cual el valor de la expresión sea verdadero. Así, la proposición $e1$ es satisfacible ya que, por ejemplo, con la asignación de variables ([a ↦ *True*, b ↦ *True*, c ↦ *True*]) el valor de la proposición es *True*. Define una función *esSatisfacible* :: *Prop* → *Bool* que devuelva *True* si la proposición que toma como parámetro es satisfacible. Por ejemplo:

? *esSatisfacible* $e1$
True :: *Bool*

? *esSatisfacible* $e2$
False :: *Bool*

Sugerencia: tendrás que evaluar la proposición bajo cualquier asignación posible de variables y ver si al menos una evaluación es verdadera. Intenta usar las funciones predefinidas *map* y *or*.

6.6 Se dice que una proposición es una tautología si, bajo cualquier asignación de variables posible, su valor es siempre verdadero. Un ejemplo de tautología es la expresión *Var* "a" :\/ *No* (*Var* "a"). Define una función *esTautología* :: *Prop* → *Bool* que devuelva *True* si la proposición que toma como parámetro es una tautología. Por ejemplo:

```
? esTautología e3
False :: Bool

? esTautología (Var "a" :\/ No (Var "a"))
True :: Bool
```

Sugerencia: tendrás que evaluar la proposición bajo cualquier asignación posible de variables y ver si para todas es verdadera. Intenta usar las funciones predefinidas *map* y *and*.

6.7 La implicación lógica (escrita como $p_1 \to p_2$) para denotar que la proposición p_1 implica la proposición p_2, se interpreta como una proposición que será cierta si siempre que se cumple p_1 se cumple p_2. Una defnición para esta proposición acorde con la interpretación anterior es $p_1 \to p_2 \equiv No\ p_1 \vee p_2$, por lo que podemos definir en Haskell el siguiente operador de implicación:

```
infixl --> 5
(-->)     :: Prop → Prop → Prop
a --> b   =  No a :\/ b
```

Se escribe $p_1 \Rightarrow p_2$ (con flecha doble) para denotar que la implicación $p_1 \to p_2$ es una tautología, es decir, es válida bajo cualquier asignación de variables. Completa la siguiente definición del operador (==>) que permita comprobar implicaciones:

```
infixl ==> 5
(==>)     :: Prop → Prop → Bool
a ==> b   =  . . .
```

Por ejemplo:

```
? Var "a" ==> Var "a" :\/ Var "b"
True :: Bool

? Var "c" ==> Var "a" :\/ Var "b"
False :: Bool
```

6.8 Del mismo modo, completa la definición de los siguientes operadores para comprobar que dos proposiciones son equivalentes:

```
infixl <--> 4
(<-->)        :: Prop → Prop → Prop
a <--> b      = (a --> b) :/\ (b --> a)
infixl <==> 4
(<==>)     :: Prop → Prop → Bool
a <==> b  =  . . .
```

6.9 Escribe expresiones en Haskell para comprobar las siguientes equivalencias lógicas:

- Doble negación: $no\ (no\ a) \quad \Leftrightarrow \quad a$
- Ley de De Morgan: $no\ (a\ \wedge\ b) \quad \Leftrightarrow \quad no\ a\ \vee\ no\ b$
- Ley de De Morgan: $no\ (a\ \vee\ b) \quad \Leftrightarrow \quad no\ a\ \wedge\ no\ b$
- Demostración absurdo: $a\ \rightarrow\ b \quad \Leftrightarrow \quad no\ b\ \rightarrow\ no\ a$
- Deducción: $a\ \wedge\ (a\ \rightarrow\ b) \quad \Rightarrow \quad b$