Práctica 1: Productos y sumas

Ejercicio 1. (Funciones). Definir en Agda habitantes de los siguientes tipos:

a) `flip : {A B C : Set} -> (A -> B -> C) -> B -> A -> C`

b) `compose : {A B C : Set} -> (B -> C) -> (A -> B) -> A -> C`

¿Qué representan desde el punto de vista lógico? ¿Qué representan desde el punto de vista computacional?

Ejercicio 2. (Booleanos). Considerar el tipo de datos inductivo de los booleanos:

```
data Bool : Set where
  false : Bool
  true  : Bool
```

a) Usando *pattern matching*, definir su principio de eliminación, que se comporta esencialmente como un `if_then_else_`:
`recBool : {C : Set} -> C -> C -> Bool -> C`

b) Usando `recBool` (y sin usar *pattern matching*), definir la función `not : Bool -> Bool` que niega su argumento. Evaluar `not true` y `not false`.

Nota: el tipo de datos de los booleanos ya se encuentra definido en la biblioteca estándar de Agda. Se puede importar con:

```
open import Data.Bool using (Bool; true; false)
```

Ejercicio 3. (Productos). Considerar el tipo inductivo de los pares:

```
data _×_ (A B : Set) : Set where
  _,_ : A -> B -> A × B
```

a) Usando *pattern matching*, definir su principio de eliminación:
`recProduct : {C : Set} -> (A -> B -> C) -> A × B -> C`

b) Dar el tipo y definir su principio de eliminación **dependiente**, que generaliza al principio de eliminación anterior para el caso en el que C es una propiedad que depende del par $A \times B$. Desde el punto de vista lógico, el principio de eliminación dependiente afirma que si $A, B : \mathcal{U}$ son tipos y $C : A \times B \to \mathcal{U}$ es una propiedad sobre los pares[1], para probar que vale $C(x)$ para todo $x : A \times B$ alcanza con probar que vale $C((a, b))$ para cada $a : A$ y cada $b : B$.

Nota. Llamamos "principio de eliminación" o "principio de recursión" a la función que abstrae el esquema de recursión primitiva para un tipo de datos inductivo A. Dados un tipo $C : \mathcal{U}$ y un elemento $a : A$, el principio de recursión construye un elemento de un tipo C. Llamamos "principio de eliminación dependiente" o "principio de inducción" a la generalización de dicha función al caso dependiente. Dados ahora una familia de tipos $C : A \to \mathcal{U}$ y un elemento $a : A$, el principio de inducción construye un elemento de tipo $C(a)$.

c) Sin usar pattern matching, y usando los principios de eliminación cuando sea necesario, demostrar:

i. $A \times B \to A$

ii. $A \times B \to B$

iii. $(\Pi_{x:A\times B}\, C(x)) \to \Pi_{a:A}\, \Pi_{b:B}\, C((a, b))$

iv. $(\Pi_{a:A}\, \Pi_{b:B}\, C((a, b))) \to \Pi_{x:A\times B}\, C(x)$

¿Cómo quedan los tipos de los últimos dos ítems en el caso particular en el que C no depende de x? ¿A qué funciones ya conocidas se corresponden?

Nota: el tipo de datos de los pares ya se encuentra definido en la biblioteca estándar de Agda, se puede importar con:

```
open import Data.Product using (_×_; _,_)
```

Ejercicio 4. (Tipo vacío). Considerar el tipo inductivo vacío:

[1] O, más precisamente, una familia de tipos indexada.

```
data ⊥ : Set where
```

A veces, en el lenguaje matemático, notamos "$\bot$" o "$\not\vdash$" al tipo vacío.

a) Usando *pattern matching*, definir el principio de eliminación, que es de tipo:
`⊥-elim : {C : Set} -> ⊥ -> C`.

b) Sin usar pattern matching, y usando el principio de eliminación cuando sea necesario, demostrar

   i. $(A \to \bot) \to A \to B$

Nota: el tipo de datos vacío ya se encuentra definido en la biblioteca estándar de Agda, se puede importar con:

```
open import Data.Empty using (⊥; ⊥-elim)
```

Ejercicio 5. (Tipo unitario). Considerar el tipo inductivo unitario (con un único habitante):

```
data ⊤ : Set where
  tt : ⊤
```

A veces, en el lenguaje matemático, notamos "$\top$" o "$\not\Vdash$" al tipo unitario.

Usando *pattern matching*, definir el principio de eliminación dependiente, que es de tipo:
`indUnit : {C : Unit -> Set} → C tt -> (x : Unit) → C x`.

Nota: el tipo de datos unitario ya se encuentra definido en la biblioteca estándar de Agda, se puede importar con:

```
open import Data.Unit using (⊤; tt)
```

Ejercicio 6. (Sumas dependientes). Considerar el tipo inductivo de las sumas dependientes:

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) → B a → Σ A B
```

1. Dar el tipo y, usando *pattern matching*, definir el principio de eliminación dependiente.
2. Definir las proyecciones $\mathtt{proj}_1$ y $\mathtt{proj}_2$.
3. Demostrar la versión débil del axioma de elección:

   i. $(\Pi_{a:A}\, \Sigma_{b:B}\, C\, a\, b) \to (\Sigma_{f:A\to B}\, \Pi_{a:A}\, C\, a\, (f\, a))$