Introducción a los Algoritmos
Validez, Satisfactibilidad, Tipos y Funciones

Pedro Sánchez Terraf

CIEM-FaMAF — Universidad Nacional de Córdoba

FaMAF
UNC
17 de marzo de 2014

Contenido

Demostraciones

$$
\begin{aligned}
& 5 * (x+3) = 20 \\
\equiv \{ & \ldots \} \\
& 5 * (3+x) = 20
\end{aligned}
$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$
\begin{array}{ll}
 & 5 * (x + 3) = 20 \\
\equiv \{ & \text{Conmutativa} + \} \\
 & 5 * (3 + x) = 20
\end{array}
$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$
\begin{aligned}
& 5 * (x + 3) = 20 \\
\equiv \{ & \text{ Conmutativa + } \} \\
& 5 * (3 + x) = 20
\end{aligned}
$$

Expandimos la justificación:

Conmutativa de +:

$$a + b = b + a$$

Sustituyo a por __
Sustituyo b por __
para obtener

$$x + 3 = 3 + x.$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$5 * (x + 3) = 20$$
$$\equiv \{ \text{ Conmutativa + } \}$$
$$5 * (3 + x) = 20$$

Expandimos la justificación:

Conmutativa de +:

$$a + b = b + a$$

Sustituyo a por __
Sustituyo b por __
para obtener

$$x + 3 = 3 + x.$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$
\begin{array}{rl}
 & 5 * (x + 3) = 20 \\
\equiv & \{ \text{ Conmutativa + } \} \\
 & 5 * (3 + x) = 20
\end{array}
$$

Expandimos la justificación:

Conmutativa de +:

$$a + b = b + a$$

Sustituyo a por $\underline{\ x\ }$
Sustituyo b por $\underline{\ 3\ }$
para obtener

$$x + 3 = 3 + x.$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$
\begin{aligned}
& 5 * (x+3) = 20 \\
\equiv \{ & \text{ Conmutativa + } \} \\
& 5 * (3+x) = 20
\end{aligned}
$$

Expandimos la justificación:

Conmutativa de +:

$$a+b=b+a$$

Sustituyo a por $\underline{\ x\ }$
Sustituyo b por $\underline{\ 3\ }$
para obtener

$$x+3=3+x.$$

1. El “equivalente” ($\equiv$) es el “si y sólo si” ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Demostraciones

$$
\begin{aligned}
& 5 * (x+3) = 20 \\
\equiv \{ & \text{ Conmutativa + } \} \\
& 5 * (3+x) = 20
\end{aligned}
$$

Expandimos la justificación:

Conmutativa de +:

$$a+b=b+a$$

Sustituyo a por $\underline{x}$
Sustituyo b por $\underline{3}$
para obtener

$$x+3=3+x.$$

1. El "equivalente" ($\equiv$) es el "si y sólo si" ($\Leftrightarrow$).
2. Debo justificar usando propiedades **válidas** o **definiciones**.

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$ (hay un ejemplo);
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$ (hay un ejemplo);
3. **no válida** si es $False$ para algún valor de sus variables;
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$ (hay un ejemplo);
3. **no válida** si es $False$ para algún valor de sus variables; (hay un contraejemplo);
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$ (hay un ejemplo);
3. **no válida** si es $False$ para algún valor de sus variables; (hay un contraejemplo);
4. **no satisfactible** si es $False$ para todos los valores de sus variables

Expressiones Booleanas Válidas y Satisfactibles

Una expresión de tipo $Bool$ puede ser:

1. **válida** si es $True$ para todos los valores de sus variables (puedo demostrar que es equivalente a $True$);
2. **satisfactible** si hay al menos un valor de las variables que las hace $True$ (hay un ejemplo);
3. **no válida** si es $False$ para algún valor de sus variables; (hay un contraejemplo);
4. **no satisfactible** si es $False$ para todos los valores de sus variables (puedo demostrar que es equivalente a $False$);

Trabajamos en la Compu

1. Abrir Terminal.
2. Abrir el intérprete de Haskell `ghci`.

Tipos Básicos

Introdujimos los siguientes tipos básicos y otros (derivados) que podemos fabricar con ellos. Una expresión tiene tipo

1. **Num**: si su valor es un número (real).
   Ejemplos:
   - $10 * 5 + 7$,
   - x^2,
   - `length` $[x, y, z]$.
2. **Bool**: si su valor es "verdadero" (*True*) o "falso" (*False*).
   Ejemplos:
   - $10 * 5 + 7 = x^2$,
   - $3 * 7 < 1$,
   - $p \equiv q$,
3. **Char**: si su valor es una caracter (letras, etc.).
   Ejemplos:
   - `'a'`,
   - `'1'`,
   - `' '` (espacio).

Tipos Básicos

Introdujimos los siguientes tipos básicos y otros (derivados) que podemos fabricar con ellos. Una expresión tiene tipo

1. **Num**: si su valor es un número (real).
   Ejemplos:
   - $10 * 5 + 7$,
   - x^2,
   - $\texttt{length}\ [x, y, z]$.
2. **Bool**: si su valor es “verdadero” ($True$) o “falso” ($False$).
   Ejemplos:
   - $10 * 5 + 7 = x^2$,
   - $3 * 7 < 1$,
   - $p \equiv q$,
3. **Char**: si su valor es una caracter (letras, etc.).
   Ejemplos:
   - $\texttt{'a'}$,
   - $\texttt{'1'}$,
   - $\texttt{' '}$ (espacio).

Tipos Básicos

Introdujimos los siguientes tipos básicos y otros (derivados) que podemos fabricar con ellos. Una expresión tiene tipo

1. **Num**: si su valor es un número (real).
   Ejemplos:
   - $10 * 5 + 7$,
   - x^2,
   - $\texttt{length}\ [x, y, z]$.
2. **Bool**: si su valor es "verdadero" ($True$) o "falso" ($False$).
   Ejemplos:
   - $10 * 5 + 7 = x^2$,
   - $3 * 7 < 1$,
   - $p \equiv q$,
3. **Char**: si su valor es una caracter (letras, etc.).
   Ejemplos:
   - `'a'`,
   - `'1'`,
   - `' '` (espacio).

Tipos en Haskell

Podemos averiguar tipos con `ghci` usando `:t`.

La compu no entiende Num

No se pueden representar todos los números reales en la compu, pero podemos usar enteros (tipo Int) números con una cantidad fija de decimales (tipo $Float$) y hay más.

Todos los demás tipos sí están definidos en Haskell.

Podemos averiguar tipos con `ghci` usando `:t`.

La compu no entiende Num

No se pueden representar todos los números reales en la compu, pero podemos usar enteros (tipo Int) números con una cantidad fija de decimales (tipo $Float$) y hay más.

Todos los demás tipos sí están definidos en Haskell.

Tipos en Haskell

Podemos averiguar tipos con `ghci` usando `:t`.

La compu no entiende Num

No se pueden representar todos los números reales en la compu, pero podemos usar enteros (tipo Int) números con una cantidad fija de decimales (tipo $Float$) y hay más.

Todos los demás tipos sí están definidos en Haskell.

Tipos Derivados

Con los tipos básicos podemos hacer listas y tuplas. Las “tuplas” (pares, ternas, etc.) se escriben entre paréntesis y tienen tamaño fijo.

1. **Listas.**
   Ejemplos:
   1. $[x, y+z]$ (de tipo $[Num]$),
   2. $[True, p]$ (de tipo $[Bool]$),
   3. `["hola", "chau"]` (de tipo $[String]$)
2. **Tuplas.**
   Ejemplos:
   1. $(3, -10, x*2)$ (de tipo (Num, Num, Num)),
   2. $(x*5, True)$ (de tipo $(Num, Bool)$),
   3. $(\texttt{"Juan"}, 1.75)$ (de tipo $(String, Num)$),

Comparar los tipos que obtenemos para ambos ejemplos (3) en Haskell.

Se pueden combinar, por ejemplo listas de listas $[[1,2],[5],[8,9,10]]$ (de tipo $[[Num]]$), listas de pares $[(\texttt{"Juan"}, 1.75), (\texttt{"Jose"}, 1.83)]$, pares de listas $([1,2], [\texttt{'b'}, \texttt{'1'}])$ etcétera.

Tipos Derivados

Con los tipos básicos podemos hacer listas y tuplas. Las "tuplas" (pares, ternas, etc.) se escriben entre paréntesis y tienen tamaño fijo.

1. **Listas.**
   Ejemplos:
   1. $[x, y+z]$ (de tipo $[Num]$),
   2. $[True, p]$ (de tipo $[Bool]$),
   3. $[$ "hola", "chau" $]$ (de tipo $[String]$)
2. **Tuplas.**
   Ejemplos:
   1. $(3, -10, x*2)$ (de tipo (Num, Num, Num)),
   2. $(x*5, True)$ (de tipo $(Num, Bool)$).
   3. $($"Juan"$, 1.75)$ (de tipo $(String, Num)$),

Comparar los tipos que obtenemos para ambos ejemplos (3) en Haskell.

Se pueden combinar, por ejemplo listas de listas $[[1,2],[5],[8,9,10]]$ (de tipo $[[Num]]$), listas de pares $[($"Juan"$, 1.75), ($"Jose"$, 1.83)]$, pares de listas $([1,2], [$'b', '1'$])$ etcétera.

Tipos Derivados

Con los tipos básicos podemos hacer listas y tuplas. Las "tuplas" (pares, ternas, etc.) se escriben entre paréntesis y tienen tamaño fijo.

1. **Listas.**
   Ejemplos:
   1. $[x, y+z]$ (de tipo $[Num]$),
   2. $[True, p]$ (de tipo $[Bool]$),
   3. $[$ `"hola"`, `"chau"` $]$ (de tipo $[String]$)
2. **Tuplas.**
   Ejemplos:
   1. $(3, -10, x*2)$ (de tipo (Num, Num, Num)),
   2. $(x*5, True)$ (de tipo $(Num, Bool)$).
   3. $($`"Juan"`$, 1.75)$ (de tipo $(String, Num)$),

Comparar los tipos que obtenemos para ambos ejemplos (3) en Haskell.

Se pueden combinar, por ejemplo listas de listas $[[1,2],[5],[8,9,10]]$ (de tipo $[[Num]]$), listas de pares $[($`"Juan"`$, 1.75), ($`"Jose"`$, 1.83)]$, pares de listas $([1,2], [$`'b'`$,$`'1'`$])$ etcétera.

Tipos Derivados

Con los tipos básicos podemos hacer listas y tuplas. Las "tuplas" (pares, ternas, etc.) se escriben entre paréntesis y tienen tamaño fijo.

1. **Listas.**
   Ejemplos:
   1. $[x, y+z]$ (de tipo $[Num]$),
   2. $[True, p]$ (de tipo $[Bool]$),
   3. [`"hola"`, `"chau"`] (de tipo $[String]$)
2. **Tuplas.**
   Ejemplos:
   1. $(3, -10, x*2)$ (de tipo (Num, Num, Num)),
   2. $(x*5, True)$ (de tipo $(Num, Bool)$).
   3. (`"Juan"`, 1.75) (de tipo $(String, Num)$),

Comparar los tipos que obtenemos para ambos ejemplos (3) en Haskell.

Se pueden combinar, por ejemplo listas de listas $[[1,2],[5],[8,9,10]]$ (de tipo $[[Num]]$), listas de pares [(`"Juan"`, 1.75), (`"Jose"`, 1.83)], pares de listas ($[1,2]$, [`'b'`, `'1'`]) etcétera.

Tipos Derivados

Con los tipos básicos podemos hacer listas y tuplas. Las "tuplas" (pares, ternas, etc.) se escriben entre paréntesis y tienen tamaño fijo.

1. **Listas**.
   Ejemplos:
   1. $[x, y+z]$ (de tipo $[Num]$),
   2. $[True, p]$ (de tipo $[Bool]$),
   3. [`"hola"`, `"chau"`] (de tipo $[String]$)
2. **Tuplas**.
   Ejemplos:
   1. $(3, -10, x*2)$ (de tipo (Num, Num, Num)),
   2. $(x*5, True)$ (de tipo $(Num, Bool)$).
   3. (`"Juan"`, 1.75) (de tipo $(String, Num)$),

Comparar los tipos que obtenemos para ambos ejemplos (3) en Haskell.

Se pueden combinar, por ejemplo listas de listas $[[1,2],[5],[8,9,10]]$ (de tipo $[[Num]]$), listas de pares [(`"Juan"`, 1.75), (`"Jose"`, 1.83)], pares de listas ($[1,2]$, [`'b'`, `'1'`]) etcétera.

Más sobre Listas

Las listas se construyen a partir de $[\,]$ (lista vacía) y de $\triangleright$ (agregar elementos)

En el *Formalismo Básico*

$$[2,3] = 2 \triangleright \underline{[3]} = 2 \triangleright (3 \triangleright [\,])$$

En *Haskell*

```
[2,3] = 2 : [3] = 2 : (3 : [])
```

Definiciones de Funciones (I)

Para **definir** una función en el *Formalismo Básico* usamos el signo $\doteq$.

Definición de S

$$S.x \doteq x + 1$$

En la clase pasada aplicamos esta definición para calcular:

$$\underline{S.(1+1)} = S.1 + S.1$$
$\equiv$ { Definición de S }
$$1+1+1 = \underline{S.1} + \underline{S.1}$$
$\equiv$ { Definición de S x2 }
$$1+1+1 = 1+1+1+1+1$$
$\equiv$ { Aritmética }
$$3 = 4$$
$\equiv$ { Aritmética }
$$False$$

Definiciones de Funciones (I)

Para **definir** una función en el *Formalismo Básico* usamos el signo $\doteq$.

Definición de S

$$S.x \doteq x + 1$$

En la clase pasada aplicamos esta definición para calcular:

$$\begin{aligned}
& \underline{S.(1+1)} = S.1 + S.1 \\
\equiv & \{ \text{ Definición de } S \text{ } \} \\
& 1+1+1 = \underline{S.1} + \underline{S.1} \\
\equiv & \{ \text{ Definición de } S \text{ x2 } \} \\
& 1+1+1 = 1+1+1+1+1 \\
\equiv & \{ \text{ Aritmética } \} \\
& 3 = 4 \\
\equiv & \{ \text{ Aritmética } \} \\
& \mathit{False}
\end{aligned}$$

Definiciones de Funciones (I)

Para **definir** una función en el *Formalismo Básico* usamos el signo $\doteq$.

Definición de S

$$S.x \doteq x + 1$$

En la clase pasada aplicamos esta definición para calcular:

$$
\begin{aligned}
& \underline{S.(1+1)} = S.1 + S.1 \\
\equiv \{ & \text{ Definición de } S \ \} \\
& 1 + 1 + 1 = \underline{S.1} + \underline{S.1} \\
\equiv \{ & \text{ Definición de } S \text{ x2 } \} \\
& 1 + 1 + 1 = 1 + 1 + 1 + 1 + 1 \\
\equiv \{ & \text{ Aritmética } \} \\
& 3 = 4 \\
\equiv \{ & \text{ Aritmética } \} \\
& \mathit{False}
\end{aligned}
$$

Definiciones de Funciones (I)

Para **definir** una función en el *Formalismo Básico* usamos el signo $\doteq$.

Definición de S

$$S.x \doteq x + 1$$

En la clase pasada aplicamos esta definición para calcular:

$$\underline{S.(1+1)} = S.1 + S.1$$

$\equiv$ { Definición de S }

$$1+1+1 = \underline{S.1} + \underline{S.1}$$

$\equiv$ { Definición de S x2 }

$$1+1+1 = 1+1+1+1+1$$

$\equiv$ { Aritmética }

$$3 = 4$$

$\equiv$ { Aritmética }

$$\mathit{False}$$

Definiciones de Funciones (I)

Para **definir** una función en el *Formalismo Básico* usamos el signo $\doteq$.

Definición de S

$$S.x \doteq x + 1$$

En la clase pasada aplicamos esta definición para calcular:

$$
\begin{array}{ll}
 & \underline{S.(1+1)} = S.1 + S.1 \\
\equiv & \{ \text{ Definición de } S \ \} \\
 & 1+1+1 = \underline{S.1} + \underline{S.1} \\
\equiv & \{ \text{ Definición de } S \text{ x2 } \} \\
 & 1+1+1 = 1+1+1+1+1 \\
\equiv & \{ \text{ Aritmética } \} \\
 & 3 = 4 \\
\equiv & \{ \text{ Aritmética } \} \\
 & \mathit{False}
\end{array}
$$

Definiciones de Funciones (II)

Un ejemplo de función que tiene una tupla como argumento.

En el *Formalismo Básico*

$$g.(x,y) \doteq (x^2 + y^2, x * y)$$

En *Haskell*

```
g (x,y) = (x**2+y**2, x*y)
```

Ejercicio

Aplicarla a $(3,4)$ y a $(x * y, x)$.

Definiciones de Funciones (II)

Un ejemplo de función que tiene una tupla como argumento.

En el *Formalismo Básico*

$$g.(x,y) \doteq (x^2 + y^2, x * y)$$

En *Haskell*

```
g (x,y) = (x**2+y**2, x*y)
```

Ejercicio

Aplicarla a $(3,4)$ y a $(x*y,x)$.

Definiciones de Funciones (III)

Un ejemplo de función que tiene una lista como argumento.

En el *Formalismo Básico*

$$head.(x \triangleright xs) \doteq x$$

En *Haskell*

```
head (x : xs) = x
```

Ejercicio

Aplicarla a $[1,2,3]$, a ["hola", "chau"] y a $[[],[1,2]]$.

Definiciones de Funciones (III)

Un ejemplo de función que tiene una lista como argumento.

En el *Formalismo Básico*

$$head.(x \triangleright xs) \doteq x$$

En *Haskell*

```
head (x : xs) = x
```

Ejercicio

Aplicarla a $[1,2,3]$, a [`"hola"`, `"chau"`] y a $[[],[1,2]]$.

Definir Funciones en `ghci`

1. Abrir Terminal.
2. Abrir Editor de texto (gedit, kate).
3. Escribir las funciones `f` y `multiplicar` del ejercicio 15 del Práctico 1 y la función `g` de antes.
4. Guardar el archivo con el nombre `apellido_nombre.hs` (ejemplo: `sanchezterraf_pedro.hs`).
5. Enviarlo a mi email (pedrost arroba gmail punto com).

Una vez creado el archivo, se puede abrir con `ghci`.

1. En `ghci`, cargamos nuestro archivo con :l (dos puntos ele).
   `:l apellido_nombre.hs`
2. Podemos ahora probar ejemplos.

Definir Funciones en `ghci`

1. Abrir Terminal.
2. Abrir Editor de texto (gedit, kate).
3. Escribir las funciones `f` y `multiplicar` del ejercicio 15 del Práctico 1 y la función `g` de antes.
4. Guardar el archivo con el nombre `apellido_nombre.hs` (ejemplo: `sanchezterraf_pedro.hs`).
5. Enviarlo a mi email (pedrost arroba gmail punto com).

Una vez creado el archivo, se puede abrir con `ghci`.

1. En `ghci`, cargamos nuestro archivo con :l (dos puntos ele).
   `:l apellido_nombre.hs`
2. Podemos ahora probar ejemplos.

Ejercicio

Plantear un problema "práctico" que se pueda resolver con una función.

Ejemplo: Tengo la lista de contactos de mi celular anotados con el nombre primero y luego el apellido. Lo necesito al revés.
Llamo *alReves* a la función que quiero; va a "comer" una lista de ternas y devuelve otra lista de ternas.

Así funcionaría:
alReves.[("Juan","Pérez",4223729), ("Pepito","Sánchez",4555555)] =
= [("Pérez","Juan",4223729), ("Sánchez","Pepito",4555555)].

Ejercicio

Plantear un problema "práctico" que se pueda resolver con una función.

Ejemplo: Tengo la lista de contactos de mi celular anotados con el nombre primero y luego el apellido. Lo necesito al revés.
Llamo $alReves$ a la función que quiero; va a "comer" una lista de ternas y devuelve otra lista de ternas.

Así funcionaría:

$alReves.[(\texttt{"Juan"},\texttt{"Pérez"},4223729), (\texttt{"Pepito"},\texttt{"Sánchez"},4555555)] =$
$= [(\texttt{"Pérez"},\texttt{"Juan"},4223729), (\texttt{"Sánchez"},\texttt{"Pepito"},4555555)].$

Ejercicio

Plantear un problema “práctico” que se pueda resolver con una función.

Ejemplo: Tengo la lista de contactos de mi celular anotados con el nombre primero y luego el apellido. Lo necesito al revés.
Llamo $alReves$ a la función que quiero; va a “comer” una lista de ternas y devuelve otra lista de ternas.

Así funcionaría:
$alReves.\big[($`"Juan","Pérez"`$,4223729), ($`"Pepito","Sánchez"`$,4555555)\big] =$
$= \big[($`"Pérez","Juan"`$,4223729), ($`"Sánchez","Pepito"`$,4555555)\big].$

Un problema más simple

Ejercicio

Definir la función $intercambia : (String, String) \to (String, String)$ que intercambia los lugares de un par.

Ejemplos:

- $intercambia.$("Juan","Pérez") = ("Pérez","Juan");
- $intercambia.$("Pepito","Sánchez")= ("Sánchez","Pepito").

Un problema más simple

Ejercicio

Definir la función $intercambia : (String, String) \to (String, String)$ que intercambia los lugares de un par.

Ejemplos:

- $intercambia$.(`"Juan","Pérez"`) = (`"Pérez","Juan"`);
- $intercambia$.(`"Pepito","Sánchez"`)= (`"Sánchez","Pepito"`).

Tareas para hoy

En este orden.

1. Crear archivo de Haskell con las 3 funciones y mandármelo por mail.
2. Entregar por escrito los ejemplos de las funciones g y $head$ (justificando con $\equiv \{\cdots\}$).
3. Entregar por escrito el problema "de la vida real" que se resuelve con una función.
4. Definir la función $intercambia$.