Tipado

Taller de álgebra I

Primer cuatrimestre de 2016

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipo de datos

Conjunto de valores (llamado el conjunto base del tipo) junto con una serie de funciones que involucran al conjunto base.

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipo de datos

Conjunto de valores (llamado el conjunto base del tipo) junto con una serie de funciones que involucran al conjunto base.

Ejemplos

1. `Integer` $= (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipo de datos

Conjunto de valores (llamado el conjunto base del tipo) junto con una serie de funciones que involucran al conjunto base.

Ejemplos

1. `Integer` $= (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. `Float` $= (\mathbb{Q}*, \{+, -, *, /\})$ es el tipo de datos que "representa" a los racionales, con la aritmética de punto flotante.

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipo de datos

Conjunto de valores (llamado el conjunto base del tipo) junto con una serie de funciones que involucran al conjunto base.

Ejemplos

1. $\texttt{Integer} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}*, \{+, -, *, /\})$ es el tipo de datos que "representa" a los racionales, con la aritmética de punto flotante.
3. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\&\&, ||, \texttt{not}\})$ representa a los valores lógicos.

Tipos de datos

De la clase pasada tenemos $\mathbb{Z}$ y `True/False` como valores posibles a usar. Y conocemos algunas operaciones. También vimos que se podía hacer `sqrt 2`, cuyo valor no es entero ni booleano.

Tipo de datos

Conjunto de valores (llamado el conjunto base del tipo) junto con una serie de funciones que involucran al conjunto base.

Ejemplos

1. $\texttt{Integer} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}*, \{+, -, *, /\})$ es el tipo de datos que "representa" a los racionales, con la aritmética de punto flotante.
3. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\&\&, ||, \texttt{not}\})$ representa a los valores lógicos.

Dado un valor de un tipo de datos, solamente se pueden aplicar a ese valor las operaciones definidas para ese tipo de datos.

Tipos de datos

En Haskell los tipos se notan con `::`. Por ejemplo, en ghci podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

Tipos de datos

En Haskell los tipos se notan con `::`. Por ejemplo, en ghci podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

A las expresiones también les corresponde un tipo de dato.

```
Prelude> :t (4^10000000 + 2) < 1
(4^10000000 + 2) < 1 :: Bool
```

Tipos de datos

En Haskell los tipos se notan con `::`. Por ejemplo, en ghci podemos ver el tipo del siguiente valor:

```
Prelude> :t True
True :: Bool
```

A las expresiones también les corresponde un tipo de dato.

```
Prelude> :t (4^10000000 + 2) < 1
(4^10000000 + 2) < 1 :: Bool
```

Dada una expresión, se puede determinar su tipo sin saber su valor.

Tipos de datos

¿Qué tipo tiene la expresión?

```
f True
```

Tipos de datos

¿Qué tipo tiene la expresión?

```
f True
```

Depende de `f`.

¿Qué tipo tiene la expresión?

```
f True
```

Depende de `f`. Si usamos

```
f :: Bool -> Bool
f x = not x
```

¿Qué tipo tiene la expresión?

```
f True
```

Depende de `f`. Si usamos

```
f :: Bool -> Bool
f x = not x
```

```
f :: Bool -> Float
f x = pi
```

¿Qué tipo tiene la expresión?

```
f True
```

Depende de `f`. Si usamos

```
f :: Bool -> Bool
f x = not x
```

```
f :: Bool -> Float
f x = pi
```

```
funcion3 :: Integer -> Integer -> Bool -> Bool
funcion3 x y b = b || (x > y)
```

Aplicación de funciones

```
Prelude>:t funcion3 10 20 True
funcion3 10 20 True :: Bool
```

Ejercicios

Determinar el tipo de las siguientes funciones

```
doble :: ??
doble x = x + x

cuadruple :: ??
cuadruple x = doble (doble x)

-- distancia entre p1 y p2 puntos
-- donde p1 = (x1, y1) y p2 = (x2, y2)
dist :: ??
dist x1 y1 x2 y2 = sqrt ((x2 - x1)^2 + (y2 - y1)^2)
```

Ejercicios

Determinar el tipo de las siguientes funciones

```
doble :: ??
doble x = x + x

cuadruple :: ??
cuadruple x = doble (doble x)

-- distancia entre p1 y p2 puntos
-- donde p1 = (x1, y1) y p2 = (x2, y2)
dist :: ??
dist x1 y1 x2 y2 = sqrt ((x2 - x1)^2 + (y2 - y1)^2)
```

Ejercicios

Determinar el tipo de las siguientes expresiones

- `doble 10`
- `dist (dist pi 0 pi 1) (doble 0) (doble 2) (3/4)`
- `doble True`

Tipos de datos

- Es importante observar la signatura de las funciones en las definiciones anteriores.
- Especificamos explícitamente el tipo de datos del dominio y el codominio de las funciones que definimos.

Tipos de datos

- Es importante observar la signatura de las funciones en las definiciones anteriores.
- Especificamos explícitamente el tipo de datos del dominio y el codominio de las funciones que definimos.

    1. No es estrictamente necesario especificarlo, dado que el mecanismo de inferencia de tipos de Haskell puede deducir la signatura más general para cada función.
    2. Sin embargo, es buena idea dar explícitamente la signatura de las funciones (¿por qué?).

Tipos de datos: Tuplas

- Dados dos tipos de datos `A` y `B`, tenemos el tipo de datos `(A,B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`.

Tipos de datos: Tuplas

- Dados dos tipos de datos `A` y `B`, tenemos el tipo de datos `(A,B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`.

- Algunas operaciones: `fst` y `snd`
  `fst (1 + 4,2)` $\rightsquigarrow$ `5`
  `snd (1,(2,3))` $\rightsquigarrow$ `(2,3)`

Tipos de datos: Tuplas

- Dados dos tipos de datos `A` y `B`, tenemos el tipo de datos `(A,B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`.
- Algunas operaciones: `fst` y `snd`
  `fst (1 + 4,2)` $\rightsquigarrow$ `5`
  `snd (1,(2,3))` $\rightsquigarrow$ `(2,3)`
- Ahora podemos definir la distancia un poco más claramente:

```
dist :: (Float,Float) -> (Float,Float) -> Float
dist p1 p2 = sqrt ((fst p1 - fst p2)^2 + (snd p1 - snd p2)^2)
```

Nota:

- Hay tuplas de distintos tamaños: `(True,1,4.0)`, `(0,0,0,0)`.

Tipos de datos: Variables de Tipos

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Uno de los mecanismos posibles se ilustra en el siguiente ejemplo:

```
fConstante10 :: a -> Integer
fConstante10 x = 10

parear :: a -> (a,a)
parear x = (x,x)
```

Notar que a va en minúscula y denota una variable de tipo.

Tipos de datos: Variables de Tipos

A veces las funciones que queremos escribir pueden funcionar sobre muchos tipos de datos. Uno de los mecanismos posibles se ilustra en el siguiente ejemplo:

```
fConstante10 :: a -> Integer
fConstante10 x = 10

parear :: a -> (a,a)
parear x = (x,x)
```

Notar que a va en minúscula y denota una variable de tipo.

A veces necesitamos restringir el dominio:

```
doble :: Num a => a -> a
doble x = x + x
```

Permite ejecutar:

```
doble 10
doble pi
```

Tipos de datos

Ejercicios

- Implementar las siguientes funciones
  - `crearPar :: a -> b -> (a,b)`
  - `invertir :: (a,b) -> (b,a)`
  - `distancia :: (Float,Float) -> (Float,Float) -> Float`
- Completar la implementación de la función `raices :: Float -> Float -> Float -> (Float, Float)` para que esta vez devuelva las dos raíces de la función cuadrática.

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

```
ite True exp1 exp2 = exp1
ite False exp1 exp2 = exp2
```

- ¿Cuánto vale? `ite (4 < 3) 2 1` $\rightsquigarrow$

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

```
ite True  exp1 exp2 = exp1
ite False exp1 exp2 = exp2
```

- ¿Cuánto vale? `ite (4 < 3)  2 1` $\rightsquigarrow$ `1`

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

```
ite True  exp1 exp2 = exp1
ite False exp1 exp2 = exp2
```

- ¿Cuánto vale? `ite (4 < 3)  2 1` $\rightsquigarrow$ `1`
- ¿A qué equivale la función `f x y = ite x x y`?

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

```
ite True  exp1 exp2 = exp1
ite False exp1 exp2 = exp2
```

- ¿Cuánto vale? `ite (4 < 3)  2 1` $\rightsquigarrow$ `1`
- ¿A qué equivale la función `f x y = ite x x y`? al operador `||`

ite (de acá!)

Hagamos una función `ite` que tome como parámetro una condición booleana y dos expresiones de algún tipo (el mismo). El resultado será la primera expresión si la condición es verdadera, y la segunda expresión si la condición es falsa.

```
ite :: Bool -> a -> a -> a
```

```
ite True  exp1 exp2 = exp1
ite False exp1 exp2 = exp2
```

- ¿Cuánto vale? `ite (4 < 3)  2 1` $\rightsquigarrow$ `1`
- ¿A qué equivale la función `f x y = ite x x y`? al operador `||`

En haskell ya existe...

```
if <condicion> then <exp1> else <exp2>
```

```
(||) :: Bool -> Bool -> Bool
(||) p q = if p then p else q
```

Un nuevo tipo: Listas

Tipo Lista

Las listas pueden contener elementos de cualquier tipo (incluso listas)

- `[1] :: [Integer]`
- `[1, 2] :: [Integer]`
- `[1.1, 2, 4, 3.2] :: [Float]`
- `[[1], [2,3], [], [1,1000,2,0]] :: [[Integer]]`
- [1, True] $\rightsquigarrow\rightsquigarrow\rightsquigarrow$ NO ES UNA LISTA VÁLIDA, ¿por qué?

Un nuevo tipo: Listas

Tipo Lista

Las listas pueden contener elementos de cualquier tipo (incluso listas)

- `[1] :: [Integer]`
- `[1, 2] :: [Integer]`
- `[1.1, 2, 4, 3.2] :: [Float]`
- `[[1], [2,3], [], [1,1000,2,0]] :: [[Integer]]`
- `[1, True]` $\rightsquigarrow\rightsquigarrow\rightsquigarrow$ NO ES UNA LISTA VÁLIDA, ¿por qué?

Ejercicio

Tipar las siguientes expresiones

- `[(1,2), (3,4), (5,2)]`
- `[maximo 2 3, fst (2+2, 3+4), 3+4 - 3/4]`
- `[ [], [], [], [], [] ]`
- `[]`
- `[if True then False else True, 3 > 3 && 4 < 2]`

Un nuevo tipo: Listas

Otras formas de definir listas

Prueben las siguientes expresiones en GHCI

- `[1..100]`
- `[1,3..100]`
- `[100..1]`

Un nuevo tipo: Listas

Otras formas de definir listas

Prueben las siguientes expresiones en GHCI

- `[1..100]`
- `[1,3..100]`
- `[100..1]`

Ejercicios

- Escribir una expresión que denote la lista estrictamente decreciente que comienza con el número 1 y termina con el número -100.
- Definir la función `listar :: a -> a -> a -> [a]` que toma 3 elementos y los convierte en una lista.
- Definir la función `rangoDePaso :: Integer -> Integer -> Integer -> [Integer]` que, dados 3 números n_1, n_2 y n_3, devuelve la lista resultante de crear el intervalo desde n_1 hasta n_2 y de paso n_3. Suponer que $n_1 \leq n_2$

Operaciones

Algunas operaciones

- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `(:) :: a -> [a] -> [a]`
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Int`
- `reverse :: [a] -> [a]`

Operaciones

Algunas operaciones

- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `(:) :: a -> [a] -> [a]`
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Int`
- `reverse :: [a] -> [a]`

Tipar y evaluar las siguientes expresiones

- `head [(1,2), (3,4), (5,2)]`
- `tail [1,2,3,4,4,3,2,1]`
- `head []`
- `head [1,2,3] : [2,3]`
- `[True, True] ++ [False, False]`
- `[1,2] : []`

Ejercicios

1. Implementar la función

```
pendiente:: (Float,Float) -> (Float,Float) -> Float
```

que toma dos puntos y calcula la pendiente de la recta que pasa por esos puntos.

2. Buscar qué significa el tipo `Char` y qué valores posee. Luego para las siguientes funciones, completar su tipo y explicar su comportamiento (para poder ejecutar estas funciones deberán importar el modulo Data.Char agregando como primer linea al archivo .hs: `import Data.Char`)

```
toUpper :: ?
isSpace :: ?
isDigit :: ?
ord :: ?
chr :: ?
```

3. Buscar qué significa el tipo `String` y dar algún ejemplo de valores del tipo.

4. Determinar qué devuelve la siguiente expresión:

```
"Ginobili" == [ G , i , n , o , b , i , l , i ]
```

5. Implementar la función `iniciales :: String -> String -> String` que dado el nombre y apellido de una persona, devuelve sus iniciales con puntos. Por ejemplo: `iniciales "Harry" "Potter"` $\rightsquigarrow$ `"H.P."`