Auxiliar 1

Introducción, Functions & Type Classes

Profesor: Federico Olmedo
Auxiliar: Damián Árquez

Repaso

Programación funcional vs Programación imperativa:

- **Programación imperativa**: Programas se escriben (y leen) en base a **cómo** hacer algo.
- **Programación funcional**: Programas se escriben (y leen) en torno a **qué** queremos hacer.

Características de Haskell:

- **Statically Typed**: Cada parte del código tiene un tipo asignado en tiempo de compilación. Si no se puede asignar un tipo entonces se tiene un error de compilación.
- **Type inference**: Haskell infiere los tipos de las expresiones automáticamente.
- **Purely functional programming**: No tiene *side-effects*.
- **Lazy**: Ninguna expresión es computada hasta que se necesite su valor.

Tipos básicos: Boolean, Int, Integer (Int más grande pero menos eficiente), Float, Double (Real con doble de precisión), Char, String

Estructuras básicas:

- **Listas**: Los elementos de una lista solo pueden tener un tipo.
  - [ ] :: [a]
  - [True, False, True] :: [Bool]
  - [1, 2, 4, 5] :: Num a => [a]
- **Tuplas**: Largo finito. Pueden tener elementos de distintos tipos.
  - () :: ()

- (True, 2) :: Num b => (Bool, b)

Type classes: Colección de **tipos** que comparten ciertas operaciones. Ejemplos: `Num`, `Integral`, `Fractional`, `Eq`, `Ord`, `Show`.

Para ver el tipo de una expresión, en `ghci`, pueden utilizar el comando `:t <expr>` o `:type <expr>`.

Para cargar un archivo `.hs` en `ghci` pueden usar el comando `:load <path-to-file>`. Por ejemplo: `:load aux1p2.hs`

Instrucciones

Cada **P*** tiene a su lado el nombre de un archivo. La idea es que vayan resolviendo los ejercicios en archivos con dichos nombres, de manera que puedan ir cargando y probando sus archivos en `ghci`. Además, la pauta tendrá el mismo nombre de archivos.

Types

P1

Escriba el tipo de las siguientes expresiones. Luego verifique utilizando `ghci`:

- `[ 'a', 'b', 'c' ]`
- `( 'a', 'b', 'c' )`
- `[(False,'0'),(True,'1')]`
- `([ False, True ], [ '0', '1' ])`
- `[ tail, init, reverse ]`

P2 (`aux1p2.hs`)

Escriba un programa con una definición para cada expresión. Luego, cargue su programa en `ghci` y pruebe que tengan el tipo que usted esperaba.

- `bools :: [Bool]`
- `nums :: [[Int]]`
- `add :: Int -> Int -> Int -> Int`
- `copy :: a -> (a, a)`
- `apply :: (a -> b) -> a -> b`

Funciones

P3 (aux1p3.hs)

Usando funciones conocidas, escriba la función `halve :: [a] -> ([a], [a])` que toma una lista de cantidad de elementos pares y retorna una tupla con las dos mitades de la lista. Debe escribir esta función en el archivo `aux1p3.hs`. Finalmente, cargue su archivo en `ghci` y pruebe que funciona correctamente. *Hint*: Puede usar la función `splitAt :: Int -> [a] -> ([a], [a])` que corta la lista en el indice indicado.

Código 1: halve

```
> halve [ 1, 2, 3, 4, 5, 6 ]
  ([ 1, 2, 3 ], [ 4, 5, 6 ])

```

P4 (aux1p4.hs)

Escriba la función `third :: [a] -> a` que retorna el tercer elemento de una lista que contiene al menos 3 elementos. Escriba dos versiones de esta función: utilizando `head` y `tail`. Puede llamarla `thirdHT`; utilizando *pattern matching*. Puede llamarla `thirdPM`. Escriba sus funciones en el archivo `aux1p4.hs` y luego carguelo en `ghci` para testear.

P5 (aux1p5.hs)

Considere la función `safetail` que se comporta igual que la función `tail` pero que en el caso de la lista vacía no arroja un error si no que devuelve dicha lista. Utilizando `tail` y `null :: [a] -> Bool`, que retorna *True* si una lista esta vacía, defina `safetail` usando:

- Conditional expressions. (`safetailCE`)
- Guarded equations. (`safetailGE`)
- Pattern matching. (`safetailPM`)

P6 (aux1p6.hs)

El algoritmo de Luhn es utilizado para validar números de tarjetas de crédito, y funciona de la siguiente manera:

- Se toma cada dígito de la tarjeta como un número individual.
- Se ignora el primer número.
- Se dobla cada número.
- Se le resta 9 a todos los números mayores que 9.

- Se suman todos los resultados.
- Si el total es divisible por 10 entoncesa la tarjeta es válida.

Defina una función `luhnDouble :: Int -> Int` que dobla un dígito y le resta 9 si es que el resultado es mayor que 9. Luego defina `luhn :: Int -> Int -> Int -> Int -> Bool` que verifica si una tarjeta de 4 dígitos es válida. Por ejemplo:

Código 2: halve

```
> luhnDouble 3
  6
> luhnDouble 6
  3
> luhn 1 4 7 8
  True
> luhn 4 7 8 3
  False

```