MATEMÁTICA DISCRETA
Teoría de Números

Prof. Sergio Salinas

Facultad de Ingeniería
Universidad Nacional de Cuyo

Agosto 2024

Contenido

Introducción a Haskell

Introducción

- **Lenguajes de programación imperativos:** es un paradigma de programación que se centra en describir *cómo* un programa debe realizar tareas mediante una secuencia de instrucciones en un orden específico que modifican el estado del programa a lo largo del tiempo.

- **Asignación de Variables**: utilizan variables para almacenar datos que pueden ser modificados a lo largo del programa mediante operaciones de asignación.

- **Control de Flujo**: emplean estructuras de control como bucles (`for`, `while`), condicionales (`if`, `else`), y ramas (`switch`, `case`) para controlar el flujo de ejecución del programa.

- **Estado del Programa**: la ejecución de un programa imperativo implica cambios en el estado del sistema, que está representado por el valor de las variables y la memoria del programa.

- **Subrutinas y Funciones**: permiten la definición de bloques de código reutilizables, como funciones y procedimientos, que pueden ser llamados en diferentes partes del programa.

Introducción

- **C**: un lenguaje de bajo nivel conocido por su eficiencia y control sobre el hardware.
- **Java**: un lenguaje de programación de alto nivel, orientado a objetos, que también sigue el paradigma imperativo.
- **Python**: aunque es multi-paradigma, incluye muchas características imperativas.
- **Pascal**: un lenguaje educativo diseñado para enseñar principios de programación estructurada.

Introducción

Ventajas:

- **Claridad**: La secuencia de pasos es clara y fácil de seguir.
- **Eficiencia**: Los lenguajes imperativos suelen ser más eficientes en términos de ejecución.
- **Control**: Proporcionan un gran control sobre el hardware y la memoria.

Desventajas:

- **Complejidad**: A medida que los programas crecen, pueden volverse complejos y difíciles de mantener.
- **Errores**: Los cambios en el estado del programa pueden introducir errores difíciles de detectar.
- **Reutilización**: Menos orientados a la reutilización de código comparado con otros paradigmas como el funcional o el orientado a objetos.

Comparación con Otros Paradigmas

- **Declarativo**: En lugar de especificar *cómo* realizar las tareas, los lenguajes declarativos describen *qué* hacer. Ejemplos incluyen SQL y lenguajes funcionales como Haskell.

- **Orientado a Objetos**: Mientras que los lenguajes orientados a objetos también pueden ser imperativos, se centran más en la organización del código alrededor de objetos y clases.

Lenguajes de programación declarativos

Los lenguajes de programación declarativos son un paradigma de programación que se centra en describir *qué* se debe lograr, en lugar de *cómo* lograrlo. Los programadores especifican los resultados deseados sin detallar los pasos necesarios para alcanzarlos.

Características Principales

- **Descriptivos**: Los programas declarativos describen el problema y el resultado esperado sin especificar los pasos para obtenerlo.
- **Alta Abstracción**: Se centran en el *qué* y no en el *cómo*, ofreciendo un mayor nivel de abstracción.
- **Menos Control de Flujo**: A menudo, no utilizan estructuras de control tradicionales como bucles y condicionales de manera explícita.
- **Inmutabilidad**: Tienden a promover la inmutabilidad, evitando cambios en el estado del programa.
- **Declaraciones en Lugar de Instrucciones**: Utilizan declaraciones para definir las relaciones y propiedades de los datos.

Ejemplos de Lenguajes Declarativos

- **SQL**: Lenguaje utilizado para gestionar y manipular bases de datos relacionales.
- **HTML**: Lenguaje de marcado utilizado para definir la estructura y el contenido de las páginas web.
- **Prolog**: Lenguaje de programación lógica utilizado en inteligencia artificial y computación simbólica.
- **Haskell**: Lenguaje funcional puro que enfatiza las expresiones y funciones inmutables.

Ventajas y Desventajas

Ventajas:

- **Simplicidad**: Los programas son a menudo más simples y fáciles de entender.
- **Mantenimiento**: Facilitan el mantenimiento y la modificación del código debido a su claridad y concisión.
- **Paralelismo**: Suelen ser más adecuados para la ejecución paralela y concurrente.

Desventajas:

- **Rendimiento**: Pueden ser menos eficientes en términos de rendimiento en comparación con los lenguajes imperativos.
- **Flexibilidad**: Menos flexibles para ciertas tareas que requieren un control detallado del flujo de ejecución.
- **Curva de Aprendizaje**: Pueden tener una curva de aprendizaje empinada.

Características de la Programación Funcional

1. **Funciones Matemáticas Puras**
   - Las funciones en programación funcional son puras.
   - No tienen efectos secundarios.
   - Siempre producen la misma salida para los mismos argumentos.
2. **Datos Inmutables**
   - Los datos no pueden ser modificados después de ser creados.
   - Facilita el razonamiento sobre el estado del programa.
   - Reduce la complejidad asociada con la gestión del estado.
3. **Menos Efectos Colaterales**
   - Evitar cambios en el estado global.
   - Hace que el comportamiento del programa sea más predecible.
   - Facilita la paralelización y la concurrencia.
4. **Declarativo**
   - Enfocado en el "qué" se quiere lograr más que en el "cómo".
   - Facilita la comprensión y el mantenimiento del código.
   - Permite al compilador optimizar el código de manera más efectiva.

Características de la Programación Funcional

5. **Fácil de Verificar**
   - La ausencia de efectos secundarios simplifica la verificación formal.
   - El comportamiento de las funciones es más fácil de predecir y testear.
   - Mejora la calidad y la fiabilidad del software.
6. **Funciones de Primera Clase y de Orden Superior**
   - Las funciones pueden ser tratadas como datos.
   - Las funciones pueden ser pasadas como argumentos y retornadas como resultados.
   - Facilita la creación de abstracciones y composiciones.
7. **Composición de Funciones**
   - Permite combinar funciones simples para construir funciones más complejas.
   - Fomenta el uso de pequeñas funciones reutilizables.
   - Mejora la modularidad y la reutilización del código.
8. **Transparencia Referencial**
   - Las expresiones pueden ser reemplazadas por sus valores sin afectar el comportamiento del programa.
   - Facilita el razonamiento y la optimización del código.
   - Aumenta la claridad y la coherencia del programa.

Suma de Dos Números en Haskell y Java

Haskell

```haskell
-- Suma dos numeros enteros
suma2 :: Int -> Int -> Int
suma2 x y = x + y

-- Funcion principal
main :: IO ()
main = do
    let a = 5
    let b = 7
    print (suma2 a b)
```

Java

```java
public class Suma2 {

    // Suma dos numeros enteros
    public static int suma2(int
    x, int y) {
        return x + y;
    }

    // Metodo principal
    public static void main(
    String[] args) {
        int a = 5;
        int b = 7;
        System.out.println(suma2
    (a, b));
    }
}
```

Funciones

Declaración de Funciones

Definición de una función

Una función en Haskell se declara mediante el tipo de función y la definición de la función especificando su nombre, seguido de los parámetros, el símbolo = y el cuerpo de la función.

Definición de la función:

$nombreFuncion :: tipo_1 \rightarrow tipo_2 \rightarrow \cdots \rightarrow tipo_n \rightarrow tipoResultado$

$nombreFuncion\ arg_1\ arg_2\ \ldots arg_n = expresion$

Aplicación de la función:

$nombre\ arg_1\ arg_2\ \cdots\ arg_n$

Por ejemplo, una función que suma dos números:

```
suma :: Int -> Int -> Int
suma x y = x + y
```

- `suma` es el nombre de la función.
- `x` y `y` son los parámetros de la función.
- `x + y` es el cuerpo de la función que realiza la suma.

Programa principal en Haskell

- En Haskell, el programa principal se define mediante la función **main**.
- Esta función es el punto de partida cuando se ejecuta un programa Haskell.
- La función **main** en Haskell es una función del tipo IO (), lo que significa que es una acción de entrada/salida.
- Dentro de main es posible usar la notación **do** para secuenciar múltiples acciones de entrada/salida.

```
main :: IO ()
main = do
  -- Acciones de entrada/salida
```

Función putStr en Haskell

putStr

- La función putStr tiene el tipo de datos String -> IO ().
- Toma una cadena de texto (String) y la imprime en la consola sin añadir una nueva línea al final.
- Se utiliza para imprimir texto sin un salto de línea al final.
- La variante **putStrLn** introduce un salto de línea al final del texto.

Ejemplo:

```
main :: IO ()
main = do
putStr "Hola Mundo"
putStr ("Hola Mundo")
putStr "Hola Mundo\n"
putStr ("Hola"++"Mundo")
putStr 2 --Error de tipo de dato
putStr (4+35) --Error de sintáxis
putStr 4+35 --Error de sintáxis
```

Función print en Haskell

putStr

- La función **print** tiene el tipo de datos **Show**, un tipo cuyo valor puede ser convertido a una cadena de texto.
- Toma un valor de cualquier tipo que sea una instancia de la clase de tipo **Show** y lo convierte en una representación textual para imprimirlo en la consola.
- Añade un salto de línea al final de la impresión.

Ejemplo:

```
main :: IO ()
main = do
print "Hola Mundo"
print ("Hola World")
print 2 --No existe un error de tipo de dato
print 2 + 9 --Error de sintáxis
print (2 + 9) --No existe error de sintáxis
print ("El resultado de sumar 10 y 5 es " ++ show (10+5))
print "El resultado de sumar 10 y 5 es " ++ show (10+5) --Error de
    sintáxis
print ("Texto 1" ++ "Texto 2")
```

Función getLine en Haskell

putStr

- La función **getLine** es una función de entrada estándar que lee una línea de texto desde la entrada del usuario.
- Tiene el tipo de IO String, lo que significa que devuelve una acción de entrada/salida que resulta en una cadena de texto (String).

Ejemplo:

```
main :: IO ()
main = do
putStrLn "Introduce tu nombre:"
nombre <- getLine
putStrLn ("Hola, " ++ nombre ++ "!")
```

Uso de `let` en Haskell

- **let** se utiliza para definir variables locales en una expresión.
- Sintaxis:

```
let <bindings> in <expression>
```

- Ejemplo:

```
let x = 5
    y = 6
in x * y
```

- El resultado de la expresión anterior es 30.

Uso de `where` en Haskell

- **where** se utiliza para definir variables locales al final de una declaración.
- Sintaxis:

```
<expression> where <bindings>
```

- Ejemplo:

```
square x = x * x
  where x = 5
```

- El resultado de `square 5` es 25.

Comparación de `let` y `where`

- **let**:
  - Se puede usar en cualquier expresión.
  - Más flexible en términos de alcance.
- **where**:
  - Se usa solo en definiciones de funciones o guardas.
  - Hace que el código sea más legible al separar las definiciones auxiliares.

Uso de `if` en Haskell

- **if** se utiliza para realizar evaluación condicional.
- Sintaxis:

```
if <condition> then <true-expression> else <false-expression>
```

- Ejemplo:

```
absoluteValue x = if x < 0 then -x else x
```

- El resultado de `absoluteValue (-5)` es 5.

Estructura del programa principal en Haskell

Organización del código 1:

```haskell
-- Función principal
main :: IO ()
main = do
  let res = suma 4 5
  putStrLn ("El resultado de sumar 4 y 5 es " ++ show res)

-- Funciones auxiliares
suma :: Int -> Int -> Int
suma x = x + x
```

Organización del código 1:

```haskell
-- Funciones auxiliares
suma :: Int -> Int -> Int
suma x = x + x

-- Función principal
main :: IO ()
main = do
  let res = suma 4 5
  putStrLn ("El resultado de sumar 4 y 5 es " ++ show res)
```

Estructura del programa principal en Haskell

Organización del código 3:

```
module Main where
import Lib (cuadrado)

main :: IO ()
main = do
  let resultado = cuadrado 28
  putStrLn ("El cuadrado de " ++ show numero ++ " es " ++ show
    resultado)

--En un archivo separado
module Lib (cuadrado,suma) where

cuadrado :: Int -> Int
cuadrado x = x * x

suma :: Int -> Int -> Int
suma x = x + y
```

Estructura del programa principal en Haskell

Organización del código 4:

```haskell
main :: IO ()
main = do
  let res = cuadrado 4
  putStrLn ("El cuadrado de 4 es: " ++ show res)
  where
    cuadrado x = x * x  -- Función local definida con where
```

Ejemplos básicos de funciones

```
suma :: Int -> Int -> Int
suma x y = x + y
```

```
multiplicar :: Int -> Int -> Int
multiplicar x y = x * y
```

```
restar :: Int -> Int -> Int
restar x y = x - y
```

```
dividir :: Float -> Float -> Float
dividir x y = x / y
```

```
cuadrado :: Int -> Int
cuadrado x = x * x
```

La instrucción *if*

Sintáxis básica:

```
if condición
  then expresión1
  else expresión2
```

Ejemplo 1:

```
main :: IO ()
main = do
  let x = 10
  let resultado = if x > 5
                  then "El número es mayor que 5"
                  else "El número es 5 o menor"
  putStrLn resultado
```

Ejemplo 2:

```
esMayorDeEdad :: Int -> String
esMayorDeEdad edad = if edad >= 18
                     then "Eres mayor de edad."
                     else "Eres menor de edad."
```

Ejemplo Combinado

```
eval x if numero > 0
       then putStrLn "El número es positivo."
       else if numero < 0
            then putStrLn "El número es negativo."
            else putStrLn "El número es cero."
```

- Uso de `let`, `where` e `if` en una función:

```
01 |        calculate x y = if result > 0
02 |                         then result
03 |                         else -result
04 |                    where
05 |                         result = let a = x * 2
06 |                         b = y * 3
07 |                         in a + b
08 |
```

- En este ejemplo, `let` se usa para definir `a` y `b`, `where` se usa para definir `result`, y `if` se usa para la evaluación condicional.

Guardas en Haskell

- Las guardas son expresiones booleanas que controlan el flujo de un programa.
- Permiten elegir entre diferentes resultados basados en condiciones.

Sintáxis:

```
funcion nombre parametros
    | condicion1 = expresion1
    | condicion2 = expresion2
    | otherwise  = expresionPorDefecto
```

- ‘condición1‘, ‘condición2‘, ...: Expresiones booleanas.
- ‘expresión1‘, ‘expresión2‘, ...: Resultados si la condición es verdadera.
- ‘otherwise‘: Caso por defecto.

Ejemplo Sencillo

```
signo :: Int -> String
signo x
    | x > 0     = "Positivo"
    | x < 0     = "Negativo"
    | otherwise = 'Cero"
```

- Si ‘x‘ es mayor que 0, devuelve ”Positivo”.
- Si ‘x‘ es menor que 0, devuelve ”Negativo”.
- Si ninguna condición se cumple, devuelve Çero”.

```
maxTres :: (Ord a) => a -> a -> a -> a
maxTres x y z
    | x >= y && x >= z = x
    | y >= x && y >= z = y
    | otherwise        = z
```

- Compara tres números y devuelve el mayor.

Ventajas de Usar Guardas

- **Legibilidad:** Las guardas hacen que el código sea más fácil de leer y entender, ya que las condiciones se expresan de manera clara y concisa.

- **Expresividad:** Permiten expresar lógica compleja de manera más directa que anidar múltiples `if-then-else`.

- **Mantenimiento:** Facilitan el mantenimiento del código, ya que las condiciones están centralizadas y es fácil añadir o modificar casos.

Tipos Básicos

- **Enteros (Integer, Int)**
  - **Integer**: Representa enteros de precisión arbitraria.

```
exampleInteger :: Integer
exampleInteger = 12345678901234567890123456789O
```

  - **Int**: Enteros de tamaño fijo.

```
exampleInt :: Int
exampleInt = 123456
```

- **Flotantes (Float, Double)**
  - **Float**: Números de punto flotante de precisión simple.

```
exampleFloat :: Float
exampleFloat = 3.14
```

  - **Double**: Números de punto flotante de precisión doble.

```
exampleDouble :: Double
exampleDouble = 3.141592653589793
```

Tipos Básicos

- **Caracteres (Char)**

```
exampleChar :: Char
exampleChar = 'a'
```

- **Cadenas de texto (String)**

```
exampleString :: String
exampleString = "Hello, Haskell!"
```

- **Listas (List)**

```
exampleList :: [Int]
exampleList = [1, 2, 3, 4, 5]
```

- **Tuplas (Tuple)**

```
exampleTuple :: (Int, String, Bool)
exampleTuple = (1, "Hello", True)
```

Introducción

- Un operador infijo en Haskell es una función binaria que se escribe entre sus dos argumentos.
- Haskell permite tanto operadores infijos predefinidos como personalizados.
- Cualquier función puede ser usada como operador infijo mediante el uso de acentos graves (`).

Ejemplos:

```
-- Suma
x = 5 + 3  -- x sera 8

-- Producto
y = 4 * 2  -- y sera 8

-- Concatenacion de listas
z = [1, 2] ++ [3, 4]  -- z sera [1, 2, 3, 4]
```

Definir Operadores Infijos Personalizados

Es posible definir operadores propios infijos utilizando símbolos. Por ejemplo, es posible definir un operador infijo *** para multiplicar dos números y luego sumar uno al resultado:

```haskell
-- Definir el operador infijo ***
(***) :: Int -> Int -> Int
a *** b = (a * b) + 1

-- Usar el operador infijo ***
result = 3 *** 4  -- result sera 13
```

Uso de Funciones como Operadores Infijos

Cualquier función puede ser utilizada como un operador infijo si se coloca entre acentos graves.
Ejemplo:

```haskell
-- Definir una funcion normal
add :: Int -> Int -> Int
add a b = a + b

-- Usar la funcion como un operador infijo
result = 5 `add` 7  -- result sera 12
```

Listas

Definición y Sintaxis

- Una lista en Haskell es una secuencia ordenada de elementos del mismo tipo.
- Se definen utilizando corchetes [] y los elementos se separan por comas.

```
-- Lista de enteros
[1, 2, 3, 4, 5]

-- Lista de caracteres (cadena de texto)
['a', 'b', 'c']

-- Lista de cadenas de texto
["Hola", "Mundo"]
```

Operaciones Básicas

- Concatenación: operador ++

```
-- Concatenacion de listas
[1, 2, 3] ++ [4, 5, 6]
-- Resultado: [1, 2, 3, 4, 5, 6]
```

- Acceso a Elementos: operador !!

```
-- Acceso al tercer elemento (indice 2)
[1, 2, 3, 4, 5] !! 2
-- Resultado: 3
```

- Cabeza y Cola: funciones `head` y `tail`

```
-- Primer elemento
head [1, 2, 3, 4, 5]
-- Resultado: 1

-- Todos los elementos excepto el primero
tail [1, 2, 3, 4, 5]
-- Resultado: [2, 3, 4, 5]
```

Operaciones Básicas

- Longitud: función `length`

```
-- Longitud de la lista
length [1, 2, 3, 4, 5]
-- Resultado: 5
```

- Inversión: función `reverse`

```
-- Lista invertida
reverse [1, 2, 3, 4, 5]
-- Resultado: [5, 4, 3, 2, 1]
```

Listas Comprehensivas

- Sintaxis para construir listas de manera declarativa.

```
-- Lista de los primeros 10 numeros al cuadrado
[x^2 | x <- [1..10]]
-- Resultado: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

-- Lista de pares (x, y) tal que 1 <= x <= 3 y 1 <= y <= 2
[(x, y) | x <- [1..3], y <- [1..2]]
-- Resultado: [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]
```

Introducción a Listas Infinitas

- Haskell soporta listas infinitas gracias a su evaluación perezosa.
- Permiten definir secuencias infinitas de manera segura.

Uso de repeat:

```haskell
repeat :: a -> [a]
repeat x = xs where xs = x : xs

-- Ejemplo de uso
infiniteOnes = repeat 1
-- infiniteOnes = [1, 1, 1, 1, ...]
```

Definición de Listas Infinitas

Uso de cycle

```haskell
cycle :: [a] -> [a]
cycle [] = error "empty list"
cycle xs = xs' where xs' = xs ++ xs'

-- Ejemplo de uso
infiniteCycle = cycle [1, 2, 3]
-- infiniteCycle = [1, 2, 3, 1, 2, 3, ...]
```

Uso de iterate

```haskell
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

-- Ejemplo de uso
infinitePowersOfTwo = iterate (*2) 1
-- infinitePowersOfTwo = [1, 2, 4, 8, 16, ...]
```

Definición de Listas Infinitas

Uso de enumFrom

```
enumFrom :: (Enum a) => a -> [a]
enumFrom x = x : enumFrom (succ x)

-- Ejemplo de uso
infiniteNaturals = [1..]
-- infiniteNaturals = [1, 2, 3, 4, 5, ...]
```

Tomar N Elementos:

```
take :: Int -> [a] -> [a]
take 5 infiniteNaturals
-- Resultado: [1, 2, 3, 4, 5]
```

Uso de Listas Infinitas

Map y Filter

```haskell
map (*2) (take 5 infiniteNaturals)
-- Resultado: [2, 4, 6, 8, 10]

filter even infiniteNaturals
-- Esto crea una lista infinita de numeros pares: [2, 4, 6, 8, 10, ...]
```

Ejemplo Completo

```haskell
-- Definir una lista infinita de numeros naturales
naturals :: [Integer]
naturals = [1..]

-- Tomar los primeros 10 numeros naturales
firstTenNaturals :: [Integer]
firstTenNaturals = take 10 naturals

-- Definir una lista infinita de potencias de 2
powersOfTwo :: [Integer]
powersOfTwo = iterate (*2) 1

-- Tomar las primeras 10 potencias de 2
firstTenPowersOfTwo :: [Integer]
firstTenPowersOfTwo = take 10 powersOfTwo

-- Definir una lista infinita de numeros pares
evenNumbers :: [Integer]
evenNumbers = filter even naturals

-- Tomar los primeros 10 números pares
firstTenEvenNumbers :: [Integer]
firstTenEvenNumbers = take 10 evenNumbers
```

Paquete de Haskell: Prelude

Funciones Matemáticas Básicas

- `(+)`, `(-)`, `(*)`, `(/)`: operadores para suma, resta, multiplicación y división.
- `abs`: valor absoluto.
- `signum`: signo de un número.
- `fromIntegral`: conversión entre tipos de datos numéricos.
- `sqrt`: raíz cuadrada.
- `exp`: exponencial.
- `log`: logaritmo natural.
- `logBase`: logaritmo en una base específica.
- `pi`: valor de π.
- `e`: valor de e (la base de los logaritmos naturales).

Funciones para Listas

- `head`: primer elemento de una lista.
- `tail`: lista sin su primer elemento.
- `last`: último elemento de una lista.
- `init`: lista sin su último elemento.
- `length`: longitud de una lista.
- `null`: verifica si una lista está vacía.
- `reverse`: invierte una lista.
- `take`: toma los primeros n elementos de una lista.
- `drop`: elimina los primeros n elementos de una lista.
- `splitAt`: divide una lista en dos partes en la posición n.

Funciones para Listas

- `elem`: verifica si un elemento está en una lista.
- `notElem`: verifica si un elemento no está en una lista.
- `filter`: filtra los elementos de una lista según un predicado.
- `map`: aplica una función a cada elemento de una lista.
- `concat`: une una lista de listas en una sola lista.
- `concatMap`: aplica una función que devuelve una lista a cada elemento y concatena los resultados.
- `foldl`: reducción de una lista desde la izquierda con una función acumuladora.
- `foldr`: reducción de una lista desde la derecha con una función acumuladora.
- `scanl`: reducción de una lista desde la izquierda, pero devuelve una lista de todos los resultados intermedios.
- `scanr`: reducción de una lista desde la derecha, con resultados intermedios.

Funciones de Entrada y Salida

- `print`: imprime un valor en la salida estándar.
- `putStrLn`: imprime una cadena con un salto de línea al final.
- `getLine`: lee una línea de entrada del usuario.
- `read`: convierte una cadena a un valor de tipo específico.
- `show`: convierte un valor a una cadena de caracteres.

Funciones de Manipulación de Cadenas

- `++`: concatenación de cadenas.
- `takeWhile`: toma los elementos de una lista mientras un predicado es verdadero.
- `dropWhile`: elimina los elementos de una lista mientras un predicado es verdadero.
- `span`: divide una lista en una tupla de dos listas según un predicado.
- `break`: similar a `span`, pero devuelve la primera parte donde el predicado falla.
- `lines`: divide una cadena en una lista de líneas.
- `words`: divide una cadena en una lista de palabras.
- `unlines`: une una lista de cadenas en una sola cadena con saltos de línea.
- `unwords`: une una lista de palabras en una sola cadena.

Funciones Booleanas y de Comparación

- `(==)`, `(/=)`, `(<)`, `(<=)`, `(>)`, `(>=)`: Operadores de comparación.
- `not`: negación lógica.
- `(&&)`: conjunción lógica.
- `(||)`: disyunción lógica.
- `otherwise`: valor que representa la verdad en cualquier caso, generalmente usado en guardas.

Funciones Generales

- `id`: función identidad que devuelve su argumento.
- `const`: devuelve una función que siempre devuelve un valor específico.
- `flip`: invierte los argumentos de una función.
- `($)`: operador de aplicación de funciones.
- `(.)`: composición de funciones.
- `error`: genera un error con un mensaje específico.

Funciones para Tuplas

- `fst`: primer elemento de una tupla.
- `snd`: segundo elemento de una tupla.
- `curry`: convierte una función de una tupla a una función de dos argumentos.
- `uncurry`: convierte una función de dos argumentos en una función de una tupla.

Tipos de Datos

- `Maybe`: tipo que representa un valor que puede estar presente (`Just x`) o ausente (`Nothing`).
- `Either`: tipo que representa un valor que puede ser de uno de dos tipos (`Left x` o `Right y`).

Fin