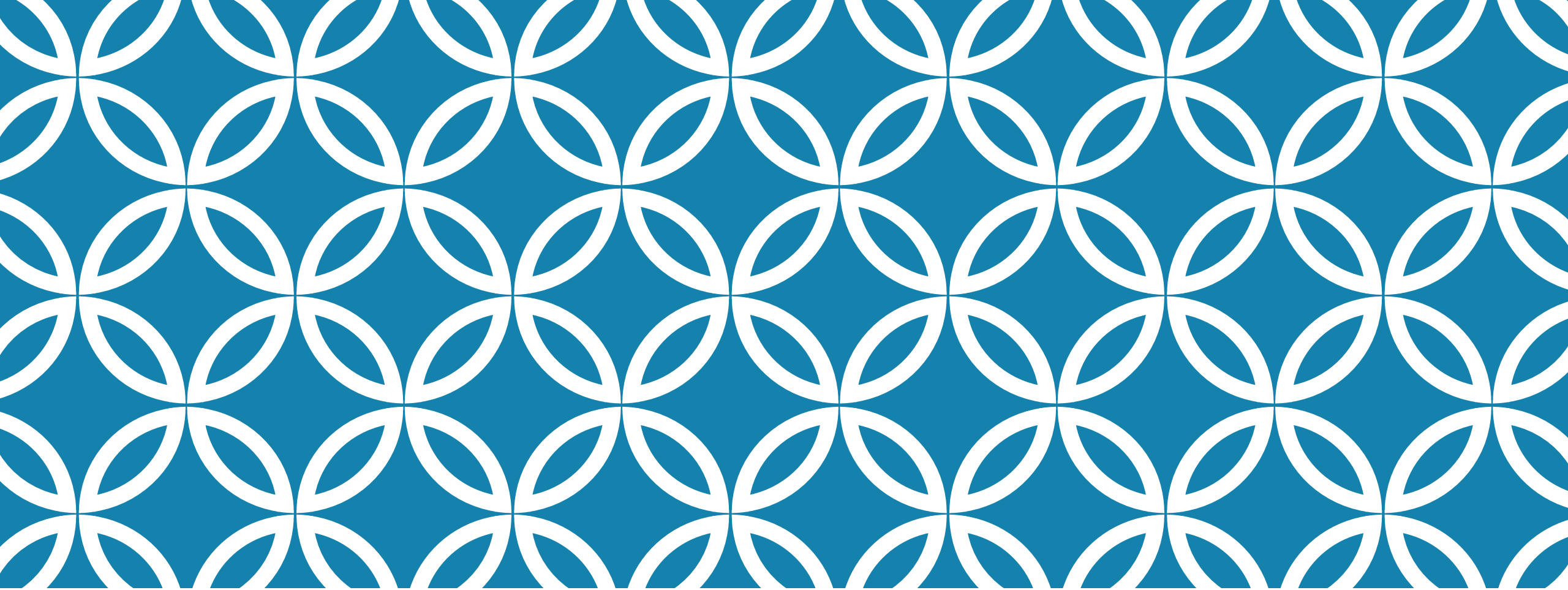

ESTRUCTURAS DISCRETAS

Guardias, Asignaciones locales , funciones de orden superior y tipos de datos.

Javier Enríquez Mendoza Mauricio E. Hernández Olvera

GUARDIAS

Pattern Matching es una forma de asegurarnos que un valor tenga cierta forma o este construido de cierta manera.

Pero hay casos en donde nos gustaría ver si el valor cumple alguna propiedad.

Hasta ahora hemos usado **if** para esto.

Pero cuando queremos comprobar varias condiciones las expresiones **if** se vuelven poco legibles.

Una alternativa mas legible y elegante son las **guardias.**

SINTAXIS

Las guardias se indican con con **pipes** que siguen al nombre de la función y sus parámetros.

Se identan con un espacio y están alineadas.

Una **guardia** es una expresión booleana, si se evalúa a **True,** entonces se ejecuta el cuerpo correspondiente.

Si se evalúa a **False,** entonces se comprueba la guardia siguiente y así sucesivamente.

Existe el caso **otherwise** que se ejecutara si ninguna de las guardias se evaluó a **True.** Tiene que ser el último caso. Es parecido al caso de **else** en un **if.**

Si todas las guardias se evalúan a **False** y no hay un caso de **otherwise** la ejecución sigue con el siguiente patrón.

EJEMPLO

Vamos a hacer una función que recibe 2 enteros y nos dice si el primero es menor, mayor o igual a segundo.

```
compara :: Int -> Int -> String
compara n m
 | n > m = “GT”
 | n < m = “LT”
 | n == m = “EQ”
```

```
compara2 :: Int -> Int -> String
compara2 n m
 | n > m = “GT”
 | n < m = “LT”
 | otherwise = “EQ”
```

ASIGNACIONES LOCALES

Hay ocasiones en donde utilizamos varias veces el mismo valor.

Se vuelve poco practico calcular el valor todas las veces que se va a usar.

Siempre buscamos escribir lo menos posible. Repetir código es una muy mala práctica.

Sería ideal calcular el valor una sola vez, asociarlo a una variable y usarlo las veces que fuera necesario en lugar de la expresión completa.

Hay dos formas de hacer esto, **where** y **let.**

WHERE

where nos permite usar variables en el cuerpo de una función y luego asignarles valores a éstas.

Va al final de la expresión.

Se puede usar para declarar varias variables.

Las variables que definamos en la sección **where** del cuerpo de una función solo tienen valor dentro de la función.

Las variables definidas con **where** no se comparten entre los cuerpos de los patrones de una función.

EJEMPLO

Sintaxis:

<expresión> **where** <definición>

Función que regresa las iniciales de un nombre.

```
iniciales :: String -> String -> String
iniciales  nombre apellido = [n] ++  ". " ++ [a] ++ "."
 where (n:_) = nombre
          (a:_) = apellido
```

LET

Las expresiones **let** al igual que **where** sirven para ligar variables a valores

La diferencia sintáctica es que mientras que en **where** las definiciones van al final del cuerpo de la funcion. **let** permite asignar los valores en cualquier momento.

Las variables definidas con **let** son accesibles solo en el cuerpo del ésta.

La principal diferencia respecto a **where** es que **let** es una expresiones por si misma.

Esto quiere decir que podemos usarlas en casi cualquier lugar.

EJEMPLO

Sintaxis:

let <definición> **in** <expresión>

Función que dice si una cadena es palíndroma.

```
palindromo :: String -> Bool
palindromo str =
    let rts = reverse str
    in str == rts
```

FUNCIONES SOBRE LISTAS

Las listas son la estructura de datos mas usada en programación funcional

Haskell tiene varias funciones predefinidas para listas. Algunas de ellas son:

- head
- tail
- last
- init
- length
- null
- reverse
- take
- drop
- maximum
- minimum
- sum
- product
- elem
- map
- filter

FUNCIONES DE ORDEN SUPERIOR

En programación funcional las funciones son de primera clase.

Esto quiere decir que pueden pasarse como parámetros o regresarse en una función.

Las funciones de orden superior son aquellas que reciben como parámetro o regresan a otra función.

MAP

Toma una función y una lista y aplica esa función a cada elemento de esa lista, produciendo una nueva lista.

map (x 5) [1,2,3,4,5,6]

[5,10,15,20,25,30]

map (++ “!”) [“LOL”, “BANG”, “POW”]

[“LOL!”, “BANG!”, “POW!”]

map fst [(1,2), (3,4), (5,6), (7,8)]

[1,3,5,7]

FILTER

Es una función que toma un predicado (una función que devuelve un valor booleano) y una lista y devuelve una lista con los elementos que satisfacen el predicado.

filter (> 3) [1,2,3,4,5,6,7,8,9,0]

[4,5,6,7,8,9]

filter even (take 50 [1..])

[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50]

TIPOS DE DATOS

Un tipo de dato es una abstracción de un conjunto de valores.

Tiene asociadas un conjunto de operaciones, restricciones o propiedades.

Ejemplo :

Tipo de dato: Bool

Conjunto de valores: { True, False}

Operaciones asociadas: && || not

Restriccion: Las operaciones son cerradas.

NUESTROS TIPOS DE DATOS

En Haskell tenemos tipos de datos primitivos.

Hay ocasiones en las que queremos trabajar con tipos de datos específicos que no están definidos en el núcleo del lenguaje.

Nosotros podemos definir nuestros propios tipos de datos a partir de los tipos primitivos.

Las definiciones de tipos de datos pueden ser recursivas.

SINTAXIS

Para definir tipos de datos en Haskell existe `data`

```
data = Constructor1 | Constructor2 | ...
```

Como ejemplo vamos a definir el tipo de dato Figura que va a representar a algunas figuras geométricas.

- Circulo
- Cuadrado
- Rectángulo
- Triángulo