Reduciendo el tamaño de tu código

Tener código con funciones que mandan a llamar funciones que mandan a llamar funciones que mandan a ... puede ser un poco engorroso. Para escribirlo de una manera más legible tenemos varias palabras reservadas que nos ayudan con eso:

El where

where es solo una forma de renombrar expresiones para poder utilizarlas en el codigo. Tomemos como ejemplo la siguiente función en la que funcion1 manda a llamar a funcion2 con el resultado de otraFuncion como primer parámetro:

```
funcion1 n = funcion2 (otraFuncion n 10 203 "parametroString" true) n
```

Se ve un poco engorroso, en especial porque se empieza a desdibujar donde empieza una llamada y termina otra. Ahí es donde entra el where

```
funcion1 n = funcion2 param2 n
    where param2 = (otraFuncion n 10 203 "parametroString" true)
```

Así, el código queda mucho más claro.

Cosas a tomar en cuenta del where

- Es una palabra que puedes usar "hacia abajo", es decir, cosas que nombres primero dentro del where puedes utilizarlas después (dentro del mismo where)

  ```
  Correcto:
  funcion1 n = funcion2 param4 n
      where param2 = (otraFuncion n 10 203 "parametroString" true)
          param3 = unaFuncionAuxiliarExtra param2 123 false "parametroString"
          param4 = div param3 10

  Incorrecto:
  funcion1 n = funcion2 param4 n
      where param2 = (otraFuncion param3 10 203 "parametroString" true)
          param3 = unaFuncionAuxiliarExtra param2 123 false "parametroString"

  Porque aún no definimos param3
  ```

- Si haces las cosas con where, puedes terminar definiendo funciones enteras ahí dentro.

  ```
  funcion1 n = funcion2 n
      where funcion2 :: Int -> String
            funcion2 0 = "cero"
            funcion2 1 = "uno"
            funcion2 n = error "No es 0 ni 1"
  ```

- Es importante la identación.

  1. El where debe de ir identado dentro de la función que lo utiliza

2. Las palabras que estén dentro del where deben de estar a la misma altura (misma columna)

```
Correcto:

funcion1 :: Int -> String
funcion1 n = funcion2 n
    where
        funcion2 :: Int -> String
        funcion2 0 = "cero"
        funcion2 1 = "uno"
        funcion2 n = error "No es uno ni cero"

funcion1 :: Int -> String
funcion1 n = funcion2 param2
    where
        funcion2 :: Int -> String
        funcion2 0 = "cero"
        funcion2 1 = "uno"
        funcion2 n = error "No es uno ni cero"
        param2 = mod n 2

funcion1 :: Int -> String
funcion1 n = funcion2 param4
    where
        param2 = mod n 2

Incorrecto:

funcion1 :: Int -> String
funcion1 n = funcion2 param4
    where
        param2 = mod n 2
            param3 = param2 + 1

funcion1 :: Int -> String
funcion1 n = funcion2 param4
where
        param2 = mod n 2
        param3 = param2 + 1
```

3. Si usas guardas, el where será visible para todos los casos.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
        skinny = 18.5
```

```
        normal = 25.0
        fat = 30.0
```

4. También puedes trabajarlo como tuplas.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

El operador Let

let no se comparte entre guardas. Se utiliza de la siguiente manera.

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Nota que **let** e **in** están identados dentro de la función. Como su nombre lo dice, simplemente es nombrar una expresion para utilizarla en un bloque de código.

Diferencias entre let y where.

Let y where parecen iguales, excepto que uno va antes de usarlo y el otro "después", pero **where** es solo una ayuda sintáctica, mientras que **let** es una expresión por sí misma. Por lo que podemos tener cosas como :

```
    ghci> 4 * (let a = 9 in a + 1) + 2
    42

    ghci> let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in
foo ++ bar)
    (6000000,"Hey there!")
```

Aqui se usan los ; para separar cada nombramiento.

El @

También se puede utilizar el @ en caso de que uno de los parámetros que recibe la función resulta muy largo y tienes que usarlo constantemente Lo que esta del lado izquierdo del @ será el "apodo" y lo que esta del lado derecho será a lo que se refiere el "apodo".

```
Con el @
funcionQueUsaAlDos :: Nat -> OtroTipo
funcionQueUsaAlDos dos@(Suc (Suc Cero)) = funcionQueUsaAlDos (Suc (Suc Cero))
```

```
Sin el @
funcionQueUsaAlDos :: Nat -> OtroTipo
funcionQueUsaAlDos (Suc (Suc Cero)) = funcionQueUsaAlDos (Suc (Suc Cero))
```

Asi ya no tenemos que referirnos al (Suc (Suc Cero)), sino solo al "dos". Quizá suene tonto en este momento, pero cuando empiecen a hacer una caza de patrones más refinada, ya no tendrán que repetir tanto código.

Finalmente, estas son solo sugerencias de estilo y aunque recomiendo usarlas, si no les sale o les parecen que están de más, pueden no usarlas.