Procesamiento funcional

# DE UN DATASET DE KAGGLE

Gael Valerio Peralta

El dataset seleccionado para este caso fue el de

# CONJUNTO DE DATOS DE ANÁLISIS DEL MERCADO DE VEHÍCULOS USADOS DE BMW

el cual puede ser encontrado en la página de Kaggle bajo la siguiente liga:

- https://www.kaggle.com/datasets/algozee/bmw-dataset

```
model,year,price,transmission,mileage,fuelType,tax,mpg,eng
 5 Series,2014,11200,Automatic,67068,Diesel,125,57.6,2.0
 6 Series,2018,27000,Automatic,14827,Petrol,145,42.8,2.0
 5 Series,2016,16000,Automatic,62794,Diesel,160,51.4,3.0
 1 Series,2017,12750,Automatic,26676,Diesel,145,72.4,1.5
 7 Series,2014,14500,Automatic,39554,Diesel,160,50.4,3.0
 5 Series,2016,14900,Automatic,35309,Diesel,125,60.1,2.0
 5 Series,2017,16000,Automatic,38538,Diesel,125,60.1,2.0
 2 Series,2018,16250,Manual,10401,Petrol,145,52.3,1.5
 4 Series,2017,14250,Manual,42668,Diesel,30,62.8,2.0
 5 Series,2016,14250,Automatic,36099,Diesel,20,68.9,2.0
 X3,2017,15500,Manual,74907,Diesel,145,52.3,2.0
 1 Series,2017,11800,Manual,29840,Diesel,20,68.9,2.0
 X3,2016,15500,Automatic,77823,Diesel,125,54.3,2.0
 2 Series,2015,10500,Manual,31469,Diesel,20,68.9,2.0
```

# EXPLICACIÓN

Columnas seleccionadas:

- Debido a que en este dataset existen más de 5 columnas, las propuestas para tarabajar en este caso fueron:
- Columna NUM1: Precio
- Columna NUM1: Kilometraje
- Columna TEXT: Transmisión

Valores usados:

- Se tomo como elección los siguientes valores:
- L (Longitud): 10
- Pad_Char: * (Signo de asterisco)

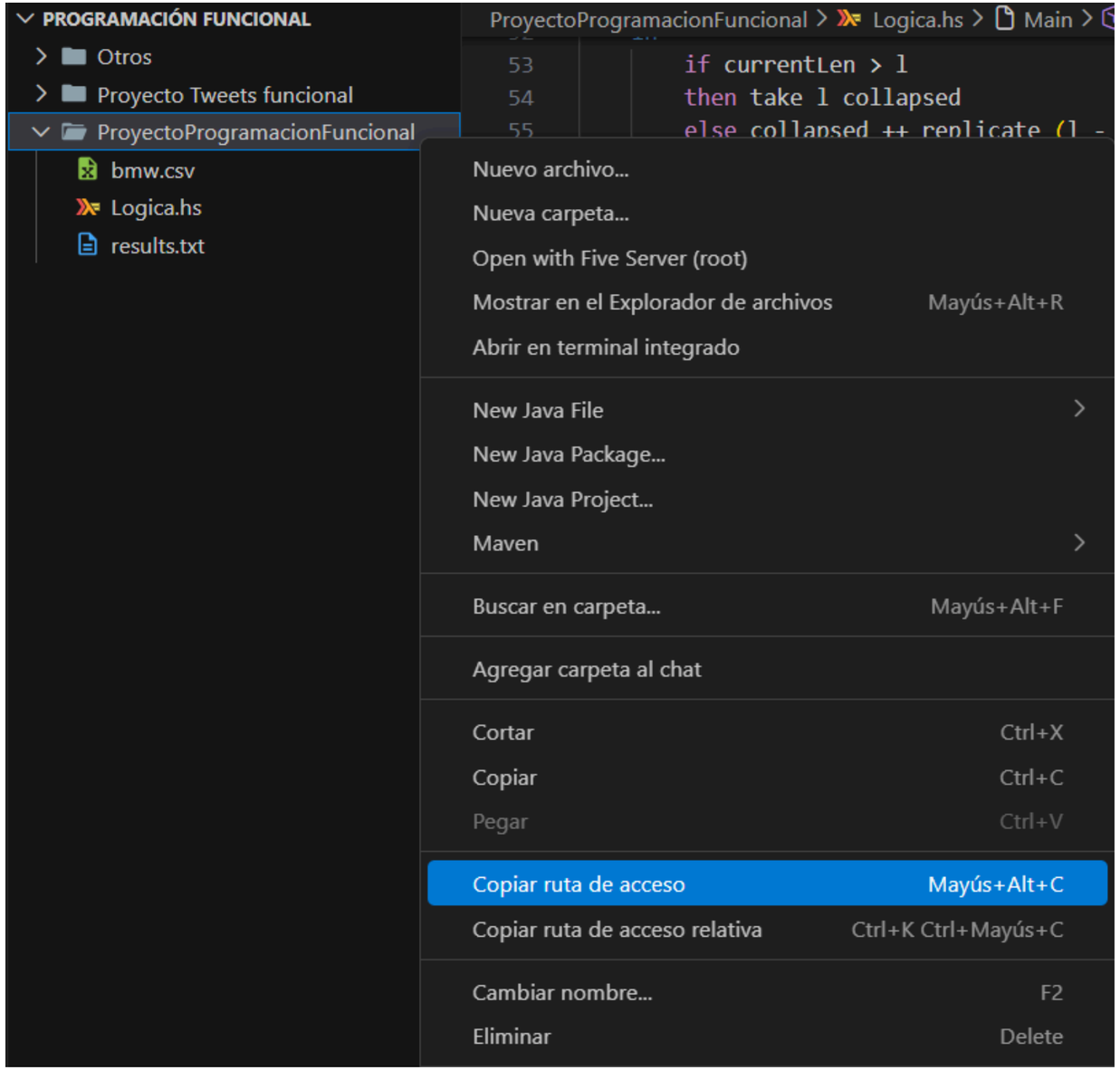
# EJECUCIÓN

Primeramente, una vez que estemos en la carpeta en donde tenemos nuestro archivo de ejecución (en este caso, Logica.hs), lo que haremos es que obtendremos la ruta exacta de la carpeta en donde se encuentra. En este caso usando VS Code:

- Daremos click derecho en la carpeta en donde tenemos el archivo

- Una vez hecho, se nos muestra un panel con distintas opciones, en donde buscaremos la opcion “Copiar ruta de acceso”, y la seleccionamos

- Después, ya tendremos nuestra dirección de la carpeta en el portapapeles



# EJECUCIÓN

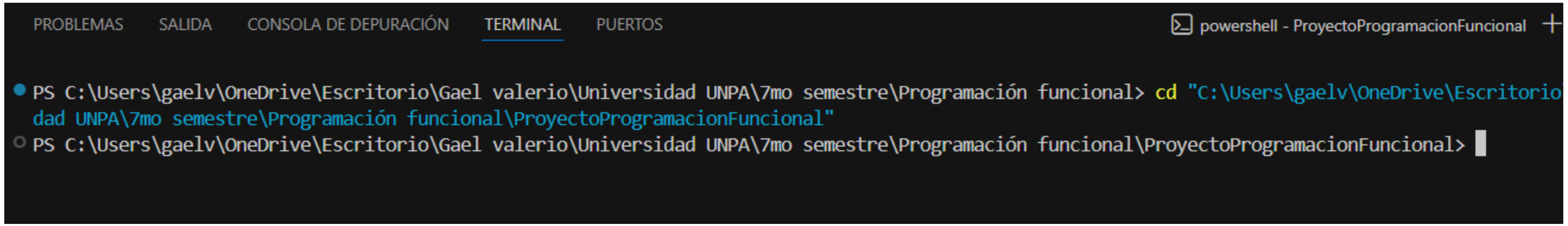
Ahora, en la terminal de VS Code, escribiremos el siguiente comando:

- cd “ ”

Y dentro de las comillas, pegaremos la ruta de acceso que copiamos anteriormente

- cd “C:\Users\gaelv\OneDrive\Escritorio\...”

Después oprimiremos la tecla Enter (Esto permitira al sistema ubicarse en la carpeta correcta para ejecutar nuestro archivo)



# EJECUCIÓN

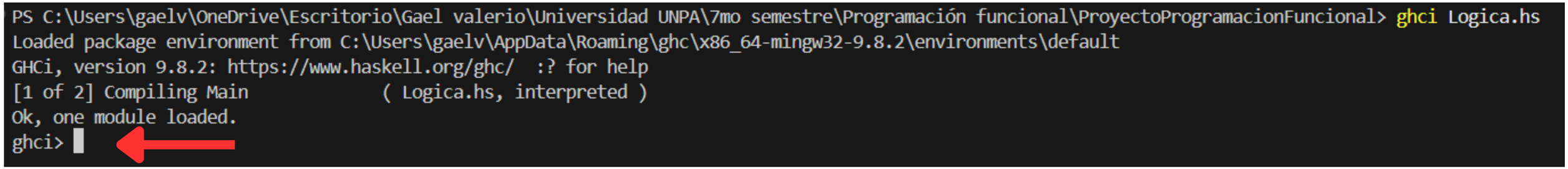
Ahora, ya en la carpeta correcta, escribiremos lo siguiente:

- ghci nombreDelArchivo.hs

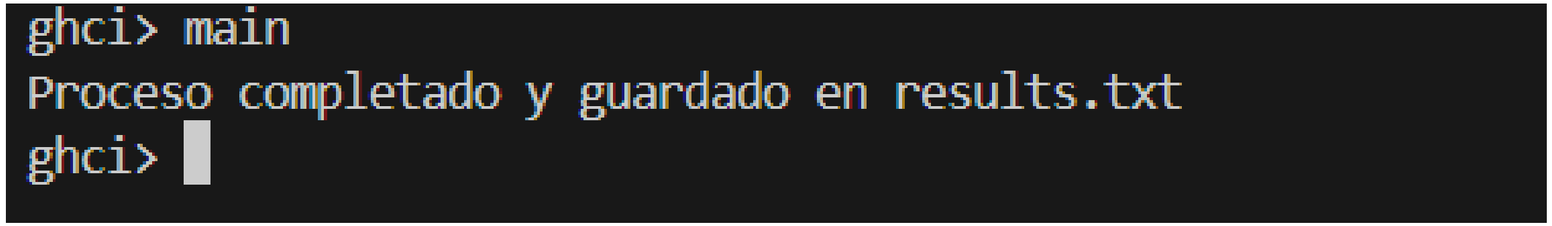
Por lo que en este caso, ya que el archivo se llama Logica.hs, entonces:

- ghci Logica.hs

Por lo que después, nos aparecerá un mensaje de confirmación en la misma terminal:




Por último, en la consola con referencia a ghci (ghci>), escribiremos unicamente “main”, por lo que se ejecutará el programa, y listo:



# INTERPRETACIÓN (FUNCIONES PRINCIPALES)

¿Qué hace?

Aquí se recibe una lista de números decimales, que primeramente se asignan los valores calculados a un objeto de tipo Stats, ya que después se harán cada uno de los cálculos, obteniendo y asignando los valores que se obtuvieron a las variables principales (líneas 4-9), logrando organizar una lista en una estructura de datos

```haskell
computeStats :: [Double] -> Stats
computeStats [] = error "No hay datos para calcular las estadisticas"
computeStats numbers = Stats
    { sSum    = suma
    , sMean   = promedio
    , sStd    = desviacionEstandar
    , sMedian = mediana
    , sMin    = minimum numbers
    , sMax    = maximum numbers
    }
  where
    n = fromIntegral (length numbers)
    suma = sum numbers
    promedio = suma / n
    varianza = sum [ (x - promedio)^2 | x <- numbers ] / n
    desviacionEstandar = sqrt varianza
    sortedNumbers = sort numbers
    mid = length numbers `div` 2
    mediana = if odd (length numbers)
              then sortedNumbers !! mid
              else (sortedNumbers !! (mid - 1) + sortedNumbers !! mid) / 2
```

# INTERPRETACIÓN (FUNCIONES PRINCIPALES)

¿Qué hace?

Esta función recibe como tal un texto, una longitud base y un caracter a utilizar, para después transformar cualquier cadena de entrada, en un formato estándar predefinido en el main, en donde inicia convirtiendo la variable en mayúsculas, para después eliminar los espacios en blanco, para finalmente verificar si el texto cumple con la longitud deseada, sino, tendrá que ser o recortado, o en su defecto, autocompletado, garantizando así siempre las condiciones propuestas para estandarizarlo.

```haskell
transformText :: String -> Int -> Char -> String
transformText text l padChar =
    let
        upperText = map toUpper text
        collapsed = unwords (words upperText)
        currentLen = length collapsed
    in
        if currentLen > l
        then take l collapsed
        else collapsed ++ replicate (l - currentLen) padChar
```

# EXPLICACIÓN (RESTO DEL CÓDIGO)

¿Qué hace?

Primeramente, se importan las librerias necesarias para hacer las distintas operaciones, como lo son el manejo de listas y formatos.

Después, en la estructura Auto, se crea una estructura para capturar los datos crudos del archivo CSV, mientras que con Stats, se diseña un contenedor para guardar los resultados finales acerca de los análisis estadísticos, permitiendo un optimo manejo de la información. En otras palabras, Auto es una estructura con datos de entrada, mientras que Stats con datos de salida.

```haskell
module Main where

import Data.List (sort)
import Text.Printf (printf)
import Data.Char (toUpper)

data Auto = Auto
    { precio       :: String
    , transmision  :: String
    , kilometraje  :: String
    } deriving Show

data Stats = Stats
    { sSum    :: Double
    , sMean   :: Double
    , sStd    :: Double
    , sMedian :: Double
    , sMin    :: Double
    , sMax    :: Double
    } deriving Show
```

# EXPLICACIÓN (RESTO DEL CÓDIGO)

```haskell
formatearReporte :: String -> Stats -> String
formatearReporte titulo s =
    printf "\nEstadisticas de %s \n" titulo ++
    printf "Suma Total:      %.2f\n" (sSum s) ++
    printf "Promedio (Mean):  %.2f\n" (sMean s) ++
    printf "Desv. estandar:  %.2f\n" (sStd s) ++
    printf "Mediana:         %.2f\n" (sMedian s) ++
    printf "Minimo:          %.2f\n" (sMin s) ++
    printf "Maximo:          %.2f\n" (sMax s) ++
    "\n"

splitOnComma :: String -> [String]
splitOnComma "" = []
splitOnComma s =
    let (extraido, resto) = break (== ',') s
    in extraido : case resto of
                    [] -> []
                    (_:r) -> splitOnComma r
```

¿Qué hace?

La primer función lo que hace es que sirve como un estándar estético que presenta los datos obtenidos de una forma más clara, sencilla y legible. Por el contrario, la segunda función recorre cada una de las lineas del archivo CSV, cortando por cada una de las coma´s encontradas para que puedan ser procesadas de forma más sencilla en una lista,

# EXPLICACIÓN (RESTO DEL CÓDIGO)

¿Qué hace?

Esta función actúa como un controlador, ya que verifica si la línea cumple con las columnas completas, para después extraer estas con las que trabajaremos y creando un objeto Auto, por lo que si están incompletas, se devuelve un Nothing, y así se evita que falle el programa. Además, se declaran los valores que usaremos para definir la estructura, como lo es la longitud necesaria y el carácter predeterminado

```haskell
procesarFila :: [String] -> Maybe Auto
procesarFila columnas
    | length columnas >= 5 = Just $ Auto (columnas !! 2)
      (columnas !! 3) (columnas !! 4)
    | otherwise = Nothing

--variables constantes
longitudL :: Int
longitudL = 10

caracterPad :: Char
caracterPad = '*'
```

# EXPLICACIÓN (RESTO DEL CÓDIGO)

¿Qué hace?

Aquí se logra la coordinación de todos los elementos, ya que el archivo se lee, los datos se almacenan temporalmente, para después hacer una comprensión en la lista para convertir cada línea del texto a un objeto Auto, para finalmente usar la función de la línea 11, que por medio de un lambda se recorren todos los elementos, aplicando los cambios y haciendo una lista comparativa de como quedó el texto antes y después de la transformación

```haskell
main :: IO ()
main = do
    contenido <- readFile "bmw.csv"
    let todasLasLineas = lines contenido
    let datosSinEncabezado = drop 1 todasLasLineas

    let listaAutos = [auto | linea <- datosSinEncabezado,
                             let columnas = splitOnComma linea,
                             Just auto <- [procesarFila columnas]]

    let transmisionesLimpias = map (\a ->
            let original = transmision a
                transformada = transformText original longitudL caracterPad
            in printf "%-12s -> %s" original transformada :: String
          ) listaAutos
```

# EXPLICACIÓN (RESTO DEL CÓDIGO)

```haskell
--Se hacen los calculos estadisticos
    let precios = [read (precio a) :: Double | a <- listaAutos]
    let statsPrecio = computeStats precios
    let kms = [read (kilometraje a) :: Double | a <- listaAutos]
    let statsKms = computeStats kms


    --Se guarda todo en un reporte
    let reporteStats = formatearReporte "precio" statsPrecio ++
                       formatearReporte "kilometraje" statsKms

    let reporteTransmisiones = "\n TEXTO TRANSFORMADO:\n" ++
                               "Original -> Transformado (Longitud="
                               ++ show longitudL ++ ", padChar " ++
                                show caracterPad ++ ")\n" ++
                              unlines transmisionesLimpias

    writeFile "results.txt" (reporteStats ++ reporteTransmisiones)

    putStrLn "Proceso completado y guardado en results.txt"
```

¿Qué hace?

Primeramente, se extraen las columnas de precios y kilometros de listaAutos, convirtiéndolas en listas decimales, para pasarlas por la función de análisis estadísticos. Después, bajo la lógica de formateo, se unen los resultados numéricos con los datos transformados, para finalmente. guardar todo en un archivo “results.txt”, concluyendo con un mensaje de confirmación en la consola

# COMPARATIVA EN ARQUITECTURA FUNCIONAL

| Función | ¿Es pura? | ¿Usa orden superior? | Responsabilidad principal |
|---|---|---|---|
| computeStats | Si | Sí<br>map, sum | Realiza las estadísticas matemáticas y calculos necesarios |
| transformText | Si | Sí<br>map | Normaliza, formatea y transforma el texto |
| splitOnComma | Si | No, porque es recursiva | Realiza la segmentación de cadenas |
| main | No | Sí (lambdas) | Permite el control de flujo (Input/Putput) con el resto de los elementos |
| procesarFila | Sí | No | Valida y convierte pedazos de texto en un objeto Auto |
| formatearReporte | Sí | No | Convierte resultados numéricos en texto legible para el reporte |

# COMPARATIVA DE ENTRADA Y SALIDA DE DATOS

| Función | Entrada | Salida | Transformación que realiza |
|---|---|---|---|
| splitOnComma | String (una línea del CSV) | Una lista de tipo [String] de palabras | Una la coma como un delimitador del texto para poder identificar elementos |
| procesarFila | Una lista de tipo [String] de palabras | Maybe Auto | Valida cada una de las columnas y las transforma en un elemento Auto |
| transformText | Un String de la columna TEXT (en este caso, elemento “transmisión”) | Un elemento String transformado | Estandariza el texto, asignando las transformaciones correspondientes (mayusculas, sin espacios) |
| computeStats | Una lista de tipo [Double](Precios y kms) | Registro Stats | Reduce los elementos para la obtención de sus estadísticas de cada columna |
| formatearReporte | Un registro Stats | String | Convierte los datos numéricos en un bloque de texto presentable |
| main | Archivos .csv | Archivo .txt | Conecta todas las funciones, además de administrar el almacenamiento |

# GRACIAS

Programación funcional