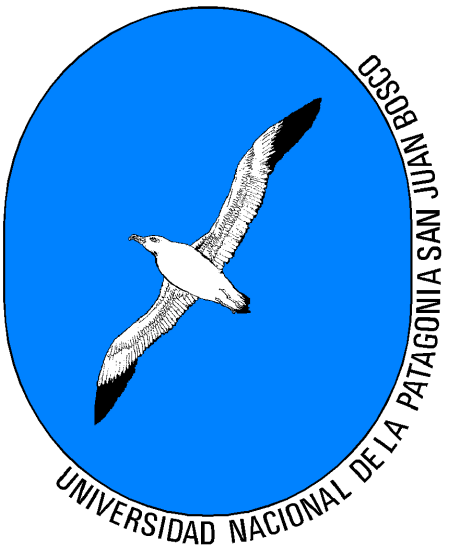
Laboratorio Programación Lógica

Paradigmas y Lenguajes de Programación - Trelew

Lic. Pablo Toledo Margalef Lic. Lautaro Pecile

Año 2024



Integrantes:

- Toro Santiago
- Perez Luciana
- Gonzalez Alejo

OBJETIVO 1

Modelar las personas que forman parte de un grupo de amigos:
Esta declaración define un tipo de dato 'persona' que se utiliza para representar a los amigos del grupo. Cada **`Persona`** tiene un campo `nombre` que identifica a esa persona. Y define clases del tipo haskell (`Show`, `Eq`, `Ord`) que permiten utilizar funciones de impresión, comparación y ordenamiento.

```haskell
data Persona = Persona {
    nombre :: String
} deriving (Show, Eq, Ord)
```

Objetivo 1.1

Modelar al grupo de amigos que tiene a Juan, Pedro y Santiago.
Los amigos los modelamos mediante el tipo de dato **Persona** con los nombres "Juan", "Pedro" y "Santiago" y se agrupan en una lista.

```haskell
main :: IO ()
main = do
    let juan = Persona "Juan" []
        pedro = Persona "Pedro" []
        santiago = Persona "Santiago" []
        amigos = [juan, pedro, santiago]
```

OBJETIVO 2

Modelar los gastos y la relación que se genera cuando una persona paga algo y se divide entre las demás personas. Tener en cuenta que una persona no puede deberse a sí misma.
La declaración de **Gasto** define un nuevo tipo de dato llamado **`Gasto`.** Este tipo tiene dos campos: **`pagador`,** que es de tipo `Persona`, y **`monto`,** que es de tipo `Float`. Es decir, un `Gasto` tiene quién pagó (representado por un objeto de tipo `Persona`) y cuánto se pagó (un valor de tipo `Float`).
La declaración de **Deuda** define otro tipo de dato llamado **`Deuda`.** `Este tipo` tiene tres campos: **`deudor`** y **`acreedor`,** ambos de tipo `Persona`, y **`montoDeuda`**, que es de tipo `Float`. Esto significa que una `Deuda` tiene quién debe (representado por `deudor`)y a quién se le debe (representado por `acreedor`) y cuánto es la deuda (un valor de tipo `Float`)

```haskell
-- Definici贸n del tipo de datos para representar un gasto
data Gasto = Gasto {
    pagador :: Persona,
    monto :: Float
} deriving (Show)


-- Definici贸n del tipo de datos para representar una deuda
data Deuda = Deuda {
    deudor :: Persona,
    acreedor :: Persona,
    montoDeuda :: Float
} deriving (Show,Eq,Ord)
```

OBJETIVO 2 .1

Modelar la siguiente situación: Juan pagó unos cafés de $60.
Las deudas quedarían así:
• Pedro le debe $20 a Juan
• Santiago le debe $20 a Juan

Para lograr este objetivo, se implementó la función:
crearDeudasDeGasto: Esta función toma un grupo de personas y un gasto, y genera una lista de deudas basada en ese gasto. Divide el monto del gasto entre el número total de personas en el grupo, asignando a cada persona, excepto al pagador, su parte proporcional del gasto como deuda al pagador.

```haskell
-- Recibe un grupo de personas y un gasto y crea deudas en base a ese gasto
crearDeudasDeGasto :: [Persona] -> Gasto -> [Deuda]
crearDeudasDeGasto amigos (Gasto pagador monto) =
    let montoPorPersona = monto / fromIntegral (length amigos)
        deudores = filter (\p -> p /= pagador) amigos
    in map (\deudor -> Deuda deudor pagador montoPorPersona) deudores
```

Con los siguientes datos en el main:

```haskell
main :: IO ()
main = do
    let juan = Persona "Juan"
        pedro = Persona "Pedro"
        santiago = Persona "Santiago"
        amigos = [juan, pedro, santiago]
        gasto1 = Gasto juan 60
```

```
Deudas completas:
Juan le debe $0.0 a Juan
Pedro le debe $20.0 a Juan
Santiago le debe $20.0 a Juan
```

OBJETIVO 3

Agregar ahora la capacidad de agregar gastos y tenerlos en cuenta a la hora de listar cuánto se deben entre sí. Tener en cuenta la situación donde una persona que le debía a otra realiza un gasto, disminuyendo (o saldando) su deuda con esa persona.

Para lograr este objetivo, se implementaron las siguientes funciones:

- **actualizarDeudas**: Esta función toma una lista de deudas y devuelve esa lista de deudas actualizada, donde se saldan las deudas que pueden ser saldadas entre los miembros del grupo.
- **eliminarDeudas**: Esta función recibe dos listas de deudas y elimina de la segunda lista todos los elementos que están presentes en la primera lista, devolviendo la segunda lista actualizada sin los elementos eliminados.

```haskell
-- Recibe una lista de deudas y devuelve esa lista de deudas actualizada
actualizarDeudas :: [Deuda] -> [Deuda]
actualizarDeudas deudas =
    let deudasEmparejadas = emparejarDeudas deudas
    in concatMap saldarDeudas deudasEmparejadas


-- Recibe dos lista de deudas y elimina de la segunda todos los elementos de la primera
eliminarDeudas :: [Deuda] -> [Deuda] -> [Deuda]
eliminarDeudas [] deudas = deudas
eliminarDeudas (x:xs) deudas = eliminarDeudas xs (filter (\d -> d /= x) deudas)
```

obtenerDeudasModificadas: Esta función toma dos listas de deudas y devuelve una lista con las deudas de la segunda lista que han sido modificadas en relación con la primera lista, es decir, las deudas que están presentes en ambas listas y han cambiado en algún aspecto.

```haskell
obtenerDeudasModificadas :: [Deuda] -> [Deuda] -> [Deuda]
obtenerDeudasModificadas deudas1 deudas2 = filter (\deuda2 -> (any (coincideDeuda deuda2) deudas1)) deudas2
  where
    coincideDeuda :: Deuda -> Deuda -> Bool
    coincideDeuda (Deuda deudor1 acreedor1 _) (Deuda deudor2 acreedor2 _) =
      (deudor1 == deudor2 || deudor1 == acreedor2) && (acreedor1 == deudor2 || acreedor1 == acreedor2)
```

emparejarDeudas: Esta función recibe una lista de elementos y genera todas las posibles combinaciones de elementos en pares sin duplicados, devolviendo una lista de tuplas que representan estas combinaciones, esto nos servirá a futuro para hacer las comparaciones entre las deudas y ver si deben saldarse o no, y cómo debe hacerse.

gestionarMontosDeDeudas: Esta función recibe una tupla de dos deudas y devuelve una lista con una o cero deudas, donde el deudor y el acreedor se toman de la deuda con el monto mayor, y el monto de la deuda resultante es la diferencia entre los montos de las dos deudas dependiendo de cuales sea mayor (esto se hace para evitar valores negativos).

```haskell
-- Recibe una lista de deudas y las empareja. Ej: [deuda1,deuda2,deuda3] -> [(deuda1,deuda2),(deuda2,deu
emparejarDeudas :: (Eq a) => [a] -> [(a, a)]
emparejarDeudas [] = []
emparejarDeudas (x:xs) = emparejarDeuda x xs ++ emparejarDeudas xs
    where
        emparejarDeuda _ [] = []
        emparejarDeuda y (z:zs) = (y, z) : emparejarDeuda y zs

-- Recibe una tupla de deudas y devuelve 1 o 0 deudas, dependiendo de los montos
gestionarMontosDeDeudas :: (Deuda, Deuda) -> [Deuda]
gestionarMontosDeDeudas (deuda1, deuda2) =
    if montoDeuda deuda1 >= montoDeuda deuda2
        then [Deuda (deudor deuda1) (acreedor deuda1)  ((montoDeuda deuda1) - (montoDeuda deuda2))]
        else [Deuda (deudor deuda2) (acreedor deuda2)  ((montoDeuda deuda2) - (montoDeuda deuda1))]
```

Luego ingresando datos en el main:

```haskell
        -- Objetivo 3
        deudasActualizadas = actualizarDeudas deudasCompletas
        deudasDiferentes = eliminarDeudas deudasCompletas deudasActualizadas
        deudasModificadas = obtenerDeudasModificadas deudasDiferentes deudasActualizadas
        deudasSinModificar = eliminarDeudas deudasModificadas deudasActualizadas
        deudasNuevas = deudasSinModificar ++ deudasDiferentes
```

Se obtiene:

```
Deudas completas:
Juan le debe $30.0 a Pedro
Pedro le debe $20.0 a Juan
Santiago le debe $20.0 a Juan
Santiago le debe $30.0 a Pedro

Deudas actualizadas:
Juan le debe $10.0 a Pedro
Juan le debe $30.0 a Pedro
Pedro le debe $20.0 a Juan
Santiago le debe $20.0 a Juan
Santiago le debe $30.0 a Pedro

Deudas diferentes:
Juan le debe $10.0 a Pedro

Deudas modificadas:
Juan le debe $10.0 a Pedro
Juan le debe $30.0 a Pedro
Pedro le debe $20.0 a Juan
```

```
Deudas sin modificar:
Santiago le debe $20.0 a Juan
Santiago le debe $30.0 a Pedro

Deudas nuevas:
Juan le debe $10.0 a Pedro
Santiago le debe $20.0 a Juan
Santiago le debe $30.0 a Pedro
```

Objetivo 4

Implementar la función consultarBalance que recibe una persona, una lista de gastos y una lista de las personas que participan en el grupo y retorne el balance de esa persona dentro del grupo.

Se utiliza la siguiente función:
ConsultarBalance: Esta función calcula el balance de una persona dentro del grupo. Toma como entrada la persona para la cual se desea consultar el balance, una lista de gastos realizados por el grupo y una lista de todas las personas que participan en el grupo. Luego, calcula las deudas de la persona especificada basadas en los gastos, suma todas las deudas en las que la persona es acreedor y resta todas las deudas en las que la persona es deudor. Finalmente, devuelve el valor absoluto de la diferencia entre el monto total adeudado y el monto total recibido por la persona. Esto representa el balance total de la persona dentro del grupo.

```haskell
-- Objetivo 4
-- Función para calcular el balance de una persona dentro del grupo
consultarBalance :: Persona -> [Gasto] -> [Persona] -> Float
consultarBalance persona gastos amigos =
    let deudas = concatMap (crearDeudasDeGasto amigos) gastos
        deudasPersona = filter (\deuda -> deudor deuda == persona || acreedor deuda == persona) deudas
        montoTotalAcreedor = sum [montoDeuda deuda | deuda <- deudasPersona, acreedor deuda == persona]
        montoTotalDeudor = sum [montoDeuda deuda | deuda <- deudasPersona, deudor deuda == persona]
    in abs (montoTotalAcreedor - montoTotalDeudor)
```

Objetivo 4.1

¿Cómo se podría generalizar consultarBalance para poder listar el balance de todos los integrantes del grupo?
listarBalances: Toma una lista de gastos y una lista de personas que participan en el grupo, y devuelve una lista de tuplas donde cada tupla contiene una persona y su balance correspondiente dentro del grupo. Utiliza la función consultarBalance para calcular el balance de cada persona en la lista de amigos.

imprimirDeudas: Recibe una lista de deudas y muestra estas deudas por pantalla. Primero, convierte la lista de deudas en un conjunto para eliminar duplicados utilizando la función Set.fromList. Luego, convierte nuevamente el conjunto a una lista y llama a la función auxiliar imprimirDeudasSet para mostrar las deudas por pantalla.

```haskell
-- Objetivo 4.1
-- Función para listar el balance de todos los integrantes del grupo
listarBalances :: [Gasto] -> [Persona] -> [(Persona, Float)]
listarBalances gastos amigos = [(persona, consultarBalance persona gastos amigos) | persona <- amigos]


-- Recibe una lista de deudas y una lista de deudas sin duplicados
imprimirDeudas :: [Deuda] -> IO ()
imprimirDeudas deudas = do
    let deudasSet = Set.fromList deudas -- Convertir la lista de deudas a un conjunto para eliminar duplicados
    imprimirDeudasSet (Set.toList deudasSet) -- Convertir el conjunto nuevamente a una lista y llamar a la función aux
```

Resultado:

```
Balance de deudas:
[(Persona {nombre = "Juan"},10.0),(Persona {nombre = "Pedro"},40.0),(Persona {nombre = "Santiago"},50.0)]
```