Laboratorio N° 2

Materia: Paradigmas y Lenguajes de Programación

Profesores:

- Pecile, Lautaro
- Toledo, Pablo

Alumnos:

- Irigoyen, Carlos
- Morillo Meneses, Alen Misael

Universidad: Universidad Nacional de la Patagonia San Juan Bosco (UNPSJB)

Tema: Haskell

Carrera: Licenciatura en Sistemas

Año: 2023

Enunciado

Un amigo nuestro es encargado de un boliche, necesita modelar su negocio para mejorar la forma en que lo administra. En el relevamiento surgieron las siguientes entidades:

- Los clientes son personas mayores de edad. Al ingresar se registra su nombre, la resistencia, y los otros clientes a quienes considera amigos.
- Los clientes van teniendo nuevos amigos. Ningún cliente puede agregar a sí mismo como amigo y no tiene sentido volver a hacerse amigo de alguien a quien ya considera amigo.
- Cuando un cliente toma ciertas bebidas, suele verse afectado de forma diferente:
    - el “grog XD” deja sin resistencia al cliente.
    - la “jarra loca” baja 10 unidades de resistencia al cliente y a todos sus amigos.
    - el “Klusener” tiene diferentes gustos, como “Huevo”, “Chocolate” y “Frutilla”, entre otros, y disminuyen tanta resistencia como la cantidad de letras que tenga el gusto. Ej: Alguien toma Klusener de Huevo, eso le baja la resistencia 5 unidades.
    - el “Tintico” (café colombiano) le sube 5 puntos de resistencia al cliente por cada amigo que tenga.
    - la “Soda”, que dependiendo de la fuerza agrega “erp” al inicio del nombre del cliente, con tantas r como indique la fuerza. Por ejemplo, si Rodri se toma una soda con fuerza 2 y luego una soda con fuerza 5, su nombre queda “errrrrperrpRodri” (y queda con la misma resistencia y amigos).
- Pero como no es todo jarana, los clientes pueden decidir “rescatarse” un cierto tiempo medido en horas, que siempre es mayor a 0. Eso le devuelve 200 de resistencia si son más de 3 horas o 100 en caso contrario.

Primer parte

Objetivo 1

Modelar el tipo de dato cliente.

Para lograr este objetivo lo que hicimos fue modelar el cliente creando un nuevo tipo de dato llamado TipoCliente utilizando la sintaxis de registro. Este registro va a estar compuesto por el nombre del cliente, su resistencia y su lista de amigos. Luego definimos que el TipoCliente pueda ser comparable derivando de la clase de tipo Eq y que pueda ser mostrable a través de la clase Show, es decir representar al TipoCliente en pantalla como una cadena. Esto se puede ver a continuación:

```haskell
data TipoCliente = Cliente {
  nombreCliente :: String,
  resistencia :: Int,
  listaAmigos :: [TipoCliente]
} deriving (Show,Eq)
```

Luego, definimos el Show del TipoCliente donde el mismo mostrara el nombre del cliente, su resistencia y también el nombre y la resistencia de los amigos si es que los tuviese. Para poder mostrar el nombre y la resistencia de los amigos del cliente se creó la función listarAmigos la cual recibe un cliente y devuelve una cadena que muestra el nombre y resistencia de cada amigo del cliente.

```haskell
listarAmigos :: TipoCliente -> String
listarAmigos cliente =
  if null (listaAmigos cliente)
    then "No tiene amigos"
    else intercalate ", " (map mostrarAmigo (listaAmigos cliente))
  where
    mostrarAmigo amigo = nombreCliente amigo ++ " (Resistencia: " ++ show (resistencia amigo) ++ ")"

instance Show TipoCliente where
  show (Cliente nombre resistencia listaAmigos bebidas) =
    "Nombre: " ++ nombre
    ++ "\nResistencia: " ++ show resistencia
    ++ "\nAmigos: " ++ listarAmigos (Cliente nombre resistencia listaAmigos bebidas)
```

Objetivo 2

- Modelar a Rodri, que tiene 55 de resistencia y no considera a nadie como amigo.
- También modela a Marcos, un cliente que tiene resistencia 40 y considera a Rodri como su único amigo.
- Tenemos a Cristian, un cliente cuya resistencia es 2, y no considera a nadie como amigo.
- Por último, está Ana, una cliente que tiene 120 de resistencia y considera a Marcos y a Rodri como amigos.

Los clientes solicitados se pueden ver creados a continuación:

```
rodri = Cliente "Rodri" 55 []

marcos :: TipoCliente
marcos = Cliente "Marcos" 40 [rodri]

cristian :: TipoCliente
cristian = Cliente "Cristian" 2 []

ana :: TipoCliente
ana = Cliente "Ana" 120 [marcos, rodri]
```

Objetivo 3

Desarrollar la función comoEsta que dice como esta un cliente:

- Si su resistencia es mayor a 50, esta “fresco”.
- Si no está “fresco”, pero tiene mas de un amigo, esta “piola”.
- En caso contrario, esta “duro”.

Lo que hicimos fue definir una función que recibe un cliente y devuelva una cadena. La cadena muestra como está un cliente basándose en su resistencia y en su cantidad de amigos.

```
-- La Funcion Como Esta devuelve cuán ebrio está un cliente: fresco, piola o duro
comoEsta :: TipoCliente -> String
comoEsta cliente
  | resistencia cliente > 50 = "fresco"
  | resistencia cliente < 50 && length (listaAmigos cliente) > 1 = "piola"
  | otherwise = "duro"
```

Objetivo 4

Hacer que un cliente reconozca como amigo a otro cliente, respetando las restricciones definidas por el negocio (no agregar más de una vez al mismo amigo ni agregarse a sí mismo como amigo, basándose en que dos clientes son iguales si tienen el mismo nombre).

Para lograr este objetivo, se creó la función agregarAmigo que recibe dos clientes y devuelve un cliente con un nuevo amigo.

- Si el cliente y el amigo son la misma persona (nombre cliente == nombre amigo), la función devuelve el cliente sin realizar ningún cambio, ya que no se puede agregar a sí mismo como amigo.
- Si el amigo ya se encuentra en la lista de amigos del cliente (amigo elem listaAmigos cliente), la función devuelve el cliente sin realizar ningún cambio, ya que no se puede agregar al mismo amigo más de una vez.
- En cualquier otro caso, la función agrega al amigo a la lista de amigos del cliente y devuelve el cliente modificado con la nueva lista de amigos.

```haskell
-- Permite agregar a un cliente a la lista de amigos de otro cliente
agregarAmigo :: TipoCliente -> TipoCliente -> TipoCliente
agregarAmigo cliente amigo
  | cliente == amigo = cliente -- No se puede agregar a sí mismo como amigo
  | amigo `elem` listaAmigos cliente = cliente -- No se puede agregar más de una vez al mismo amigo
  | otherwise = cliente { listaAmigos = amigo : listaAmigos cliente }
```

Objetivo 5

Representar con la abstracción que crea conveniente

- a cada una de las bebidas mencionadas
- y cómo queda un cliente luego de tomar cualquiera de las bebidas mencionadas.

Para representar las bebidas, creamos un nuevo tipo de datos TipoBebida. Este nuevo tipo tendrá la siguiente estructura:

```haskell
-- Definimos el tipo de bebida
data TipoBebida = GrogXD | JarraLoca | Klusener { sabor :: String } | Tintico | Soda { fuerza :: Int }
```

Adicionalmente se definió el instance Show para especificar la representación de un TipoBebida en cadena de caracteres. El instance Show definido es el siguiente:

```haskell
-- Derivamos la instancia de Show para TipoBebida
instance Show TipoBebida where
  show GrogXD = "GrogXD"
  show JarraLoca = "JarraLoca"
  show (Klusener sabor) = "Klusener " ++ sabor
  show Tintico = "Tintico"
  show (Soda fuerza) = "Soda " ++ show fuerza
```

Para saber cómo queda un cliente después de tomar cualquier bebida, hicimos una función que recibe un cliente y una bebida, y devuelve un cliente afectado por la bebida. Para lograr esto, se tuvo en cuenta el efecto de cada bebida que se describe en el enunciado.

```haskell
-- Efecto de la bebida en el cliente
tomarBebida :: TipoBebida -> TipoCliente -> TipoCliente
tomarBebida GrogXD cliente = cliente { resistencia = 0, bebidas = GrogXD : bebidas cliente}
tomarBebida JarraLoca cliente = cliente { resistencia = resistencia cliente - 10, bebidas = JarraLoca : bebidas cliente, listaAmigos = map afectarAmigo (listaAmigos cliente)}
  where
    afectarAmigo amigo = amigo { resistencia = resistencia amigo - 10}
tomarBebida (Klusener gusto) cliente = cliente { resistencia = (resistencia cliente - length gusto), bebidas = (Klusener gusto) : bebidas cliente }
tomarBebida Tintico cliente = cliente { resistencia = resistencia cliente + (length $ listaAmigos cliente) * 5, bebidas = Tintico : bebidas cliente}
tomarBebida (Soda fuerza) cliente = cliente { nombreCliente = "e" ++ replicate fuerza 'r' ++ "p" ++ nombreCliente cliente, bebidas = (Soda fuerza) : bebidas cliente}


-- Función para rescatarse
```

Objetivo 6

Hacer que un cliente pueda rescatarse.

Para lograr este objetivo se creó la función rescatarse el cual recibe una cantidad de horas y un cliente y devuelve al cliente con la resistencia aumentada. La definición de la función se ve a continuación:

```
-- Función para rescatarse
rescatarse :: Int -> TipoCliente -> TipoCliente
rescatarse horas cliente
  | horas > 3 = cliente { resistencia = resistencia cliente + 200 }
  | otherwise = cliente { resistencia = resistencia cliente + 100 }
```

Objetivo 7

Escribir la consulta en la consola que permita realizar el siguiente itinerario con Ana: tomarse una jarra loca, un klusener de chocolate, rescatarse 2 horas y luego tomar un klusener de huevo.

Antes de ejecutar la acción solicitada en este punto verificamos cual es el estado de Ana y de sus amigos. De la consulta se obtuvo que:

- Ana tiene una resistencia de 120
- Marcos una resistencia de 40
- Rodri una resistencia de 55
- Ana no Tomó ninguna Bebida

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Para ejecutar la consulta se escribió en la consola**:**
((tomarBebida JarraLoca) . (tomarBebida (Klusener “Chocolate”)) . (rescatarse 2) . (tomarBebida (Klusener “huevo”))).
Luego de ejecutar la consulta se obtuvo lo siguiente:

- Ana tiene una resistencia de 196.
- Marcos tiene una resistencia de 30.
- Rodri tiene una resistencia de 45.
- Ana tomó las siguientes bebidas: JarraLoca,Klusener Chocolate,Klusener Huevo.

Esto se puede ver a continuación:

```
*Main> ((tomarBebida JarraLoca).(tomarBebida (Klusener "chocolate")) . (rescatarse 2) . (tomarBebida (Klusener "huevo"))) ana
Nombre: Ana
Resistencia: 196
Amigos: Marcos (Resistencia: 30), Rodri (Resistencia: 45)
Bebidas: JarraLoca, Klusener chocolate, Klusener huevo
*Main>
```

Casos de prueba

Objetivo 3

- Cristian debe estar “duro”
- Rodri debe estar “fresco”
- Marcos debe estar “duro”

Para los primeros tres incisos se hizo uso de la función comoEsta que dado un cliente devuelve una cadena que representa el estado de ebriedad del cliente. Dicha función tiene en cuenta la cantidad de amigos que tiene cliente como indica en el apartado ya que un cliente con una resistencia menor a 50 y con menos de dos amigos va a estar “duro” y esto se ve reflejado cuando se hace la consulta comoEsta con Marcos y con Cristian. Esto se puede ver a continuacion:

```
*Main> comoEsta cristian
"duro"
*Main> comoEsta rodri
"fresco"
*Main> comoEsta marcos
"duro"
*Main>
```

- Si Marcos se hace amigo de Ana y Rodri, está “piola”

Para lograr esta prueba, se hizo una composición de funciones entre la función comoEsta y agregarAmigos. Al hacerlo, se obtuvo lo siguiente:

```
*Main> (comoEsta . agregarAmigo (agregarAmigo marcos rodri) $ ana)
"piola"
```

Objetivo 4

- Intentar agregar a Rodri como amigo de Rodri.

Al intentar agregar a Rodri como amigo de sí mismo, la función agregarAmigo no lo permitió debido a que está diseñada de tal manera que no permita que un cliente se agregue a él mismo como amigo. En su lugar, devolverá al cliente sin ningún cambio. Esto se puede ver a continuación:

```
*Main> (agregarAmigo rodri rodri)
Nombre: Rodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

- Intentar que Marcos reconozca a Rodri como amigo (que ya lo conoce).

Al intentar agregar a Rodri como amigo de Marcos no sucede ningún cambio debido a que la función agregarAmigo está diseñada para que no se agregue de vuelta a alguien que ya es amigo.

```
*Main> (agregarAmigo marcos rodri)
Nombre: Marcos
Resistencia: 40
Amigos: Rodri (Resistencia: 55)
Bebidas: Klusener guinda
*Main>
```

- Hacer que Rodri reconozca a Marcos como amigo. Debe arrancar con 0 amigos y luego agregarlo a Marcos como su único amigo.

Como paso previo se procede a consultar el estado de Rodri antes de reconocer a Marcos como amigo. Para esto escribimos en consola “rodri” y se obtiene lo siguiente:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

Luego se procedió a hacer que Rodri reconozca a marcos como amigo. Para ello se ejecuta la función agregarAmigo pasando a Rodri y a Marcos como el amigo a agregar y se obtiene lo siguiente:

```
*Main> (agregarAmigo rodri marcos)
Nombre: Rodri
Resistencia: 55
Amigos: Marcos (Resistencia: 40)
Bebidas: Tintico
*Main>
```

Objetivo 5

- Ana toma GrogXD. Queda con resistencia 0.

Como paso previo se procede a consultar el estado de Ana antes de tomar un GrogXD. Para ello se escribe por consola “ana” y se obtiene lo siguiente:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego se procedió a ejecutar la función tomarBebida con el GrogXD y el cliente Ana y Ana queda con resistencia 0 y con GrogXD como su bebida tomada. Esto se puede observar a continuación:

```
*Main> (tomarBebida GrogXD ana)
Nombre: Ana
Resistencia: 0
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: GrogXD
*Main>
```

- Ana toma la Jarra Loca. Queda con resistencia 110, su amigo Marcos queda con 30 de resistencia y su amigo Rodri queda con 45 de resistencia.

Como paso previo se procede a consultar el estado de Ana antes de tomar una Jarra Loca:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego se procedió a ejecutar la función tomarBebida con la Jarra Loca y el cliente Ana , Ana queda con resistencia 110 con la Jarra Loca como su bebida tomada y a sus amigos se le baja en 10 la resistencia:

```
*Main> (tomarBebida JarraLoca ana)
Nombre: Ana
Resistencia: 110
Amigos: Marcos (Resistencia: 30), Rodri (Resistencia: 45)
Bebidas: JarraLoca
*Main>
```

- Ana toma un Klusener de huevo, queda con 115 de resistencia

Como paso previo se procede a consultar el estado de Ana antes de tomar un Klusener de huevo:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego se procedió a ejecutar la función tomarBebida con el Klusener de huevo. El resultado es que Ana queda con resistencia 115 y con el Klusener de huevo como su bebida tomada.

```
ghci> tomarBebida (Klusener "huevo") ana
Nombre: Ana
Resistencia: 115
Amigos:Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: Klusener huevo
```

- Ana toma un Klusener de chocolate, queda con 111 de resistencia

Como paso previo se consulta el estado de Ana antes de tomar el Klusener de chocolate y el mismo es el siguiente:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego de ejecutar la función tomarBebida con el Klusener de chocolate y pasando también a Ana como parámetro se obtiene que Ana queda con 111 de resistencia.

```
*Main>  (tomarBebida (Klusener "chocolate")ana)
Nombre: Ana
Resistencia: 111
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: Klusener chocolate
*Main>
```

- Cristian toma un Tintico, queda con 2 de resistencia por no tener amigos.

Antes de hacer que Cristian tome el Tintico procedemos a consultar su estado y el mismo es el siguiente:

```
*Main> cristian
Nombre: Cristian
Resistencia: 2
Amigos: No tiene amigos
Bebidas: GrogXD, JarraLoca
*Main>
```

A continuación, ejecutamos la funcion tomarBebida y le pasamos el Tintico y a Cristian y nos queda como resultado lo siguiente:

```
*Main> (tomarBebida Tintico cristian)
Nombre: Cristian
Resistencia: 2
Amigos: No tiene amigos
Bebidas: Tintico, GrogXD, JarraLoca
*Main>
```

Donde, como se puede ver, Cristian queda con 2 de resistencia.

- Ana toma un Tintico, pasa de 120 a 130 de resistencia (tiene 2 amigos)

Antes de hacer que Ana se tome el Tintico debemos consultar su estado actual y el mismo es el siguiente:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

A continuación, ejecutamos la función tomarBebida pasando al Tintico y a Ana como parámetro y se obtiene lo siguiente:

```
*Main> (tomarBebida Tintico ana)
Nombre: Ana
Resistencia: 130
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: Tintico
*Main>
```

Donde, Ana queda con 130 de resistencia al tener 2 amigos.

- Rodri toma una Soda de fuerza 2, queda con nombre “errpRodri”

Previo a tomar la Soda de fuerza 2, el nombre de Rodri era el siguiente:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

Luego de ejecutar la función tomarBebida pasando la Soda de fuerza 2 y a Rodri, su nombre quedó de la siguiente manera:

```
*Main> (tomarBebida(Soda 2) rodri)
Nombre: errpRodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Soda 2, Tintico
*Main>
```

- Ana toma una Soda de fuerza 10, queda con nombre “errrrrrrrrrpAna”

Antes de tomar la Soda de fuerza 10, Ana tiene el siguiente nombre:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego de tomar la Soda de fuerza 10, Ana pasa a tener el siguiente nombre:

```
*Main> (tomarBebida(Soda 10) ana)
Nombre: errrrrrrrrrpAna
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: Soda 10
*Main>
```

- Ana toma una Soda de fuerza 0, queda con nombre “epAna”

Antes de tomar la soda de fuerza 0, Ana tiene el siguiente nombre:

```
*Main> ana
Nombre: Ana
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
*Main>
```

Luego de tomar la soda de fuerza 0, Ana queda con el siguiente nombre:

```
*Main> (tomarBebida(Soda 0) ana)
Nombre: epAna
Resistencia: 120
Amigos: Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: Soda 0
*Main>
```

Objetivo 6

- Rodri se rescata 5 horas, queda con 255 de resistencia (55 + 200 ya que son más de 3 horas)

El estado de Rodri previo a rescatarse es el siguiente:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

Para poder llevar a cabo la prueba, hay que ejecutar la función rescatarse pasando como parámetro 5 (5 horas) y a Rodri. Luego de rescatarse 5 horas, el estado de Rodri pasó a ser el siguiente:

```
*Main> (rescatarse 5 rodri)
Nombre: Rodri
Resistencia: 255
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

- Rodri se rescata 1 hora, queda con 155 de resistencia (55 + 100 porque son menos de 3 horas)

Previo a rescatarse, Rodri tiene el siguiente estado:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

Luego de rescatarse por 1 hora, el estado de Rodri cambio al siguiente:

```
*Main> (rescatarse 1 rodri)
Nombre: Rodri
Resistencia: 155
Amigos: No tiene amigos
Bebidas: Tintico
*Main>
```

Objetivo 7

Luego de evaluar el itinerario de Ana, queda con 196 de resistencia, teniendo como amigos a Marcos (30 de resistencia) y Rodri (45 de resistencia).

Para hacer que Ana realice su itinerario se debe escribir en consola lo siguiente: **(tomarBebida (Klusener "huevo") . (rescatarse 2) . (tomarBebida (Klusener "Chocolate")) . (tomarBebida (JarraLoca))) ana**

Lo cual dio como resultado lo siguiente:

```
ghci> (tomarBebida (Klusener "huevo") . (rescatarse 2) . (tomarBebida (Klusener "Chocolate")) . (tomarBebida (JarraLoca))) ana
Nombre: Ana
Resistencia: 196
Amigos:Marcos (Resistencia: 30), Rodri (Resistencia: 45)
Bebidas: Klusener huevo, Klusener Chocolate, JarraLoca
```

Segunda parte

Objetivo 1

De los clientes, además del nombre, resistencia y sus amigos, se desea saber qué bebidas tomó.

- Hacer las modificaciones en la abstracción cliente y considerar

  - ➢ Que Rodri tomó un tintico.
  - ➢ Que Marcos tomó un Klusener de guinda.
  - ➢ Que Ana no tomó nada.
  - ➢ Que Cristian tomó un grog XD y una jarra Loca.

Nota 1: Para mantener la compatibilidad con lo desarrollado anteriormente, recordar que se considera que dos clientes son iguales si tienen el mismo nombre. En otras palabras, al cambiar el tipo cliente, revisar que la entrega 1 siga funcionando de acuerdo al enunciado.

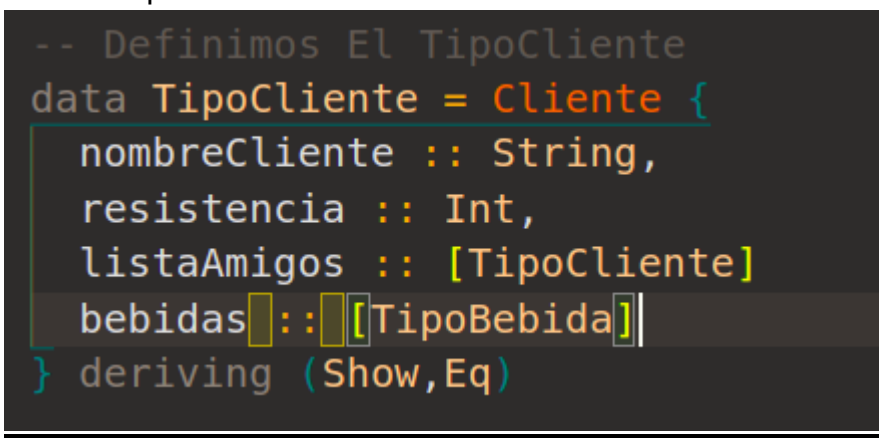
Nota 2: Si aparece un error “no instance for Show” leer acá: http://wiki.uqbar.org/wiki/articles/no-hay-instancias-para-el-show.html

- Hacer que un cliente pueda tomar una bebida. Además del efecto que le causa la bebida en sí, se debe registrar esa bebida en su historial de bebidas tomadas.

- Desarrollar la función tomarTragos, la cual recibe a un cliente y una lista de tragos y retorna al cliente luego de tomarlos todos los tragos.

- Hacer la función dameOtro, que hace que un cliente vuelva a tomarse el último trago que se tomó.

Para realizar este objetivo, lo que hicimos fue agregar una lista de bebidas en el tipo de dato TipoCliente. Esto se ve a continuación:



Luego, agregamos bebidas que tomaron los clientes que tenemos definidos (Rodri, Marcos, Ana, Cristian).

```haskell
-- Definimos los clientes que se solicitaron
rodri :: TipoCliente
rodri = Cliente "Rodri" 55 [] [Tintico]  -- Rodri tomó un tintico

marcos :: TipoCliente
marcos = Cliente "Marcos" 40 [rodri] [Klusener "guinda"]  -- Marcos tomó un Klusener de guinda

cristian :: TipoCliente
cristian = Cliente "Cristian" 2 [] [GrogXD, JarraLoca]  -- Cristian tomó un grog XD y una jarraLoca

ana :: TipoCliente
ana = Cliente "Ana" 120 [marcos, rodri] []  -- Ana no tomó nada
```

Para hacer que un cliente pueda tomar una lista de bebidas, definimos una función recursiva llamada tomarTragos que recibe un cliente y una lista de tragos y devuelve un cliente con esa bebida tomada.

En el caso base devuelve al cliente y en el caso recursivo, se toma el primer trago de la lista y se aplica su efecto al cliente utilizando la función tomarBebida. Luego, se llama recursivamente a tomarTragos pasando el cliente afectado y la lista de tragos restantes. La función entonces, es la siguiente:

```haskell
--Dada una lista de bebidas y un cliente devuelve al cliente luego de haaber tomado las bebidas
tomarTragos :: TipoCliente -> [TipoBebida] -> TipoCliente
tomarTragos cliente [] = cliente -- Caso base: no hay más tragos para tomar
tomarTragos cliente (trago:restoTragos) = tomarTragos (tomarBebida trago cliente) restoTragos -- Hay más tragos por tomar
```

Para desarrollar la función dameOtro hicimos uso de guardas para que dado un cliente devuelva al cliente con la última bebida(con el efecto que esta provoca) que pidió. Para ello hicimos uso de la función head que nos devuelve el primer elemento de una lista (tomando como base que el primer elemento de la lista es el último trago tomado).

En el caso de que el cliente no tenga ninguna bebida en su historial directamente se devuelve al cliente sin ningún cambio. Esto se puede ver a continuación:

```
dameOtro :: TipoCliente -> TipoCliente
dameOtro cliente
  | length ( bebidas cliente) == 0 = cliente
  | otherwise = tomarBebida (head(bebidas cliente)) cliente
```

Objetivo 2

a. Definir la función cualesPuedeTomar, la cual recibe a un cliente y una lista de tragos y nos dice cuáles de esas bebidas lo dejarían con una resistencia mayor a cero, en caso de tomarlas solas.

La función cualesPuedeTomar fue definida de la siguiente manera:

```
cualesPuedeTomar :: TipoCliente -> [TipoBebida] -> [TipoBebida]
cualesPuedeTomar _ [] = []
cualesPuedeTomar cliente (bebida:restoBebidas)
  | resistencia (tomarBebida bebida cliente) > 0 = bebida : cualesPuedeTomar cliente restoBebidas
  | otherwise = cualesPuedeTomar cliente restoBebidas
```

Esta función va a recibir un cliente y una lista de bebidas y va a devolver la lista de bebidas que el cliente va a poder tomar sin que su resistencia llegue a 0.

La función va a tener un caso base que es cuando no hay una lista de bebidas y por lo tanto se devuelve la lista vacía y un caso recursivo. En el caso recursivo se ira comprobando como queda la resistencia de un cliente luego de tomar la bebida y se verifica que la misma sea mayor a 0.

b. Definir la función cuantasPuedeTomar, que devuelva la cantidad de bebidas en base al punto anterior.

La funcion cuantasPuedeTomar va a quedar definida de la siguiente manera:

```
cuantasPuedeTomar :: TipoCliente -> [TipoBebida] -> Int
cuantasPuedeTomar cliente bebidas = length (cualesPuedeTomar cliente bebidas)
```

Esta función hace uso de la función del item anterior ya que recibe un cliente y una lista de bebidas y devuelve un entero que representa la cantidad de bebidas que el cliente puede tomar que lo dejan con una resistencia mayor a 0. Para eso hace uso de la función cualesPuedeTomar de la cual obtiene la lista de bebidas que cumplan con la condición y luego devuelve la longitud de la lista.

Objetivo 3

Ahora, aparecen diferentes itinerarios que pueden realizar los clientes, de los cuales se registran su nombre, una duración estimada y lo más importante, el detalle de las acciones que componen el itinerario.

Para esta parte definimos el tipo de dato TipoItinerario. Este tipo de dato va a estar compuesto por un nombre, la duración del itinerario y un detalle el cual será una lista que irá de TipoCliente a TipoClliente y tendrá en su interior funciones que recibirán un cliente y lo devolverán modificado. La definición del tipo se puede ver a continuación:

```
data TipoItinerario = Itinerario {
  nombreItinerario :: String,
  duracion :: Float,
  detalle :: [TipoCliente -> TipoCliente]
}

instance Show TipoItinerario where
    show i = nombreItinerario i
```

Algunos itinerarios son los siguientes:

- Mezcla explosiva, se recomienda para 2.5 horas y consiste en tomarse 2 Grog XD, 1 Klusener de Huevo y otro de Frutilla.
- Itinerario básico, es como el del punto 7 de la primera parte, en 5 horas.
- Salida de amigos, se recomienda para 1 hora, consiste en tomarse una soda, de nivel 1, un tintico, hacerse amigo de Roberto Carlos (sí, es cliente de este boliche) y tomarse una jarra loca.

a. Modelar los itinerarios existentes, para lo cual considerar que:
   - Roberto Carlos no tiene amigos, tiene 165 de resistencia y no tomó nada.
   - Deben utilizar como expresiones robertoCarlos, mezclaExplosiva, itinerarioBasico, salidaDeAmigos.

Se modelaron los itinerarios solicitados de la siguiente forma:

```
--Definicion de los itinerarios
mezclaExplosiva:: TipoItinerario
mezclaExplosiva  = Itinerario "Mezcla explosiva" 2.5 [tomarBebida (GrogXD) ,tomarBebida (GrogXD) ,tomarBebida (Klusener "huevo") , tomarBebida (Klusener "frutilla") ]


itinerarioBasico:: TipoItinerario
itinerarioBasico = Itinerario "Itinerario basico" 5.0 [ tomarBebida (JarraLoca), tomarBebida (Klusener "Chocolate"), rescatarse (2), tomarBebida (Klusener "huevo")]

salidaDeAmigos:: TipoItinerario
salidaDeAmigos = Itinerario "Salida de amigos" 1.0 [tomarBebida (Soda 1), tomarBebida (Tintico), agregarARC (robertoCarlos), tomarBebida (JarraLoca)]
```

Donde Roberto Carlos tendrá la siguiente forma:

```
robertoCarlos :: TipoCliente
robertoCarlos = Cliente "Roberto Carlos" 165 [] []
```

Y agregarARC va a ser una función que realizara lo mismo que agregarAmigo pero con los roles invertidos. Es decir, donde estaba cliente ira amigo y donde estaba amigo ira cliente. Entonces, agregarARC queda definido de la siguiente manera:

```
--Permite agregar a Roberto Carlos como amigo
agregarARC :: TipoCliente -> TipoCliente -> TipoCliente
agregarARC amigo cliente
  | nombreCliente cliente == nombreCliente amigo = cliente -- No se puede agregar a sí mismo como amigo
  | amigo `elem` listaAmigos cliente = cliente -- No se puede agregar más de una vez al mismo amigo
  | otherwise = cliente { listaAmigos = amigo : listaAmigos cliente }
```

b. Mostrar cómo Rodri hace una salida de amigos y Marcos una mezcla explosiva.

Antes de hacer la salida de amigos, revisamos el estado de Rodri y el mismo es el siguiente:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Tintico
```

Vemos que Rodri inicialmente no tiene amigos y la única bebida que tomo es el Tintico. Ahora bien, el itinerario Salida de Amigos es el siguiente:

```
salidaDeAmigos:: TipoItinerario
salidaDeAmigos = Itinerario "Salida de amigos" 1.0 [tomarBebida (Soda 1), tomarBebida (Tintico), agregarARC (robertoCarlos), tomarBebida (JarraLoca)]
```

Teniendo en cuenta el itinerario Salida de Amigos, Rodri debería terminar con Roberto Carlos como amigo, con JarraLoca, Tintico, Soda 1 y Tintico en su historial de bebidas tomadas, con la misma resistencia y con el nombre “erpRodri” por el efecto de la soda.

Para poder comprobar si esto es verdad, realizamos la consulta en la consola y obtenemos lo siguiente:

```
*Main> hacerItinerario (detalle salidaDeAmigos) rodri
Nombre: erpRodri
Resistencia: 45
Amigos:Roberto Carlos (Resistencia: 155)
Bebidas: JarraLoca, Tintico, Soda 1, Tintico
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Tintico
*Main>
```

Donde, en efecto, Rodri cambió su estado a lo que se había pensado anteriormente.

Objetivo 4

a. Conocer la intensidad de un itinerario, que se calcula como la cantidad de acciones que realiza por hora.

Tip: si tenés problemas de tipos, podés intentar usar la función genericLength en lugar de la función length. Para eso tenés que escribir import Data.List arriba de todo en tu archivo.hs.

Para conocer la intensidad de un itinerario se realizó la siguiente función:

```
--Objetivo 4a
intensidadItinerario:: TipoItinerario -> Float
intensidadItinerario itinerario = (genericLength (detalle itinerario)) / duracion itinerario
```

La cual recibe un itinerario y devuelve un flotante que es resultado de dividir la longitud de la lista de detalles del itinerario por su duración en horas. De esta manera se obtiene cuantas acciones se realizan en 1 hora. Se uso el genericLength para no tener conflictos entre los tipos Int (longitud del detalle Itinerario) y Float (duración).

b. Hacer que un cliente realice el itinerario más intenso, entre un conjunto de itinerarios dado.

Para que un cliente realice el itinerario más intenso de una lista de itinerarios se usaron 2 funciones. La primera es la siguiente:

```
hacerItinerarioIntenso:: [TipoItinerario] -> TipoCliente -> TipoCliente
hacerItinerarioIntenso [] cliente = cliente
hacerItinerarioIntenso [itinerario] cliente = hacerItinerario (detalle itinerario) cliente
hacerItinerarioIntenso itinerarios cliente = (hacerItinerario(detalle (obtenerItinerarioMasIntenso itinerarios)) cliente)
```

Esta función recibirá una lista de itinerarios y si está vacía, devolverá al cliente normal. En caso de que tenga 1 solo elemento realizará el itinerario de ese elemento y en caso de tener varios elementos realizará una llamada a la función hacerItinerario que recibe como parámetros el detalle del itinerario más intenso que se obtenga de obtenerItinerarioIntenso y además recibirá un cliente. La función obtenerItinerarioIntenso es la siguiente:

```
--Objetivo 4b
obtenerItinerarioMasIntenso:: [TipoItinerario] -> TipoItinerario
obtenerItinerarioMasIntenso[itinerario] = itinerario
obtenerItinerarioMasIntenso(primero:resto) = mayorItinerario primero resto
                    where
                    mayorItinerario itinerario1 [] = itinerario1
                    mayorItinerario itinerario1 (itinerario2:restoAux)
                            |   (intensidadItinerario itinerario1) <= (intensidadItinerario itinerario2) = mayorItinerario itinerario2
                            |   otherwise = mayorItinerario itinerario1 restoAux
```

Esta función recibirá una lista de itinerarios e ira recorriéndola mientras compara la intensidad de cada itinerario y se queda con el itinerario que tiene la intensidad más alta.

Usando estas 2 funciones es como se logra obtener el itinerario más intenso.

Objetivo 5

¡Se agrega al boliche la jarra popular! Cuando un cliente toma de la jarra popular, se vuelve amigo de los amigos de sus amigos, y de los amigos de los amigos de sus amigos, etc. La cantidad máxima de indirecciones está determinada por el nivel de

espirituosidad de la jarra popular. Debe tenerse en cuenta que, como ya se explicó anteriormente, no se agregue amigos que ya tiene, ni a sí mismo.

Para lograr este objetivo se realizo lo siguiente:

En primer lugar, lo que se hizo fue agregar el tipo de bebida JarraPopular con su correspondiente Show:

```haskell
-- Definimos el tipo de bebida
data TipoBebida = GrogXD | JarraLoca | Klusener { sabor :: String } | Tintico | Soda { fuerza :: Int } | JarraPopular { espirituosidad :: Int}  deriving Eq



-- Derivamos la instancia de Show para TipoBebida
instance Show TipoBebida where
  show GrogXD = "GrogXD"
  show JarraLoca = "JarraLoca"
  show (Klusener sabor) = "Klusener " ++ sabor
  show Tintico = "Tintico"
  show (Soda fuerza) = "Soda De Fuerza:" ++ show fuerza
  show (JarraPopular espirituosidad) = "Jarra popular de espirituosidad: " ++ show espirituosidad
```

Luego en tomarBebida se agregó el efecto que produce la JarraPopular la cual es hacer amigos de los amigos respetando las restricciones ya impuestas. Esta función quedo definida como lo siguiente:

```haskell
tomarBebida (JarraPopular espirituosidad) cliente = agregarAmigosCascada espirituosidad (listaAmigos cliente) cliente { bebidas = JarraPopular espirituosidad : bebidas cliente }
  where
    agregarAmigosCascada 0 _ cliente = cliente
    agregarAmigosCascada _ [] cliente = cliente
    agregarAmigosCascada n (amigo:restoAmigos) cliente = agregarAmigosCascada (n - 1) (listaAmigos amigo ++ restoAmigos) (agregarAmigo cliente amigo)
```

Luego, hacemos uso de una función auxiliar la cual es agregarAmigosCascada la cual trabaja de la siguiente forma:

- Si el nivel de espirituosidad es igual a cero o la lista de amigos está vacía, se devuelve el cliente sin cambios.
- Si el nivel de espiritualidad no es cero y la lista de amigos no está vacía, se toma al primer amigo de la lista (amigo) y se agrega como amigo del cliente utilizando la función agregarAmigo. Luego, se llama recursivamente a la función agregarAmigosCascada con el nivel de espirituosidad decrementado en uno. Esto se va a ir haciendo hasta llegar al caso base.

Casos de prueba

Objetivo 1b

- Marcos toma una soda de nivel 3 y queda con 2 bebidas
- Marcos toma una soda de nivel 3 y queda con 40 de resistencia

Para realizar estos objetivos se escribe lo siguiente en consola:
tomarBebida (Soda 3) marcos

Lo cual da como resultado:

```
Nombre: errrpMarcos
Resistencia: 40
Amigos:Rodri (Resistencia: 55)
Bebidas: Soda 3, Klusener guinda
```

Donde cómo se puede ver, Marcos quedó con resistencia 40 y con 2 bebidas en su historial.

Objetivo 1c

- Rodri toma una soda de nivel 1 y una soda de nivel 2 y queda con nombre “errperpRodri”.

Para realizar este objetivo se realiza la siguiente consulta:
(tomarBebida (Soda 2) . tomarBebida (Soda 1)) rodri
Lo cual nos queda como resultado:

```
*Main> (tomarBebida (Soda 2) . tomarBebida (Soda 1)) rodri
Nombre: errperpRodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Soda 2, Soda 1, Tintico
*Main>
```

Se puede observar que las sodas quedan registradas en el historial de bebidas de Rodri y que el nombre cambia producto del efecto de las dos sodas.

- Marcos toma un klusener de huevo, un tintico y una jarraLoca y queda con 30 de resistencia.

Para realizar este objetivo se escribió en consola lo siguiente:
(tomarBebida JarraLoca . tomarBebida Tintico. tomarBebida (Klusener "huevo"))marcos
Y nos da como resultado:

```
*Main> (tomarBebida JarraLoca . tomarBebida Tintico . tomarBebida (Klusener "huevo"))marcos
Nombre: Marcos
Resistencia: 30
Amigos:Rodri (Resistencia: 45)
Bebidas: JarraLoca, Tintico, Klusener huevo, Klusener guinda
*Main>
```

Se puede observar que Marcos queda en su historial de bebidas con las bebidas usadas en la consulta más el Klusener de guinda que ya lo tenía desde el principio y, ademas, la resistencia de Rodri que al inicio estaba en 55 se ve afectada por la Jarra Loca.

- Marcos toma un klusener de huevo, un tintico y una jarraLoca y queda con 4 bebidas en el historial.

Para hacer que Marcos se tome un klusener de huevo, un tintico y una jarraLoca y quede con 30 de resistencia y 4 bebidas en el historial se debe escribir lo siguiente en consola:
(tomarBebida (Klusener “huevo”) . tomarBebida (Tintico) . tomarBebida (JarraLoca)) marcos

Lo cual nos dará como resultado:

```
Nombre: Marcos
Resistencia: 30
Amigos:Rodri (Resistencia: 45)
Bebidas: Klusener huevo, Tintico, JarraLoca, Klusener guinda
```

Donde, como se puede observar, queda con 30 de resistencia y 4 bebidas en su historial.

Objetivo 1d

- Ana pide “dame otro” y debe dar error.

Para hacer que Ana pida otro se escribió en consola lo siguiente: dameOtro ana y el resultado fue el siguiente:

```
ghci> dameOtro ana
Nombre: Ana
Resistencia: 120
Amigos:Marcos (Resistencia: 40), Rodri (Resistencia: 55)
Bebidas: No ha tomado ninguna bebida
```

No devuelve ningún error porque decidimos que no hacía falta que termine con un error, sino que debería devolver al cliente sin ningún cambio.

- Marcos pide “dame otro” y tiene 2 bebidas en el historial.
- Marcos pide “dame otro” y lo deja con 34 de resistencia.

Para hacer los 2 objetivos que implican a Marcos se escribió en consola: dameOtro marcos y el resultado fue el siguiente:

```
ghci> dameOtro marcos
Nombre: Marcos
Resistencia: 34
Amigos:Rodri (Resistencia: 55)
Bebidas: Klusener guinda, Klusener guinda
```

Donde, como se puede ver, Marcos queda con una resistencia de 34 y con 2 bebidas en su historial.

- Rodri toma una soda de nivel 1, y “dameOtro” da como resultado que tiene 3 bebidas.
- Rodri toma una soda de nivel 1, y “dameOtro” da como resultado que su nombre queda “erperpRodri”.

Para que Rodri tome una soda de nivel 1 y a continuacion haga un dameOtro se escribió en consola lo siguiente: (dameOtro . (tomarBebida (Soda 1))) rodri y se obtuvo lo siguiente:

```
ghci> (dameOtro . (tomarBebida (Soda 1))) rodri
Nombre: erperpRodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Soda 1, Soda 1, Tintico
```

Donde, Rodri queda con el nombre “erperpRodri” y con 3 bebidas en su historial.

Objetivo 2b

- Rodri puede tomar dos bebidas, entre un grog XD, un tintico y un klusener de frutilla

Al consultar el estado de Rodri antes de ejecutar la consulta tenemos lo siguiente:

```
2
*Main>  rodri
Nombre: Rodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Tintico
*Main>
```

Por lo cual al ejecutar la consulta de cuantasPuedeTomar debería dar como resultado 2 ya que el GrogXD lo deja con una resistencia igual a 0. Entonces, la consulta realizada es la siguiente:
cuantasPuedeTomar rodri [GrogXD,Tintico,Klusener"Frutilla"] y el resultado de la misma nos da:

```
*Main> cuantasPuedeTomar rodri [GrogXD,Tintico,Klusener"Frutilla"]
2
*Main>
```

- Entre un grog XD, un tintico, un klusener de “fruuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuutilla” se puede tomar una sola bebida

Para poder comprobar cuantas bebidas se pueden tomar entre el GrogXD, el Tintico y un Klusener de “fruuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuutilla” se tomará a Rodri para que sea el cliente de prueba:

Antes de saber cuántas bebidas puede tomar Rodri, queremos saber su estado actual. El mismo es el siguiente:

```
Nombre: Rodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Tintico
```

Luego, para saber cuántas bebidas se pueden tomar se debe ejecutar cuantosPuedeTomar junto con rodri y la lista de bebidas. El resultado es el siguiente:

```
ghci> cuantasPuedeTomar rodri [GrogXD, Tintico, (Klusener "fruuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuutilla")]
1
ghci>
```

Es lógico ya que el GrogXD lo deja en 0 de resistencia y el Klusener decrementa la resistencia según la cantidad de letras que tenga el sabor.

Objetivo 3b

- Rodri hace una salida de amigos y debe quedar con un amigo
- Rodri hace una salida de amigos y se debe llamar “erpRodri”
- Rodri hace una salida de amigos y debe quedar con 45 de resistencia
- Rodri hace una salida de amigos y su primer y único amigo Roberto Carlos debe quedar con 155 de resistencia
- Rodri hace una salida de amigos y debe quedar con 4 bebidas en su historial.

Para que Rodri haga una salida de amigos se escribió en consola lo siguiente:
hacerItinerario (detalle salidaDeAmigos) rodri
Y como resultado se obtuvo:

```
ghci> hacerItinerario (detalle salidaDeAmigos) rodri
Nombre: erpRodri
Resistencia: 45
Amigos:Roberto Carlos (Resistencia: 155)
Bebidas: JarraLoca, Tintico, Soda 1, Tintico
```

Donde Rodri quedo con 1 solo amigo que es Roberto Carlos quien quedó con 155 de resistencia, quedó también con 45 de resistencia, con el nombre erpRodri y con 4 bebidas en su historial.

Objetivo 4a

- La intensidad de la mezcla explosiva es 1.6
- La intensidad de la salidaDeAmigos es 4.0
- La intensidad del itinerario basico es 0.8

Para calcular la intensidad de la mezclaExplosiva se sabe que mezclaExplosiva tiene lo siguiente:

```
mezclaExplosiva:: TipoItinerario
mezclaExplosiva  = Itinerario "Mezcla explosiva" 2.5 [tomarBebida
(GrogXD) ,tomarBebida (GrogXD) ,tomarBebida (Klusener "huevo") ,
tomarBebida (Klusener "frutilla") ]
```

Y la función que calcula la intensidad está dada como:

```
intensidadItinerario:: TipoItinerario -> Float
intensidadItinerario itinerario = (genericLength (detalle itinerario)) / duracion itinerario
```

Por lo cual, al calcular la intensidad del itinerario debería ser igual a (4 / 2.5) = 1.6. Entonces, para poder comprobar la intensidad del itinerario se debe escribir en consola: intensidadItinerario mezclaExplosiva. Una vez escrito eso el resultado es el siguiente:

```
ghci> intensidadItinerario mezclaExplosiva
1.6
```

Lo cual es correcto ya que la cantidad de acciones que se realizan es igual a 4 y la duración es de 2.5 horas.

A continuación, para poder calcular la intensidad del itinerario salidaDeAmigos debemos tener en cuenta que el mismo está definido de la siguiente manera:

```
salidaDeAmigos:: TipoItinerario
salidaDeAmigos = Itinerario "Salida de amigos" 1.0 [tomarBebida (Soda
1), tomarBebida (Tintico), agregarARC (robertoCarlos), tomarBebida
(JarraLoca)]
```

Por lo que, teniendo en cuenta la función de la intensidad sabemos que la cantidad de acciones es igual a 4 y la duración es de 1.0. Si realizamos la división nos va a dar que su intensidad es de 4 horas. Entonces, para poder ver la intensidad de salidaDeAmigos se debe escribir lo siguiente en consola: intensidadItinerario salidaDeAmigos.

Luego de escribirlo, se obtiene lo siguiente:

```
ghci> intensidadItinerario salidaDeAmigos
4.0
```

Lo cual es correcto.

Por último, para poder calcular la intensidad del itinerario básico tenemos que su definición es la siguiente:

```
itinerarioBasico:: TipoItinerario
itinerarioBasico = Itinerario "Itinerario basico" 5.0 [ tomarBebida
(JarraLoca), tomarBebida (Klusener "Chocolate"), rescatarse (2),
tomarBebida (Klusener "huevo")]
```

Donde la cantidad de acciones que se realizan es de 4 y su duración es de 5 horas. Por lo tanto, su intensidad es (4 / 5) = 0.8 .
Para poder comprobar la intensidad del itinerario escribimos por consola: intensidadItinerario itinerarioBasico. A continuación, obtenemos lo siguiente:

```
ghci> intensidadItinerario itinerarioBasico
0.8
```

Objetivo 4b

- Entre la salida de amigos, la mezcla explosiva y el itinerario básico, el itinerario más intenso es la salida de amigos.

- Rodri hace el itinerario más intenso entre una salida de amigos, la mezcla explosiva y el itinerario básico y queda con el nombre “erpRodri”

- Rodri hace el itinerario más intenso entre una salida de amigos, la mezcla explosiva y el itinerario básico y queda con resistencia 45.

- Rodri hace el itinerario más intenso entre una salida de amigos, la mezcla explosiva y el itinerario básico y queda con un amigo: Roberto Carlos

Primero vemos el estado de Rodri antes de cualquier itinerario. El estado de Rodri es el siguiente:

```
*Main> rodri
Nombre: Rodri
Resistencia: 55
Amigos:No tiene amigos
Bebidas: Tintico
```

Vemos que no tiene amigos, que solo bebió un Tintico y que su resistencia es de 55. Para poder hacer los apartados solicitados en este punto hay que declarar una lista de itinerarios de la siguiente forma itinerarios = [salidaDeAmigos, mezclaExplosiva, itinerarioBasico]. Esto se realiza en la consola de Haskell de la siguiente manera:

```
*Main> itinerarios = [salidaDeAmigos, mezclaExplosiva, itinerarioBasico]
```

Luego, se procedió a llamar a la función hacerItinerarioIntenso con la lista de itinerarios que se declaró y con el cliente Rodri y nos dio como resultado

```
*Main> hacerItinerarioIntenso itinerarios rodri
Nombre: erpRodri
Resistencia: 45
Amigos:Roberto Carlos (Resistencia: 155)
Bebidas: JarraLoca, Tintico, Soda De Fuerza:1, Tintico
*Main>
```

Como se puede observar en esta consulta se ven reflejados los apartados pedidos

- Rodri queda con Roberto Carlos como único amigo.
- La resistencia de Rodri queda en 45.
- El nombre de Rodri queda en erpRodri.

Adicionalmente si preguntamos por el itinerario más intenso obtenemos que el mismo es el siguiente:

```
*Main> obtenerItinerarioMasIntenso itinerarios
Salida de amigos
*Main>
```

Por lo cual, la salida de amigos es el itinerario más intenso.

Objetivo 5

- Roberto Carlos se hace amigo de Ana, toma una jarra popular de espirituosidad 0, sigue quedando con una sola amiga (Ana).

Para que Roberto Carlos tome una jarra popular y comience con Ana como amiga se definió a robertoAux el cual agrega a ana como amiga de robertoCarlos y luego se sobreescribe el valor de robertoCarlos con robertoAux. Una vez tenido esto, se ejecuta la funcion tomarBebida pasandole la JarraPopular con espiritualidad 0 y a robertoCarlos y se obtiene lo siguiente:

```
*Main> let robertoAux= (agregarAmigo robertoCarlos ana)
*Main> let robertoCarlos=robertoAux
*Main> (tomarBebida(JarraPopular 0)robertoCarlos)
Nombre: Roberto Carlos
Resistencia: 165
Amigos:Ana (Resistencia: 120)
Bebidas: Jarra popular de espirituosidad: 0
*Main>
```

Donde, como se puede ver, Roberto Carlos solo se queda con Ana como su única amiga debido a que la jarra popular tiene espiritualidad 0.

- Roberto Carlos se hace amigo de Ana, toma una jarra popular de espirituosidad 3, queda con 3 amigos (Ana, Marcos y Rodri).

Previo a que Roberto Carlos tomé la jarra popular se realiza lo mismo del ítem anterior para poder guardar a Ana como amiga y luego se ejecuta la función tomarBebida pasandole la jarra popular con espirituosidad 3 y a Roberto Carlos. El resultado obtenido es el siguiente:

```
*Main> let robertoAux= (agregarAmigo robertoCarlos ana)
*Main> let robertoCarlos=robertoAux
*Main> (tomarBebida(JarraPopular 3)robertoCarlos)
Nombre: Roberto Carlos
Resistencia: 165
Amigos:Rodri (Resistencia: 55), Marcos (Resistencia: 40), Ana (Resistencia: 120)
Bebidas: Jarra popular de espirituosidad: 3
*Main>
```

Donde, como se puede ver, Roberto Carlos queda con Ana, Marcos y Rodri como amigos ya que al ser la Jarra Popular de espirituosidad 3, el mismo se hace amigo de los amigos de los amigos. Es decir, Ana es amiga de Rodri y Marcos por lo cual ellos también se hacen amigos de Roberto Carlos.

- Cristian se hace amigo de Ana. Roberto Carlos se hace amigo de Cristian, toma una jarra popular de espirituosidad 4, queda con 4 amigos (Cristian, Ana, Marcos y Rodri).

Antes de que Roberto Carlos tomé la jarra popular se realiza lo mismo que se realizó en los ítems anteriores tanto para Roberto Carlos como para Cristian para guardar a Ana como amiga de ellos y luego se ejecuta la función tomarBebida pasandole la jarra popular con espirituosidad 4 y a Roberto Carlos. El resultado obtenido es el siguiente:

```
Ok, one module loaded.
*Main> let cristianAux= (agregarAmigo cristian ana)
*Main> let cristian=cristianAux
*Main> let robertoAux= (agregarAmigo robertoCarlos cristian)
*Main> let robertoCarlos=robertoAux
*Main> (tomarBebida(JarraPopular 4)robertoCarlos)
Nombre: Roberto Carlos
Resistencia: 165
Amigos:Rodri (Resistencia: 55), Marcos (Resistencia: 40), Ana (Resistencia: 120), Cristian (Resistencia: 2)
Bebidas: Jarra popular de espirituosidad: 4
*Main>
```

Donde, Roberto Carlos queda con 4 amigos debido a que la jarra popular es de espirituosidad 4 y esto le permite hacerse amigo de los amigos de los amigos de los amigos.