1 Introducción - Teoría de patrones

1.1 ¿Que es un patrón?

En Haskell, un patrón es una manera de hacer coincidir el valor de una expresión con una estructura específica. Los patrones son utilizados principalmente en las definiciones de funciones para especificar cómo se comportará la función en diferentes casos. Esto es fundamental en Haskell, ya que el lenguaje está basado en la evaluación de expresiones mediante la coincidencia de patrones.

Ahora, hablemos de la importancia de cubrir todos los casos en una función. En Haskell, al definir una función, es crucial considerar todos los posibles casos de entrada y asegurarse de que la función tenga un comportamiento definido para cada uno de ellos. Si no se cubren todos los casos, Haskell lanzará un error de "no exhaustividad" o "no exhaustivo", indicando que la definición de la función no cubre todas las posibilidades.

La importancia de cubrir todos los casos radica en garantizar la totalidad de la función, es decir, asegurarse de que la función pueda manejar cualquier entrada posible de manera predecible. Esto hace que el código sea más robusto y menos propenso a errores.

Aquí tienes un ejemplo que no se trata de calcular el factorial, sino de una función simple que convierte un número a su nombre en español:

Listing 1: Función que convierte un número del 0 al 5 a su nombre en español

```
numerosEnPalabras :: Int → String
numerosEnPalabras 0 = "cero"
numerosEnPalabras 1 = "uno"
numerosEnPalabras 2 = "dos"
numerosEnPalabras 3 = "tres"
numerosEnPalabras 4 = "cuatro"
numerosEnPalabras 5 = "cinco"
numerosEnPalabras n
    | n < 0     = "negativo"
    | otherwise = "demasiado grande"
```

En este ejemplo, la función `numerosEnPalabras` convierte un número entero en su representación de palabras en español. Se definen patrones para los números del 0 al 5, y luego se utiliza un patrón más general para cubrir cualquier otro número. Es importante notar que se cubren tres casos: números del 0 al 5, números negativos y números mayores que 5. Si no se hubiera incluido el último patrón, Haskell habría lanzado un error de no exhaustividad. Esto demuestra la importancia de cubrir todos los casos en la definición de una función en Haskell.

1.2 Sintaxis de patrones

En Haskell, la sintaxis de patrones se utiliza para definir diferentes casos de comportamiento de una función en función de los valores de entrada. La sintaxis básica de patrones se ve así:

```
\begin{verbatim}
nombreFuncion patrón1 = expresión1
nombreFuncion patrón2 = expresión2
...
nombreFuncion patrónN = expresiónN
```

Aquí hay una breve explicación de cada parte:

- `nombreFuncion`: Es el nombre de la función que estás definiendo.
- `patrón1`, `patrón2`, ..., `patrónN`: Son los patrones que la función coincidirá con la entrada.
- `expresión1`, `expresión2`, ..., `expresiónN`: Son las expresiones que se evaluarán si el patrón correspondiente coincide con la entrada.

Los patrones pueden ser simples, como valores literales o variables, o más complejos, como patrones de lista o tupla. Aquí hay algunos ejemplos de patrones:

Listing 2: Ejemplos de patrones en Haskell

```haskell
-- Patrones literales:
f 0 = "Cero"
f 1 = "Uno"

-- Patrones de variables:
g x = x * 2

-- Patrones de lista:
sumarLista [] = 0
sumarLista (x:xs) = x + sumarLista xs

-- Patrones de tupla:
getFirstElement (x, _) = x

-- Guardas:
categorizarEdad edad
    | edad < 18 = "Menor de edad"
    | edad >= 18 && edad < 65 = "Adulto"
    | otherwise = "Adulto mayor"
```

La coincidencia de patrones se evalúa en orden, de arriba a abajo, y la primera coincidencia se usa. Si ningún patrón coincide, Haskell generará un error en tiempo de ejecución. Es por eso que es importante cubrir todos los casos posibles al definir una función.

1.3 Patrones literales

Los patrones literales en Haskell son una forma de coincidir exactamente con un valor específico de entrada. Esto significa que si la entrada es igual al valor especificado en el patrón, entonces se ejecutará la expresión asociada. Aquí tienes más detalles sobre los patrones literales:

- **Sintaxis**: Los patrones literales son simplemente valores literales, como números enteros, caracteres, cadenas de texto o valores booleanos.

  Por ejemplo, en la función:

  ```haskell
  esCero 0 = True
  esCero _ = False
  ```

  El patrón literal `0` se utiliza para verificar si la entrada es igual a cero.

- **Coincidencia exacta**: Los patrones literales coinciden exactamente con los valores dados. Si la entrada no coincide exactamente con el patrón literal, la coincidencia no se producirá.

- **Prioridad de coincidencia**: Cuando se evalúa una función con múltiples patrones, los patrones literales se evalúan en el orden en que se han definido. El primer patrón que coincida con la entrada se utilizará, y los patrones posteriores no se evaluarán.

- **Uso en guardas**: Los patrones literales también pueden usarse en combinación con guardas para aplicar condiciones adicionales a la coincidencia.

1.4 Patrones de variables

Los patrones de variables en Haskell son una forma de vincular un nombre a un valor que coincide con el patrón de entrada. Estos patrones coinciden con cualquier valor de entrada y se utilizan para capturar ese valor y utilizarlo en la expresión asociada. Aquí tienes más detalles sobre los patrones de variables:

- **Sintaxis**: Los patrones de variables son simplemente nombres de variables que se utilizan en lugar de valores específicos en los patrones de coincidencia. Estas variables pueden ser utilizadas en la expresión asociada para realizar cálculos o tomar decisiones basadas en el valor capturado.
- **Coincidencia con cualquier valor**: Los patrones de variables coinciden con cualquier valor de entrada. Esto significa que la variable en el patrón se vinculará al valor de entrada, independientemente de lo que sea.
- **Uso en expresiones**: Una vez que se ha vinculado una variable a un valor de entrada, esa variable se puede utilizar en la expresión asociada para realizar cálculos o tomar decisiones.
- **Nombre de la variable**: El nombre de la variable en el patrón puede ser cualquier identificador válido en Haskell y puede ser utilizado como cualquier otra variable en el ámbito donde se define la función.

Aquí tienes un ejemplo de cómo se utilizan los patrones de variables en Haskell:

Listing 3: Ejemplo de patrones de variables en Haskell

```
doble x = x * 2
```

En este ejemplo, el patrón de variable `x` se utiliza para capturar cualquier valor de entrada que se pase a la función `doble`. La variable `x` se vincula al valor de entrada y luego se utiliza en la expresión `x * 2` para calcular el doble del valor de entrada.

Los patrones de variables son útiles cuando necesitas acceder al valor de entrada dentro de la función para realizar operaciones o tomar decisiones basadas en ese valor.

1.5 Patrones de lista

Los patrones de lista en Haskell permiten descomponer una lista en sus elementos individuales para su procesamiento. Estos patrones son especialmente útiles cuando se trabaja con funciones que operan en listas, ya que permiten manejar diferentes casos de manera más concisa y expresiva. Aquí tienes más detalles sobre los patrones de lista:

- **Sintaxis**: Los patrones de lista se definen entre corchetes `[ ]`, con los elementos separados por comas. También pueden incluir un patrón para el resto de la lista utilizando el operador `:`.
- **Descomposición de listas**: Los patrones de lista permiten descomponer la lista en sus elementos individuales y/o en una cabeza (primer elemento) y una cola (resto de la lista).
- **Captura de elementos individuales**: Los elementos individuales de la lista pueden ser capturados por patrones específicos.
- **Patrón de lista vacía**: Es posible definir un patrón para una lista vacía utilizando simplemente `[]`.
- **Patrón de lista con un solo elemento**: También se puede definir un patrón para una lista con un solo elemento utilizando la forma `[x]`, donde `x` es el elemento.
- **Patrón de lista con múltiples elementos**: Para descomponer una lista con múltiples elementos, se puede utilizar el operador `:` para separar el primer elemento (cabeza) del resto de la lista (cola).

Aquí tienes un ejemplo de cómo se utilizan los patrones de lista en Haskell:

Listing 4: Función que calcula la longitud de una lista

```
longitudLista [] = 0
longitudLista (x:xs) = 1 + longitudLista xs
```

En este ejemplo, la función `longitudLista` calcula la longitud de una lista. En el primer patrón, `[]` coincide con una lista vacía, por lo que devuelve `0`. En el segundo patrón, `(x:xs)` descompone la lista en su primer elemento `x` y el resto de la lista `xs`, y se calcula recursivamente la longitud de `xs`, sumando `1` para cada elemento de la lista.

Los patrones de lista son una característica poderosa de Haskell que permite expresar operaciones comunes en listas de forma concisa y elegante.

1.6 Patrones de tupla

Los patrones de tupla en Haskell permiten descomponer una tupla en sus componentes individuales para su procesamiento. Estos patrones son especialmente útiles cuando se trabaja con funciones que operan en tuplas, ya que permiten manejar los elementos de la tupla de manera más concisa y expresiva. Aquí tienes más detalles sobre los patrones de tupla:

- **Sintaxis**: Los patrones de tupla se definen utilizando paréntesis `( )`, con los patrones para cada componente separados por comas.
- **Descomposición de tuplas**: Los patrones de tupla permiten descomponer la tupla en sus componentes individuales, lo que facilita su manipulación dentro de una función.
- **Captura de componentes individuales**: Los patrones de tupla permiten capturar cada componente de la tupla utilizando patrones específicos para cada uno.
- **Nombre de las variables**: Los nombres de las variables utilizadas en los patrones de tupla pueden ser cualquier identificador válido en Haskell y se pueden utilizar en la expresión asociada para realizar cálculos o tomar decisiones basadas en los componentes de la tupla.

Aquí tienes un ejemplo de cómo se utilizan los patrones de tupla en Haskell:

```
1 getFirstElement (x, _) = x
```

En este ejemplo, la función `getFirstElement` toma una tupla como entrada y devuelve el primer elemento de la tupla. El patrón `(x, _)` descompone la tupla en su primer componente `x` y el segundo componente (que se ignora con el patrón comodín `_`), y luego simplemente devuelve `x`.

Los patrones de tupla son útiles cuando necesitas manipular tuplas dentro de una función y acceder a sus componentes de manera individual para realizar operaciones específicas sobre ellos.

1.7 Guardas

Los patrones en guardas en Haskell permiten aplicar condiciones adicionales a los patrones de coincidencia. Esto significa que puedes combinar la coincidencia de patrones con expresiones booleanas (guardas) para determinar si un patrón específico debe aplicarse o no. Aquí tienes más detalles sobre el uso de patrones en guardas:

- **Sintaxis**: Las guardas se definen utilizando el símbolo `|` seguido de una expresión booleana (guarda) que se evalúa. Si la guarda es verdadera, el patrón correspondiente se aplica. Si es falsa, Haskell pasa al siguiente patrón.
- **Condiciones adicionales**: Las guardas permiten especificar condiciones adicionales que deben cumplirse para que se aplique un patrón específico.
- **Flexibilidad**: Las guardas pueden ser tan simples o tan complejas como se desee, lo que permite una gran flexibilidad en la definición de patrones en funciones.
- **Orden de evaluación**: Las guardas se evalúan en el orden en que aparecen en la definición de la función. El primer patrón cuya guarda sea verdadera se aplicará y el resto de los patrones se ignorarán.

Aquí tienes un ejemplo de cómo se utilizan los patrones en guardas en Haskell:

```haskell
categorizarEdad edad
    | edad < 18 = "Menor de edad"
    | edad >= 18 && edad < 65 = "Adulto"
    | otherwise = "Adulto mayor"
```

En este ejemplo, la función `categorizarEdad` toma la edad como entrada y devuelve una cadena que representa la categoría de edad. La primera guarda verifica si la edad es menor que 18, la segunda guarda verifica si la edad está entre 18 y 64 (inclusive) y la última guarda (usando `otherwise`, que es equivalente a `True`) captura cualquier otra edad que no haya sido cubierta por los patrones anteriores.

Los patrones en guardas son una herramienta poderosa que permite definir funciones con comportamientos específicos basados en condiciones adicionales más allá de la coincidencia de patrones básica.

1.8 Patrones con datos algebraicos

La aplicación de patrones con datos algebraicos en Haskell permite descomponer y manipular valores de tipos de datos definidos algebraicamente. Estos tipos de datos son definidos mediante una suma de productos de tipos, lo que significa que pueden contener variantes diferentes y cada variante puede contener varios campos. Aquí tienes más detalles sobre la aplicación de patrones con datos algebraicos:

- **Tipos de datos algebraicos**: Los tipos de datos algebraicos en Haskell se definen mediante declaraciones de datos (declaraciones `data`), donde se enumeran las diferentes variantes del tipo de datos junto con los tipos de sus campos, si los tienen.
- **Descomposición de datos algebraicos**: Los patrones se pueden utilizar para descomponer valores de tipos de datos algebraicos en sus componentes individuales. Esto permite acceder a los campos de los valores y manipularlos de acuerdo a las necesidades.
- **Coincidencia de patrones con variantes**: Cada variante de un tipo de datos algebraicos se puede utilizar como un patrón en las definiciones de funciones. Esto permite definir diferentes comportamientos para cada variante del tipo de datos.
- **Patrones anidados**: Los patrones anidados permiten descomponer tipos de datos algebraicos que contienen otros tipos de datos algebraicos como campos.
- **Flexibilidad y expresividad**: La aplicación de patrones con datos algebraicos proporciona una gran flexibilidad y expresividad en la definición de funciones en Haskell, permitiendo definir comportamientos específicos para diferentes variantes de tipos de datos.

Aquí tienes un ejemplo de cómo se utilizan los patrones con datos algebraicos en Haskell:

```haskell
data Animal = Perro String Int | Gato String

esPerro :: Animal → Bool
esPerro (Perro _ _) = True
esPerro _ = False
```

En este ejemplo, `Animal` es un tipo de datos algebraicos que tiene dos variantes: `Perro`, que tiene un nombre y una edad como campos, y `Gato`, que solo tiene un nombre. La función `esPerro` toma un valor de tipo `Animal` como entrada y devuelve `True` si es un perro (es decir, si es de la variante `Perro`), y `False` en caso contrario.

La aplicación de patrones con datos algebraicos es una característica poderosa de Haskell que permite trabajar de manera eficiente y expresiva con tipos de datos definidos por el usuario.

1.9 Patrones con datos algebraicos recursivos

Cuando se trabajan con datos algebraicos recursivos en Haskell, los patrones se utilizan para descomponer y manipular estos tipos de datos de manera eficiente. Los datos algebraicos recursivos son tipos de datos

que se definen en términos de sí mismos, lo que permite crear estructuras de datos complejas y recursivas. Aquí tienes más detalles sobre el uso de patrones en datos algebraicos recursivos:

- **Recursión estructural**: Los datos algebraicos recursivos se definen de manera que una variante del tipo de datos pueda contener uno o más campos que son del mismo tipo que el tipo que se está definiendo. Esto permite la creación de estructuras de datos que pueden ser recursivamente anidadas.
- **Patrones de datos recursivos**: Los patrones se utilizan para descomponer los datos algebraicos recursivos en sus componentes individuales. Esto implica la aplicación de patrones recursivamente para trabajar con cada nivel de la estructura de datos.
- **Casos base en la recursión**: En el caso de datos algebraicos recursivos, es importante definir casos base en las funciones que manipulan estos datos. Estos casos base manejan las instancias más simples de la estructura de datos, evitando así la recursión infinita.
- **Patrones anidados**: Los patrones anidados se utilizan para descomponer datos algebraicos recursivos que contienen otros datos algebraicos recursivos como campos.
- **Recursión estructural descendente**: En general, la recursión estructural en datos algebraicos recursivos sigue un enfoque descendente, donde se descompone la estructura de datos en sus partes más pequeñas y luego se trabaja en cada parte individualmente.

Aquí tienes un ejemplo de cómo se utiliza el uso de patrones en datos algebraicos recursivos en Haskell:

```haskell
data Arbol = Nodo Int Arbol Arbol | Hoja Int | Nulo

sumaArbol :: Arbol → Int
sumaArbol Nulo = 0
sumaArbol (Hoja valor) = valor
sumaArbol (Nodo valor izquierda derecha) =
    valor + sumaArbol izquierda + sumaArbol derecha
```

En este ejemplo, `Arbol` es un tipo de datos algebraicos recursivos que representa un árbol binario, donde cada nodo tiene un valor entero y dos hijos (que son árboles). La función `sumaArbol` calcula la suma de todos los valores en el árbol. Tiene tres patrones: uno para el caso base `Nulo`, uno para las hojas del árbol y otro para los nodos internos. En el último caso, la función se llama recursivamente sobre los subárboles izquierdo y derecho.

El uso de patrones en datos algebraicos recursivos es una técnica fundamental en Haskell para trabajar con estructuras de datos complejas y recursivas de manera clara y concisa.

2 Aplicación de patrones en casos prácticos

2.1 Ejemplo 1:

Vamos a representar un tren de carga usando Haskell y para ello se deben definir nuevos tipos:

- Primeramente se debe definir el tipo `Item` que tiene constructores `Azucar, Cafe, Maiz, Trigo` y `Yerba`, todos ellos sin parámetros. El tipo `Item` **no debe pertenecer** a la clase `Eq`.
- Luego se debe definir el tipo `Toneladas` como sinónimo del tipo `Int`.
- El tipo `Cargamento` debe tener dos constructores:
  - Constructor `SinCarga`: No tiene parámetros y representa el cargamento vacío.
  - Constructor `Carga`: Tiene dos parámetros, el primero de tipo `Item` (indica el tipo de carga que tien el cargamento) y el segundo parámetro es de tipo `Toneladas` (indica la cantidad de toneladas de ese tipo de carga).
- El tipo `Numeracion` que debe ser un sinónimo del tipo `Int`.
- El tipo `Tren` que tiene dos constructores
  - Constructor `Vagon`: Tiene tres parámetros, el primero de tipo `Numeracion` (la numeración del vagón), el segundo de tipo `Cargamento` (la carga que lleva el vagón) y el tercero de tipo `Tren` que es el resto del tren.
  - Constructor `Fin`: No tiene parámetros y representa el final del tren.

Asegurarse que los tipos `Tren`, `Cargamento` e `Item` estén en la clase `Show`.

Ejercicio 1
Programar la función

```
vagones_item :: Tren → Item → [Numeracion]
```

que dado un tren `ts` y un ítem `i`, devuelve las numeraciones de los vagones del tren `ts` que transportan el ítem `i`.

Ejercicio 2
Dar una expresión de tipo `Tren` que tenga al menos 3 elementos donde se utilicen los dos constructores de tipo.

Primero se deben representar los tipos de datos que se piden en el enunciado:

```haskell
data Item = Azucar | Cafe | Maiz | Trigo | Yerba
data Cargamento = SinCarga | Carga Item Toneladas
type Toneladas = Int
type Numeracion = Int
data Tren = Vagon Numeracion Cargamento Tren | Fin
```

Luego la consigna nos dice que dado un tren `ts` y un ítem `i`, se debe devolver las numeraciones de los vagones del tren `ts` que transportan el ítem `i`. Para ello se debe programar la función `vagones_item` que recibe un tren y un ítem y devuelve una lista de numeraciones. Esto significa que debemos recorrer el tren y verificar si el vagón transporta el ítem que se nos pide. Si es así, debemos agregar la numeración del vagón a la lista de numeraciones. Si no, debemos seguir recorriendo el tren. La función `vagones_item` se puede programar de la siguiente manera:

- Dado que `Item` no pertenece a la clase `Eq`, no podemos utilizar la función `==` para comparar dos ítems. Por lo tanto, debemos programar una función que compare dos ítems y nos diga si son iguales o no. Esta función se puede programar de la siguiente manera:

```haskell
    son_iguales :: Item → Item → Bool
    son_iguales Azucar Azucar = True
    son_iguales Cafe Cafe = True
    son_iguales Maiz Maiz = True
    son_iguales Trigo Trigo = True
    son_iguales Yerba Yerba = True
    son_iguales _ _ = False
```

En este ejemplo utilizamos patrones literales para comparar dos ítems. Si los ítems son iguales, devolvemos `True`, si no, devolvemos `False`.

- Podemos tambien definir una función auxiliar para verificar si tiene un item:

```haskell
    tiene_item :: Cargamento → Item → Bool
    tiene_item SinCarga _ = False
    tiene_item (Carga i _) j = son_iguales i j
```

- Luego, la función `vagones_item` se puede programar de la siguiente manera:

```haskell
    vagones_item :: Tren → Item → [Numeracion]
    vagones_item Fin _ = []
    vagones_item (Vagon n c t) i
        | tiene_item c i = n : vagones_item t i
        | otherwise = vagones_item t i
```

En este ejercicio, usamos pattern matching en un tipo de dato algebraico recursivo para recorrer el tren y verificar si cada vagón transporta el ítem que se nos pide. Si es así, agregamos la numeración del vagón a la lista de numeraciones. Si no, seguimos recorriendo el tren.

El código completo de este ejercicio se puede ver a continuación:

```haskell
-- Definicion de tipos
data Item = Azucar | Cafe | Maiz | Trigo | Yerba
data Cargamento = SinCarga | Carga Item Toneladas
type Toneladas = Int
type Numeracion = Int
data Tren = Vagon Numeracion Cargamento Tren | Fin

-- Funciones auxiliares
son_iguales :: Item → Item → Bool
son_iguales Azucar Azucar = True
son_iguales Cafe Cafe = True
son_iguales Maiz Maiz = True
son_iguales Trigo Trigo = True
son_iguales Yerba Yerba = True
son_iguales _ _ = False

tiene_item :: Cargamento → Item → Bool
tiene_item SinCarga _ = False
tiene_item (Carga i _) j = son_iguales i j

-- Funcion principal
vagones_item :: Tren → Item → [Numeracion]
vagones_item Fin _ = []
vagones_item (Vagon n c t) i
    | tiene_item c i = n : vagones_item t i
    | otherwise = vagones_item t i
```

Ahora hay que dar la expresión de tipo `Tren` que tenga al menos 3 elementos donde se utilicen los dos constructores de tipo. Esto se puede hacer de la siguiente manera:

```
tren_ejemplo :: Tren
tren_ejemplo = Vagon 1 (Carga Azucar 10) (Vagon 2 (Carga Cafe 5) (Vagon 3 (Carga
    Maiz 15) Fin))
```

En este ejemplo, utilizamos los dos constructores de tipo `Tren` para crear un tren con al menos 3 elementos. El tren tiene tres vagones, cada uno con un número de vagón, un tipo de carga y el resto del tren. El último vagón tiene el constructor `Fin` para indicar que es el final del tren. Podriamos ejecutar la funcion con este tren:

```
*Main> vagones_item tren_ejemplo Azucar
[1]
*Main> vagones_item tren_ejemplo Cafe
[2]
*Main> vagones_item tren_ejemplo Maiz
[3]
*Main> vagones_item tren_ejemplo Trigo
[]
```

2.2 Ejemplo 2:

Se van a representar los elementos de la heladera usando tipos en Haskell. Los productos que tendremos en cuenta son: `Leche`, `Carne`, `Queso`. La idea es poder detallar para cada tipo de producto, las características mas importantes. En tal sentido identificamos las siguientes características de cada uno de estos productos a tener en cuenta:
`Leche`:

- `TipoDeLeche`, que es un tipo enumerado con las siguientes opciones: `Descremada`, `Entera`, `Condenzada`, `Polvo`.
- `UsoDeLeche`, que es un tipo enumerado con las siguientes opciones: `Bebida`, `Preparaciones`.
- `Precio`, que es un sinónimo de `Int` indicando el precio.

`Carne`

- `Corte`, que es un tipo enumerado con las siguientes opciones: `Bife`, `Molida`, `Pulpa`.
- `Peso`, que es un sinónimo de `Float` indicando el peso.
- `Precio`, que es un sinónimo de `Int` indicando el precio.

`Queso`

- `TipoDeQueso`, que es un tipo enumerado con las siguientes opciones: `Barra`, `Cremoso`, `Duro`.
- `Peso`, que es un sinónimo de `Float` indicando el peso.
- `Precio`, que es un sinónimo de `Int` indicando el precio.

Para ello

Ejercicio 1
Definir el tipo `Perecedero` que consta de los contructores `Leche`, `Carne` y `Queso`, constructores con parámetros descritos arriba (Se deben tambien definir los tipos enumerados `TipoDeLeche`, `UsoDeLeche`, `Corte`, `TipoDeQueso` y los sinónimos de `Precio` y de `Peso`). **Los tipos `Perecedero` y `TipoDeQueso` no deben estar en la clase Eq ni en la clase Ord**.

Ejercicio 2
Definir la función `cuantosQuesos` de la siguiente manera

```
1        cuantosQuesos :: [Perecedero] → TipoDeQueso → Int
```

que dada una lista de `Perecedero lp` y un valor `q` de tipo `TipoDeQueso`, me devuelve un entero indicando la cantidad de quesos que hay en `lp` con el tipo `q`. **Nota:** dejar como comentario un ejemplo donde hayas probado la funcion `cuantosQuesos` con una lista con al menos 3 `Perecedero`.

Ejercicio 3
Definir igualdad para el tipo de `Perecedero`: de tal manera que, dos elementos de tipo `Leche` son iguales sólo si tienen el mismo **tipo de leche** y el mismo **uso de leche**, dos `Carne` son iguales solo si tienen el mismo **corte**, mientras que dos `Quesos` son iguales si tiene el mismo **tipo de queso**. Y como es de suponer las `Leches`, `Carnes` y `Quesos` son distintos entre sí. **Nota:** Dejar como comentario en el código dos ejemplos en los que probaste la igualdad.

Primero se deben representar los tipos de datos que se piden en el enunciado:

```
data TipoDeLeche = Descremada | Entera | Condenzada | Polvo
data UsoDeLeche = Bebida | Preparaciones
type Precio = Int

data Corte = Bife | Molida | Pulpa
type Peso = Float

data TipoDeQueso = Barra | Cremoso | Duro
```

Luego se debe definir el tipo `Perecedero` que consta de los contructores `Leche`, `Carne` y `Queso`, constructores con parámetros descritos arriba. Además, se deben definir los tipos enumerados `TipoDeLeche`, `UsoDeLeche`, `Corte`, `TipoDeQueso` y los sinónimos de `Precio` y de `Peso`. Los tipos `Perecedero` y `TipoDeQueso` no deben estar en la clase Eq ni en la clase Ord. Esto se puede hacer de la siguiente manera:

```
data Perecedero = Leche TipoDeLeche UsoDeLeche Precio | Carne Corte Peso Precio |
    Queso TipoDeQueso Peso Precio
```

Luego se debe definir la función `cuantosQuesos` que recibe una lista de `Perecedero` y un valor de tipo `TipoDeQueso` y devuelve un entero indicando la cantidad de quesos que hay en la lista con el tipo `q`. Esto se puede hacer de la siguiente manera:

- Dado que no podemos comparar los valores de tipo `TipoDeQueso` con la función `==`, debemos programar una función que compare dos valores de tipo `TipoDeQueso` y nos diga si son iguales o no. Esta función se puede programar de la siguiente manera:

```
    son_iguales_queso :: TipoDeQueso → TipoDeQueso → Bool
    son_iguales_queso Barra Barra = True
    son_iguales_queso Cremoso Cremoso = True
    son_iguales_queso Duro Duro = True
    son_iguales_queso _ _ = False
```

- Luego, la función `cuantosQuesos` se puede programar de la siguiente manera:

```
    cuantosQuesos :: [Perecedero] → TipoDeQueso → Int
    cuantosQuesos [] _ = 0
    cuantosQuesos (Queso t _ _ : ps) q
        | son_iguales_queso t q = 1 + cuantosQuesos ps q
        | otherwise = cuantosQuesos ps q
    cuantosQuesos (_ : ps) q = cuantosQuesos ps q
```

En este ejercicio, usamos pattern matching en un tipo de dato algebraico para recorrer la lista de `Perecedero` y verificar si cada elemento es un queso del tipo que se nos pide. Si es así, agregamos `1` a la cantidad de quesos. Si no, seguimos recorriendo la lista.

Ahora, para definir igualdad para el tipo de `Perecedero`, debemos programar funciones auxiliares que nos permitan comparar tambien los valores de tipo `UsoDeLeche`, `TipoDeLeche` y `Corte`. Esto se puede hacer de la siguiente manera:

```
son_iguales_uso_leche :: UsoDeLeche → UsoDeLeche → Bool
son_iguales_uso_leche Bebida Bebida = True
son_iguales_uso_leche Preparaciones Preparaciones = True
son_iguales_uso_leche _ _ = False

son_iguales_tipo_leche :: TipoDeLeche → TipoDeLeche → Bool
son_iguales_tipo_leche Descremada Descremada = True
son_iguales_tipo_leche Entera Entera = True
son_iguales_tipo_leche Condenzada Condenzada = True
son_iguales_tipo_leche Polvo Polvo = True
son_iguales_tipo_leche _ _ = False

son_iguales_corte :: Corte → Corte → Bool
son_iguales_corte Bife Bife = True
son_iguales_corte Molida Molida = True
son_iguales_corte Pulpa Pulpa = True
son_iguales_corte _ _ = False
```

Luego, la función `==` para el tipo `Perecedero` se puede programar de la siguiente manera:

```
instance Eq Perecedero where
    (Leche t1 u1 _) == (Leche t2 u2 _) = son_iguales_tipo_leche t1 t2 &&
        son_iguales_uso_leche u1 u2
    (Carne c1 _ _) == (Carne c2 _ _) = son_iguales_corte c1 c2
    (Queso t1 _ _) == (Queso t2 _ _) = son_iguales_queso
```

Los ejemplos de la funcion `cuantosQuesos` se pueden ver a continuación:

```
*Main> cuantosQuesos [Queso Barra 10 100, Queso Cremoso 5 50, Queso Duro 15 150,
    Queso Barra 20 200] Barra
2
*Main> cuantosQuesos [Queso Barra 10 100, Queso Cremoso 5 50, Queso Duro 15 150,
    Queso Barra 20 200] Cremoso
1
*Main> cuantosQuesos [Queso Barra 10 100, Queso Cremoso 5 50, Queso Duro 15 150,
    Queso Barra 20 200] Duro
1
*Main> cuantosQuesos [Queso Barra 10 100, Queso Cremoso 5 50, Queso Duro 15 150,
    Queso Barra 20 200] Polvo
0
```

Y los ejemplos de la igualdad se pueden ver a continuación:

```
*Main> Leche Descremada Bebida 10 == Leche Descremada Bebida 20
True
*Main> Leche Descremada Bebida 10 == Leche Descremada Preparaciones 10
False
*Main> Carne Bife 10 100 == Carne Bife 10 100
True
*Main> Carne Bife 10 100 == Carne Molida 10 100
False
*Main> Queso Barra 10 100 == Queso Barra 10 100
True
*Main> Queso Barra 10 100 == Queso Cremoso 10 100
False
```

El código completo de este ejercicio se puede ver a continuación:

```haskell
-- Definicion de tipos
data TipoDeLeche = Descremada | Entera | Condenzada | Polvo
data UsoDeLeche = Bebida | Preparaciones
type Precio = Int

data Corte = Bife | Molida | Pulpa
type Peso = Float

data TipoDeQueso = Barra | Cremoso | Duro

data Perecedero = Leche TipoDeLeche UsoDeLeche Precio | Carne Corte Peso Precio |
    Queso TipoDeQueso Peso Precio

-- Funciones auxiliares
son_iguales_queso :: TipoDeQueso → TipoDeQueso → Bool
son_iguales_queso Barra Barra = True
son_iguales_queso Cremoso Cremoso = True
son_iguales_queso Duro Duro = True
son_iguales_queso _ _ = False

son_iguales_uso_leche :: UsoDeLeche → UsoDeLeche → Bool
son_iguales_uso_leche Bebida Bebida = True
son_iguales_uso_leche Preparaciones Preparaciones = True
son_iguales_uso_leche _ _ = False

son_iguales_tipo_leche :: TipoDeLeche → TipoDeLeche → Bool
son_iguales_tipo_leche Descremada Descremada = True
son_iguales_tipo_leche Entera Entera = True
son_iguales_tipo_leche Condenzada Condenzada = True
son_iguales_tipo_leche Polvo Polvo = True
son_iguales_tipo_leche _ _ = False

son_iguales_corte :: Corte → Corte → Bool
son_iguales_corte Bife Bife = True
son_iguales_corte Molida Molida = True
son_iguales_corte Pulpa Pulpa = True
son_iguales_corte _ _ = False

-- Funciones principales
cuantosQuesos :: [Perecedero] → TipoDeQueso → Int
cuantosQuesos [] _ = 0
cuantosQuesos (Queso t _ _ : ps) q
    | son_iguales_queso t q = 1 + cuantosQuesos ps q
    | otherwise = cuantosQuesos ps q
cuantosQuesos (_ : ps) q = cuantosQuesos ps q

instance Eq Perecedero where
    (Leche t1 u1 _) == (Leche t2 u2 _) = son_iguales_tipo_leche t1 t2 &&
        son_iguales_uso_leche u1 u2
    (Carne c1 _ _) == (Carne c2 _ _) = son_iguales_corte c1 c2
    (Queso t1 _ _) == (Queso t2 _ _) = son_iguales_queso t1 t2
```

2.3 Ejemplo 3:

Queremos hacer un programa, para que los de una escuela de futbol puedan saber si sus alumnos de una categoria pueden pasar al siguiente nivel o no.

Ejercicio 1
Definir un tipo recursivo `NotasDelClub`, que permite guardar las notas que tuvo cada alumno de una categoria en el año. El tipo `NotasDelClub`, tendrá dos contructores:

- `EvolucionDelJugador`, que tiene 6 parámetros:
  - `String`, para el nombre y apellido del alumno.
  - `Division` (tipo enumerado con la `Categoria` actual del jugador, `Septima`, `Sexta`, `Quinta` y `Cuarta`)
  - `Nota` (Evalúa la capacidad defensiva del jugador, Int entre 1 y 10)
  - `Nota` (Evalúa la capacidad de ataque del jugador, Int entre 1 y 10)
  - `Nota` (Evalúa la calidad de pases, Int entre 1 y 10)
  - `NotasDelClub`, recursión con el resto de las notas
- `NoHayMasJugadores`, que es un constructor sin parámetros, similar al de la lista vacía, para indicar que se terminaron las notas.

Ejercicio 2
La condición para poder promover a la siguiente división se describen a continuación, basándose en las notas obtenidas:

- Si el jugador está en `Septima` o `Sexta` división, debe evaluar por sobre 7 en alguna de las capacidades, y haber realizado pases con `calidad` de al menos 6.
- Si el jugador está en `Quinta`, debe tener al menos un 7 en cada capacidad, y al menos un 8 de calidad de pases.

Programar la función `pasaDeDivision`, que toma como primer parametro notas del tipo `NotasDelClub`, y como segundo parámetro `nombre` del tipo `String` y retorna un valor de tipo `Bool`, indicando si el alumno llamado `nombre` **pasa de división o no**.

```
1        pasaDeDivision :: NotasDelClub → String → Bool
```

Nota: dejar como comentario un ejemplo donde hayas probado `pasaDeDivision` con un parámetro de tipo `NotasDelClub` con al menos 3 alumnos.

Ejercicio 3
Programar la función `devolverDivision` con la siguiente declaración.

```
1        devolverDivision :: NotasDelClub → String → Maybe Division
```

que toma una variable de tipo `NotasDelClub`, y como segundo argumento un `nombre`, que identifica el alumno, y en caso que el alumno esté en `notas` (con una division `d`), retorna `Just d`, y en caso contrario `Nothing`. **Nota:** dejar como comentario un ejemplo donde hayas probado `devolverDivision`.

Primero se deben representar los tipos de datos que se piden en el enunciado:

```
1 data Division = Septima | Sexta | Quinta | Cuarta
2 type Nota = Int
3 data NotasDelClub = EvolucionDelJugador String Division Nota Nota Nota
    NotasDelClub | NoHayMasJugadores
```

Luego se debe programar la función `pasaDeDivision` que toma como primer parametro notas del tipo `NotasDelClub`, y como segundo parámetro `nombre` del tipo `String` y retorna un valor de tipo `Bool`, indicando si el alumno llamado `nombre` **pasa de división o no**. Esto se puede hacer de la siguiente manera:

- Dado que no podemos comparar los valores de tipo `Division` con la función `==`, debemos programar una función que compare dos valores de tipo `Division` y nos diga si son iguales o no. Esta función se puede programar de la siguiente manera:

```
1    son_iguales_division :: Division → Division → Bool
2    son_iguales_division Septima Septima = True
3    son_iguales_division Sexta Sexta = True
4    son_iguales_division Quinta Quinta = True
5    son_iguales_division Cuarta Cuarta = True
6    son_iguales_division _ _ = False
```

- Luego, la función `pasaDeDivision` se puede programar de la siguiente manera:

```
1    pasaDeDivision :: NotasDelClub → String → Bool
2    pasaDeDivision NoHayMasJugadores _ = False
3    pasaDeDivision (EvolucionDelJugador n d n1 n2 n3 t) nombre
4        | n == nombre = (son_iguales_division d Septima ||
                son_iguales_division d Sexta) && (n1 > 7 || n2 > 7 || n3 > 7) &&
                n3 > 6
5        | n == nombre = son_iguales_division d Quinta && n1 >= 7 && n2 >= 7 &&
                n3 >= 8
6        | otherwise = pasaDeDivision t nombre
```

  En este ejercicio, usamos pattern matching en un tipo de dato algebraico recursivo para recorrer las notas y luego por guardas cubrimos los casos que nos da la consigna para verificar si el alumno llamado `nombre` pasa de división o no. Si es así, devolvemos `True`, si no, seguimos recorriendo las notas.

Como ejemplos de la función `pasaDeDivision` se pueden ver a continuación:

```
1  *Main> let notas = EvolucionDelJugador "Juan" Septima 8 7 6 (EvolucionDelJugador "
       Pedro" Sexta 7 8 9 (EvolucionDelJugador "Carlos" Quinta 7 7 8 NoHayMasJugadores
       ))
2  *Main> pasaDeDivision notas "Juan"
3  True
4  *Main> pasaDeDivision notas "Pedro"
5  True
6  *Main> pasaDeDivision notas "Carlos"
7  True
8  *Main> pasaDeDivision notas "Lucas"
9  False
```

Ahora para programar la función `devolverDivision` se puede hacer de la siguiente manera:

```
1  devolverDivision :: NotasDelClub → String → Maybe Division
2  devolverDivision NoHayMasJugadores _ = Nothing
3  devolverDivision (EvolucionDelJugador n d _ _ _ t) nombre
4      | n == nombre = Just d
5      | otherwise = devolverDivision t nombre
```

Y como ejemplos de la función `devolverDivision` se pueden ver a continuación:

```
1  *Main> let notas = EvolucionDelJugador "Juan" Septima 8 7 6 (EvolucionDelJugador "
       Pedro" Sexta 7 8 9 (EvolucionDelJugador "Carlos" Quinta 7 7 8 NoHayMasJugadores
       ))
2  *Main> devolverDivision notas "Juan"
3  Just Septima
```

```
*Main> devolverDivision notas "Pedro"
Just Sexta
*Main> devolverDivision notas "Carlos"
Just Quinta
*Main> devolverDivision notas "Lucas"
Nothing
```

El código completo de este ejercicio se puede ver a continuación:

```haskell
-- Definicion de tipos
data Division = Septima | Sexta | Quinta | Cuarta
type Nota = Int
data NotasDelClub = EvolucionDelJugador String Division Nota Nota Nota
    NotasDelClub | NoHayMasJugadores

-- Funciones auxiliares
son_iguales_division :: Division → Division → Bool
son_iguales_division Septima Septima = True
son_iguales_division Sexta Sexta = True
son_iguales_division Quinta Quinta = True
son_iguales_division Cuarta Cuarta = True
son_iguales_division _ _ = False

-- Funciones principales
pasaDeDivision :: NotasDelClub → String → Bool
pasaDeDivision NoHayMasJugadores _ = False
pasaDeDivision (EvolucionDelJugador n d n1 n2 n3 t) nombre
    | n == nombre = (son_iguales_division d Septima || son_iguales_division d
       Sexta) && (n1 > 7 || n2 > 7 || n3 > 7) && n3 > 6
    | n == nombre = son_iguales_division d Quinta && n1 >= 7 && n2 >= 7 && n3 >= 8
    | otherwise = pasaDeDivision t nombre

devolverDivision :: NotasDelClub → String → Maybe Division
devolverDivision NoHayMasJugadores _ = Nothing
devolverDivision (EvolucionDelJugador n d _ _ _ t) nombre
    | n == nombre = Just d
    | otherwise = devolverDivision t nombre
```