Conjuntos y combinatoria

Taller de Álgebra I

Segundo cuatrimestre 2019

En las últimas clases estuvimos resolviendo problemas utilizando *recursión*.
Este método se basa en definir las funciones para dos casos escenciales:

Esquema recursivo

- El caso base (o *los casos base*).
- El caso general, basada en una solución para un parámetro "más cercano" al caso base.

En las últimas clases estuvimos resolviendo problemas utilizando *recursión*.
Este método se basa en definir las funciones para dos casos escenciales:

Esquema recursivo

- El caso base (o *los casos base*).
- El caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿A qué nos referimos con "más cercano"?

En las últimas clases estuvimos resolviendo problemas utilizando *recursión*.
Este método se basa en definir las funciones para dos casos escenciales:

Esquema recursivo

- El caso base (o *los casos base*).
- El caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿A qué nos referimos con "más cercano"?
Si queremos calcular `f(n)` a partir de `f( g(n) )`, necesitamos que la función `g` sea tal que al aplicarla a `n`, y luego a su resultado, y a su resultado, . . . , nos lleve a un caso base **en una cantidad finita de aplicaciones**.

En las últimas clases estuvimos resolviendo problemas utilizando *recursión*.
Este método se basa en definir las funciones para dos casos escenciales:

Esquema recursivo

- El caso base (o *los casos base*).
- El caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿A qué nos referimos con "más cercano"?
Si queremos calcular `f(n)` a partir de `f( g(n) )`, necesitamos que la función `g` sea tal que al aplicarla a `n`, y luego a su resultado, y a su resultado, . . . , nos lleve a un caso base **en una cantidad finita de aplicaciones**.
Con números naturales, una forma básica es definir `f(0)`, y luego `f(n)` utilizando `f(n-1)`.

Repasemos algunos problemas que resolvimos definiendo funciones recursivas:

- Casos que se resuelven aplicando recursivamente la definición sobre el entero anterior

Factorial

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Repaso de clases anteriores

Recursión en enteros

Repasemos algunos problemas que resolvimos definiendo funciones recursivas:

- Casos que se resuelven aplicando recursivamente la definición sobre el entero anterior

Factorial

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- Casos cuya definición recursiva no es respecto del entero inmediatamente anterior

División

```
dividir :: Integer -> Integer -> Integer
dividir a b | a < b      = 0
            | otherwise = 1 + dividir (a-b) b
```

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de n.

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de n.

```
sumaDivisores :: Integer -> Integer
sumaDivisores n = sumaDivisoresHasta n n

sumaDivisoresHasta :: Integer -> Integer -> Integer
sumaDivisoresHasta n 0 = 0
sumaDivisoresHasta n m
          | n `mod` m == 0 = m + sumaDivisoresHasta n (m-1)
          | otherwise      = sumaDivisoresHasta n (m-1)
```

Casos donde la recursión se debe realizar sobre un valor que no es el parámetro de entrada

Suma de divisores

Definir `sumaDivisores` que calcule la suma de todos los divisores positivos de n.

```
sumaDivisores :: Integer -> Integer
sumaDivisores n = sumaDivisoresHasta n n

sumaDivisoresHasta :: Integer -> Integer -> Integer
sumaDivisoresHasta n 0 = 0
sumaDivisoresHasta n m
          | n `mod` m == 0 = m + sumaDivisoresHasta n (m-1)
          | otherwise      = sumaDivisoresHasta n (m-1)
```

El esquema no cambia, pero debemos definir una función nueva con un parámetro adicional para poder aplicar la recursión.

Casos donde la recursión se debe realizar sobre múltiples parámetros

Sumas dobles

Definir una función que calcule $f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i \times j$

Casos donde la recursión se debe realizar sobre múltiples parámetros

Sumas dobles

Definir una función que calcule $f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i \times j$

```
sumaInterior :: Integer -> Integer -> Integer
sumaInterior n 0 = 0
sumaInterior n m = (n*m) + sumaInterior n (m-1)

sumaDoble :: Integer -> Integer -> Integer
sumaDoble 0 m = 0
sumaDoble n m = sumaInterior n m + sumaDoble (n-1) m
```

Casos donde la recursión se debe realizar sobre múltiples parámetros

Sumas dobles

Definir una función que calcule $f(n, m) = \sum_{i=1}^{n} \sum_{j=1}^{m} i \times j$

```
sumaInterior :: Integer -> Integer -> Integer
sumaInterior n 0 = 0
sumaInterior n m = (n*m) + sumaInterior n (m-1)

sumaDoble :: Integer -> Integer -> Integer
sumaDoble 0 m = 0
sumaDoble n m = sumaInterior n m + sumaDoble (n-1) m
```

El esquema no cambia, pero la operación para obtener `sumaDoble n m` a partir de `sumaDoble (n-1) m` involucra, a su vez, una función recursiva en m.

Repaso de clases anteriores

Listas

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

A diferencia de las tuplas, las listas de distintas longitudes son del mismo tipo. Sin embargo, las tuplas pueden definirse con elementos de distintos tipos.

El tipo lista (`[a]`) representa una colección de objetos de un mismo tipo.

A diferencia de las tuplas, las listas de distintas longitudes son del mismo tipo. Sin embargo, las tuplas pueden definirse con elementos de distintos tipos.

Los patrones fundamentales (*constructores*) de las listas son:

- `[]`, la lista vacía.
- `(x:xs)`, una lista con *head* `x` y *tail* `xs`.

Como con los enteros, vamos a usar recursión para resolver problemas sobre listas.
Volvamos al esquema recursivo:

Esquema recursivo

- Definimos una solución para el caso base (o *los casos base*).
- Definimos una solución en el caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿Cómo se aplica esta idea en una lista?

Como con los enteros, vamos a usar recursión para resolver problemas sobre listas.
Volvamos al esquema recursivo:

Esquema recursivo

- Definimos una solución para el caso base (o *los casos base*).
- Definimos una solución en el caso general, basada en una solución para un parámetro "más cercano" al caso base.

¿Cómo se aplica esta idea en una lista?
Vamos a necesitar saber resolver el problema para una lista con pocos (o ningún) elemento, y ser capaces de resolver el problema para una lista de n elementos suponiendo que tenemos la solución para una lista con **estrictamente menos** elementos.

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía:

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento:

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento:

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento: `(x:xs)`
- Patrón para la lista con dos elementos:

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento: `(x:xs)`
- Patrón para la lista con dos elementos: `(x:y:[])`, `[x,y]`
- Patrón para la lista con *al menos* dos elementos:

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento: `(x:xs)`
- Patrón para la lista con dos elementos: `(x:y:[])`, `[x,y]`
- Patrón para la lista con *al menos* dos elementos: `(x:y:xs)`

... Esto se puede extender hasta donde haga falta.

Para preguntar si estamos en el caso base, podemos utilizar la función `length`, o, en el caso de la lista vacía, haciendo `xs==[]`.
Pero también podemos utilizar *pattern matching* para describir esta pregunta de manera más directa:

- Patrón para la lista vacía: `[]`
- Patrón para la lista con un elemento: `(x:[])`, `[x]`
- Patrón para la lista con *al menos* un elemento: `(x:xs)`
- Patrón para la lista con dos elementos: `(x:y:[])`, `[x,y]`
- Patrón para la lista con *al menos* dos elementos: `(x:y:xs)`

... Esto se puede extender hasta donde haga falta.
En cualquier posición, si el valor del parámetro en el patrón no me interesa, le puedo dar nombre _ para que Haskell lo descarte.
Ej.

```
head :: [a] -> a
head (x:_) = x
```

Las listas pueden ser tanto parte de la entrada como de la salida de una función:

- Funciones que toman una lista como entrada

Longitud

```
longitud :: [Integer] -> Integer
longitud []     = 0
longitud (_:xs) = 1 + longitud xs
```

Las listas pueden ser tanto parte de la entrada como de la salida de una función:

- Funciones que toman una lista como entrada

Longitud

```
longitud :: [Integer] -> Integer
longitud []     = 0
longitud (_:xs) = 1 + longitud xs
```

- Funciones que devuelven una lista como salida

Listar Impares

```
imparesHasta :: Integer -> [Integer]
imparesHasta 0 = []
imparesHasta n | n `mod` 2 == 1 = n : imparesHasta (n-1)
               | otherwise      = imparesHasta (n-1)
```

También podemos tener un caso donde tanto la entrada como salida son listas

- Por ejemplo: definir una función que dada una lista l de enteros, devuelva una lista l' con los valores de l que son múltiplo de 7.

Filtrar listas

```
multiplosDe7 :: [Integer] -> [Integer]
multiplosDe7 []     = []
multiplosDe7 (x:xs) | x `mod` 7 == 0 = x:(multiplosDe7 xs)
                    | otherwise = multiplosDe7 xs
```

También podemos tener un caso donde tanto la entrada como salida son listas

- Por ejemplo: definir una función que dada una lista l de enteros, devuelva una lista l' con los valores de l que son múltiplo de 7.

Filtrar listas

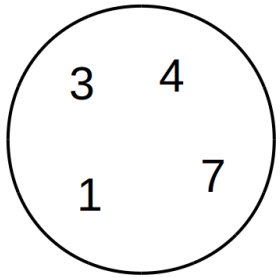
```
multiplosDe7 :: [Integer] -> [Integer]
multiplosDe7 []     = []
multiplosDe7 (x:xs) | x `mod` 7 == 0 = x:(multiplosDe7 xs)
                    | otherwise = multiplosDe7 xs
```

Todos estos esquemas son importantes!

Repasen los ejercicios que fueron vistos en clase, resuelvan los que no pudieron hacer antes y ¡consulten!

Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

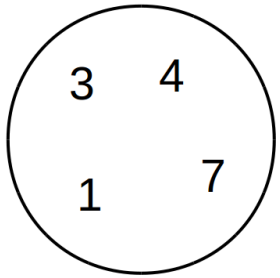


¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.

Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

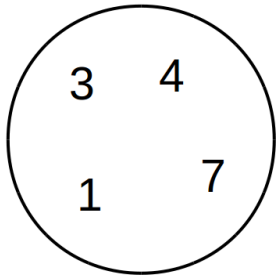


¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - El orden de los elementos es relevante para las listas, pero no para conjuntos.

Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

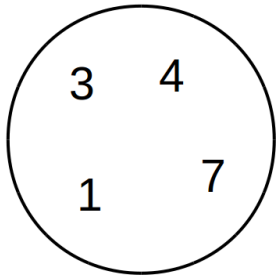


¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - El orden de los elementos es relevante para las listas, pero no para conjuntos.
- ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?

Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.

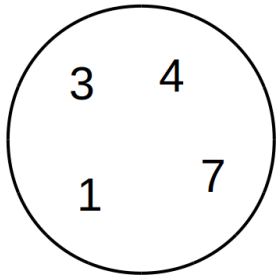


¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - El orden de los elementos es relevante para las listas, pero no para conjuntos.
- ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?
  - Las listas pueden tener elementos repetidos, pero eso no tiene sentido con conjuntos.

Conjuntos

Supongamos que queremos representar un **conjunto** de números enteros.



¿Es buena idea usar una lista `[Integer]`?

- Podríamos representar ese conjunto con la lista `[1,3,4,7]`.
  - También con `[4,1,3,7]`, `[3,7,4,1]`, `[7,3,1,4]`, ...
  - Todas estas listas son **distintas**, pero representan al **mismo** conjunto.
  - El orden de los elementos es relevante para las listas, pero no para conjuntos.
- ¿Y la lista `[1,3,4,7,7,7,1,4,7]`? ¿Sirve para representar a nuestro conjunto?
  - Las listas pueden tener elementos repetidos, pero eso no tiene sentido con conjuntos.

Vamos a usar `[Integer]` para representar conjuntos, pero dejando claro que hablamos de conjuntos (sin orden ni repetidos). Para eso podemos hacer un renombre de tipos.

Conjuntos

Definición de tipo usando `type`

Definamos un renombre de tipos para conjuntos: `type Set a = [a]`

- Otra forma de escribir lo mismo, pero más descriptivo.
- `type` es la palabra reservada del lenguaje, `Set` es el nombre que le pusimos nosotros.
- Si bien internamente es una lista, la idea es tratar a `Set a` como si fuera conjunto (es un contrato entre programadores).
- Si nuestra función recibe un conjunto, **vamos a suponer** que no contiene elementos repetidos. (Haskell no hace nada para verificarlo.)
- Si nuestra función devuelve un conjunto, **debemos asegurar** que no contiene elementos repetidos. (Haskell tampoco hace nada automático.)
- Además, no hace falta preocuparse por el orden de los elementos. (Haskell no lo sabe.)

Conjuntos

Definición de tipo usando `type`

Definamos un renombre de tipos para conjuntos: `type Set a = [a]`

- Otra forma de escribir lo mismo, pero más descriptivo.
- `type` es la palabra reservada del lenguaje, `Set` es el nombre que le pusimos nosotros.
- Si bien internamente es una lista, la idea es tratar a `Set a` como si fuera conjunto (es un contrato entre programadores).
- Si nuestra función recibe un conjunto, **vamos a suponer** que no contiene elementos repetidos. (Haskell no hace nada para verificarlo.)
- Si nuestra función devuelve un conjunto, **debemos asegurar** que no contiene elementos repetidos. (Haskell tampoco hace nada automático.)
- Además, no hace falta preocuparse por el orden de los elementos. (Haskell no lo sabe.)

Ejercicios entre todos

- Definir `vacio :: Set Integer` que represente el conjunto vacío
- Implementar entre todos la función
  `agregar :: Integer -> Set Integer -> Set Integer`
  que dado un entero y un conjunto agrega el primero al segundo (ayuda: La función "pertenece" en Haskell existe y se llama "elem")

Ejercicios simples

1. Implementar una función `incluido :: Set Integer -> Set Integer -> Bool` que determina si el primer conjunto está incluido en el segundo.
2. Implementar una función `iguales :: Set Integer -> Set Integer -> Bool` que determina si dos conjuntos son iguales.

```
Ejemplo> iguales [1,2,3,4,5] [2,3,1,4,5]
True
```

3. Implementar una función `agregarC :: Set Integer -> Set (Set Integer) -> Set (Set Integer)` que dado un conjunto de enteros y un conjunto de conjunto de enteros agrega el primero al segundo.

```
Ejemplo> agregarC [1,2] [[4,5],[2,3,1],[2,1]]
[[4,5],[2,3,1],[2,1]
```

Partes de un conjunto

Ejercicios

1. Implementar la función `agregarATodos :: Integer -> Set (Set Integer) -> Set (Set Integer)` que dado un número n y un conjunto de conjuntos cls agrega a n en cada conjunto de cls.
2. Implementar una función `partes :: Integer -> Set (Set Integer)` que genere todos los subconjuntos del conjunto $\{1, 2, 3, \ldots, n\}$.

```
Ejemplo> partes 2
[[], [1], [2], [1, 2]]
```