Algoritmos y Estructuras de Datos II
Introducción

Juan Manuel Rabasedas

11/03/2015

basado en las transparencias de Mauro Jaskelioff

- Qué es una estructura de datos?

- Qué es una estructura de datos?
- Una estructura de datos queda definida si damos:
  - El conjunto de valores que puede tomar,

- Qué es una estructura de datos?
- Una estructura de datos queda definida si damos:
  - El conjunto de valores que puede tomar,
  - Un conjunto de operaciones definidas sobre estos valores,

- Qué es una estructura de datos?
- Una estructura de datos queda definida si damos:
  - El conjunto de valores que puede tomar,
  - Un conjunto de operaciones definidas sobre estos valores,
  - Un conjunto de propiedades que relacionan todo lo anterior.

- ¿Qué nos interesa saber?

AyED en Lenguaje Funcional

- ¿Qué nos interesa saber?
- ¿Por qué usamos un lenguaje funcional?

AyED en Lenguaje Funcional

- ¿Qué nos interesa saber?
- ¿Por qué usamos un lenguaje funcional?

Ejemplo

```
foo( a, lo, hi ) int a[], hi, lo;{'
 int h, l, p, t;'
 if (lo < hi) {'
 l = lo;'
 h = hi;'
 p = a[hi];'
 do{'
   while ((l < h) && (a[l] <= p))'
   l = l+1;'
   while ((h > l) && (a[h] >= p))'
   h = h-1;'
```

Ejemplo

```
    if (l < h) {'
      t = a[l];'
      a[l] = a[h];'
      a[h] = t;'
      }'
   }while (l < h);'
   t = a[l];'
   a[l] = a[hi];'
   a[hi] = t;'
   foo( a, lo, l-1 );'
   foo( a, l+1, hi );'
   }'
 }'
```

Ejemplo

$$
\begin{array}{l}
foo\ [\,] = [\,] \\
foo\ (x : xs) = foo\ menor + \!\!+ [x] + \!\!+ foo\ mayor \\
\quad \textbf{where} \\
\qquad menor = [y \mid y \leftarrow xs, y < x] \\
\qquad mayor = [y \mid y \leftarrow xs, y \geqslant x]
\end{array}
$$

Pseudocódigo Funcional

- Vamos a usar un pseudocódigo funcional y vamos a implementar en ML

Pseudocódigo Funcional

- Vamos a usar un pseudocódigo funcional y vamos a implementar en ML
- La aplicación de funciones se denota con un espacio y asocia a la izquierda.

Matemática	Seudocodigo
$f(x)$	$f\ x$
$f(x, x)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x * g\ y$

- La aplicación tiene mayor precedencia que cualquier otro operador: $f\ x + y = (f\ x) + y$

Nombres y comentarios

- Las funciones y sus argumentos deben empezar con minúscula, y pueden ser seguidos por cero o más letras (mayúsculas o minúsculas), dígitos, guiones bajos, y apóstrofes.
- Las palabras reservadas son: **case data default do else if in let newtype of then type where**
- En una serie de definiciones, cada definición debe empezar en la misma columna.

$$
\begin{aligned}
&a = b + c\\
&\qquad\quad \textbf{where}\\
&\qquad\qquad b = 1\\
&\qquad\qquad c = 2\\
&d = a + 2
\end{aligned}
$$

Operadores infijos

- Los operadores infijos son funciones como cualquier otra.
- Una función se puede hacer infija con backquotes:

$$10 \text{ `}div\text{` } 4 = div\ 10\ 4$$

- Se pueden definir operadores infijos usando alguno de los símbolos disponibles:

$$a ** b = (a * b) + (a + 1) * (b - 1)$$

Tipos

- Un tipo es un nombre para una colección de valores
  - Ej: Bool contiene los valores True y False.
  - Escribimos True :: Bool y False :: Bool.
- En general, si una expresión e tiene tipo t escribimos

$$e :: t$$

- **en nuestro pseudocódigo, toda expresión válida tiene un tipo**
- Si no es posible encontrar un tipo (por ejemplo (True $+ 4$)) la expresión es incorrecta.

Tipos básicos

Algunos tipos básicos de vamos a considerar son:

- Bool , booleanos
- Char , caracteres
- Int, enteros de precisión fija.
- Integer, enteros de precisión arbitraria.
- Float, números de punto flotante de precisión simple.

Listas

- Una lista es una secuencia de valores del mismo tipo
  - $[\text{True}, \text{True}, \text{False}, \text{True}] :: [\text{Bool}]$
  - $[\texttt{'h'}, \texttt{'o'}, \texttt{'l'}, \texttt{'a'},] :: [\text{Char}]$
- en general, $[t]$ es una lista con elementos de tipo t
- t, puede ser cualquier tipo válido.
- No hay restricción con respecto a la longitud de las listas.

Tuplas

- Una secuencia es una secuencia finita de valores de tipos (posiblemente) distintos.
  - (True, True) :: (Bool, Bool)
  - (True, `'a'`, `'b'`) :: (Bool, Char, Char)
- En general, $(t_1, t_2, \ldots ; t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i , para i en $1 \ldots n$.
- A diferencia de las listas, las tuplas tienen explicitado en su tipo la cantidad de elementos que almacenan.
- Los tipos de las tuplas no tiene restricciones.
  (`'a'`; (True; `'c'`)) :: (Char; (Bool; Char))

Funciones

- Una función mapea valores de un tipo en valores de otro:
  - $not :: \mathsf{Bool} \rightarrow \mathsf{Bool}$
  - $isDigit :: \mathsf{Char} \rightarrow \mathsf{Bool}$
- En general, Un tipo $t_1 \rightarrow t_2$ mapea valores de t_1 en valores de t_2.
- Se pueden escribir funciones con múltiples argumentos o resultados usando tuplas y listas.

$$
\begin{array}{l}
add :: (\mathsf{Int}; \mathsf{Int}) \rightarrow \mathsf{Int} \\
add\ (x; y) = x + y \\
deceroa :: \mathsf{Int} \rightarrow [\mathsf{Int}] \\
deceroa\ n = [0 \mathinner{..} n]
\end{array}
$$

Currificación y aplicación parcial

- Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$add' :: \mathsf{Int} \longrightarrow (\mathsf{Int} \longrightarrow \mathsf{Int})$$
$$add'\ x\ y = x + y$$

- A diferencia de add, add' toma los argumentos de a uno por vez. Se dice que add' está currificada.

Currificación y aplicación parcial

- Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$
\begin{aligned}
&add' :: \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int}) \\
&add'\ x\ y = x + y
\end{aligned}
$$

- A diferencia de add, add' toma los argumentos de a uno por vez. Se dice que add' está currificada.
- La ventaja de la versión currificada es que permite la aplicación parcial:

$$
\begin{aligned}
&suma3 :: \mathsf{Int} \to \mathsf{Int} \\
&suma3 = add'\ 3
\end{aligned}
$$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: \mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}))$$
$$mult\ x\ y\ z = x * y * z$$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$\begin{aligned} &mult :: \mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int})) \\ &mult\ x\ y\ z = x * y * z \end{aligned}$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha. $mult :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: \mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha. $mult :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$
- Notar que esta convención es consistente con la aplicación asociando a la izquierda.

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: \mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow (\mathsf{Int} \rightarrow \mathsf{Int}))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha. $mult :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$
- Notar que esta convención es consistente con la aplicación asociando a la izquierda.
- En nuestro pseudocodigo por omisión todas las funciones están currificadas.

Nombres de los tipos

- A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.

Nombres de los tipos

- A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.
- El espacio de nombres de los tipos está completamente separado del espacio de nombres de las expresiones.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [\,a\,] \rightarrow \mathsf{Int}$ Para cualquier tipo a la función $length$ es la misma.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$ Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$ Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$ Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

$$length\ [\text{False}; \text{True}] \leftarrow a = \text{Bool}$$
$$length\ [\texttt{'a'}; \texttt{'b'}] \leftarrow a = \text{Char}$$

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [\,a\,] \rightarrow \mathsf{Int}$ Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

$$length\ [\mathsf{False}; \mathsf{True}] \leftarrow a = \mathsf{Bool}$$
$$length\ [\,\texttt{'a'}; \texttt{'b'}\,] \leftarrow a = \mathsf{Char}$$

- A veces se llama polimorfismo paramétrico a este tipo de polimorfismo.

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$
\begin{aligned}
&abs :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&abs\ n = \textbf{if}\ n > 0\ \textbf{then}\ n\ \textbf{else} - n
\end{aligned}
$$

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$abs :: \mathsf{Int} \rightarrow \mathsf{Int}$$
$$abs\ n = \textbf{if}\ n > 0\ \textbf{then}\ n\ \textbf{else} - n$$

- Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo.

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$abs :: \mathsf{Int} \rightarrow \mathsf{Int}$$
$$abs\ n = \textbf{if}\ n > 0\ \textbf{then}\ n\ \textbf{else}\ -n$$

- Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo.
- Las expresiones condicionales siempre deben tener la rama **else**

- Las funciones pueden ser definidas usando expresiones condicionales

$$
\begin{array}{l}
abs :: \mathsf{Int} \rightarrow \mathsf{Int} \\
abs\ n = \textbf{if}\ n > 0\ \textbf{then}\ n\ \textbf{else} - n
\end{array}
$$

- Para que la expresión condicional tenga sentido, ambas ramas de la misma deben tener el mismo tipo.
- Las expresiones condicionales siempre deben tener la rama **else**
- Por lo tanto no hay ambigüedades en caso de anidamiento:

$$
\begin{array}{l}
signum :: \mathsf{Int} \rightarrow \mathsf{Int} \\
signum\ n = \textbf{if}\ n < 0\ \textbf{then} - 1\ \textbf{else} \\
\qquad\qquad\qquad \textbf{if}\ n \equiv 0\ \textbf{then}\ 0\ \textbf{else}\ 1
\end{array}
$$

Guardas

- Una alternativa a los condicionales es el uso de ecuaciones con guardas.

$$\begin{aligned} abs\ n \mid n > 0 &= n \\ \mid otherwise &= -n \end{aligned}$$

Guardas

- Una alternativa a los condicionales es el uso de ecuaciones con guardas.

$$
\begin{aligned}
abs\ n \mid n > 0 &= n \\
\mid otherwise &= -n
\end{aligned}
$$

- Se usan para hacer ciertas definiciones más fáciles de leer.

$$
\begin{aligned}
signum\ n \mid n < 0 &= \ -1 \\
\mid n \equiv 0 &= 0 \\
\mid otherwise &= 1
\end{aligned}
$$

Guardas

- Una alternativa a los condicionales es el uso de ecuaciones con guardas.

$$\begin{aligned} abs\ n \mid n > 0 \quad &= n \\ \mid otherwise &= -n \end{aligned}$$

- Se usan para hacer ciertas definiciones más fáciles de leer.

$$\begin{aligned} signum\ n \mid n < 0 &= \ -1 \\ \mid n \equiv 0 &= 0 \\ \mid otherwise &= 1 \end{aligned}$$

- La condición *otherwise* se define como

$$otherwise = \text{True}$$

Pattern Matching

- Muchas funciones se definen más claramente usando pattern matching.

$$
\begin{aligned}
&not :: \mathsf{Bool} \rightarrow \mathsf{Bool} \\
&not\ \mathsf{False} = \mathsf{True} \\
&not\ \mathsf{True}\ \ = \mathsf{False}
\end{aligned}
$$

Pattern Matching

- Muchas funciones se definen más claramente usando pattern matching.

$$
\begin{aligned}
&\mathit{not} :: \mathsf{Bool} \rightarrow \mathsf{Bool} \\
&\mathit{not}\ \mathsf{False} = \mathsf{True} \\
&\mathit{not}\ \mathsf{True}\ = \mathsf{False}
\end{aligned}
$$

- Los patrones se componen de constructores de datos y variables (salvo los patrones 0 y n + 1) .
- Una variable es un patrón que nunca falla.

$$
\begin{aligned}
&\mathit{succ} :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&\mathit{succ}\ n = n + 1
\end{aligned}
$$

Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

$$
\begin{aligned}
&(\wedge) :: \mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}\\
&\mathsf{True} \;\wedge\; \mathsf{True} \;=\; \mathsf{True}\\
&\mathsf{True} \;\wedge\; \mathsf{False} = \mathsf{False}\\
&\mathsf{False} \wedge \mathsf{True} \;=\; \mathsf{False}\\
&\mathsf{False} \wedge \mathsf{False} = \mathsf{False}
\end{aligned}
$$

- Usando el ingenio se pueden obtener definiciones concisas.

$$
\begin{aligned}
&(\wedge) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{True} \ \wedge \mathsf{True} \ = \mathsf{True}\\
&\mathsf{True} \ \wedge \mathsf{False} = \mathsf{False}\\
&\mathsf{False} \wedge \mathsf{True} \ = \mathsf{False}\\
&\mathsf{False} \wedge \mathsf{False} = \mathsf{False}
\end{aligned}
$$

  puede ser escrita en forma compacta como

$$
\begin{aligned}
&(\wedge) :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}\\
&\mathsf{True} \wedge \mathsf{True} = \mathsf{True}\\
&\quad \_ \wedge \ \_ \qquad = \mathsf{False}
\end{aligned}
$$

- Notar la importancia del orden de las ecuaciones.

Patrones de tuplas

- Una tupla de patrones es un patrón.

$$
\begin{aligned}
&fst :: (a, b) \rightarrow a \\
&fst\ (x, \_) = x \\
&snd :: (a, b) \rightarrow b \\
&snd\ (\_, y) = y
\end{aligned}
$$

Patrones de tuplas

- Una tupla de patrones es un patrón.

$$
\begin{aligned}
&fst :: (a, b) \to a \\
&fst\ (x, \_) = x \\
&snd :: (a, b) \to b \\
&snd\ (\_, y) = y
\end{aligned}
$$

- ¿Qué hace la siguiente función? $f\ (x, (y, z)) = ((x, y), z)$

Patrones de tuplas

- Una tupla de patrones es un patrón.

$$
\begin{aligned}
& fst :: (a, b) \rightarrow a \\
& fst\ (x, \_) = x \\
& snd :: (a, b) \rightarrow b \\
& snd\ (\_, y) = y
\end{aligned}
$$

- ¿Qué hace la siguiente función? $f\ (x, (y, z)) = ((x, y), z)$
- En general, los patrones pueden anidarse

Patrones de Listas

- Toda lista (no vacía) se contruye usando el operador (:) (llamado $cons$) que agrega un elemento al principio de la lista.

$$[1, 2, 3, 4] \equiv 1 : (2 : (3 : (4 : [\,])))$$

Patrones de Listas

- Toda lista (no vacía) se contruye usando el operador $(:)$ (llamado $cons$) que agrega un elemento al principio de la lista.

$$[1,2,3,4] \equiv 1 : (2 : (3 : (4 : [\,])))$$

- Por lo tanto, puedo definir funciones usando el patrón $(x : xs)$

$$\begin{array}{l} head :: [\,a\,] \rightarrow a \\ head\ (x : \_) = x \\ tail :: [\,a\,] \rightarrow [\,a\,] \\ tail\ (\_ : xs) = xs \end{array}$$

Patrones de Listas

- Toda lista (no vacía) se contruye usando el operador $(:)$ (llamado $cons$) que agrega un elemento al principio de la lista.

$$[1, 2, 3, 4] \equiv 1 : (2 : (3 : (4 : [\,])))$$

- Por lo tanto, puedo definir funciones usando el patrón $(x : xs)$

$$\begin{array}{l} head :: [\,a\,] \rightarrow a \\ head\ (x : \_) = x \\ tail :: [\,a\,] \rightarrow [\,a\,] \\ tail\ (\_ : xs) = xs \end{array}$$

- $(x : xs)$ sólo matchea el caso de listas no vacías $head\ [\,]$ **Error!**

Secciones

- Un operador infijo, puede ser escrito en forma prefija usando paréntesis:

$$> (+)\ 1\ 2$$
$$3$$

Secciones

- Un operador infijo, puede ser escrito en forma prefija usando paréntesis:

$$> (+)\ 1\ 2$$
$$3$$

- También uno de los argumentos puede ser incluído en los paréntesis

$$> (1+)\ 2$$
$$3$$
$$> (+2)\ 1$$
$$3$$

- Un operador infijo, puede ser escrito en forma prefija usando paréntesis:

$$> (+)\ 1\ 2$$
$$3$$

- También uno de los argumentos puede ser incluído en los paréntesis

$$> (1+)\ 2$$
$$3$$
$$> (+2)\ 1$$
$$3$$

- En general, dado un operador $\oplus$, entonces las funciones de la forma $(\oplus)$, $(x\oplus)$, $(\oplus y)$ son llamadas secciones.

- En matemáticas, una manera de construir conjuntos a partir de conjuntos existentes es con la notación por comprensión

$$\{x^2 | x \in \{1 \dots 5\}\}$$

describe el conjunto $\{1, 4, 9, 16, 25\}$ o (lo que es lo mismo) el conjunto de todos los números x^2 tal que x sea un elemento del conjunto $\{1 \dots 5\}$

Listas por comprensión

- En Haskell, una manera de construir listas a partir de listas existentes es con la notación por comprensión

$$[x \uparrow 2 \mid x \leftarrow [1\,..\,5]]$$

describe la lista $[1, 4, 9, 16, 25]$ o (lo que es lo mismo) la lista de todos los números $x \uparrow 2$ tal que x sea un elemento de la lista $[1\,..\,5]$

- La expresión $x \leftarrow [1\,..\,5]$ es un *generador*, ya que dice como se generan los valores de x.

- La expresión $x \leftarrow [1\,..\,5]$ es un *generador*, ya que dice como se generan los valores de x.
- Una lista por comprensión puede tener varios generadores, separados por coma.

$$> [(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]]$$
$$[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]$$

- La expresión $x \leftarrow [1\,..\,5]$ es un *generador*, ya que dice como se generan los valores de x.
- Una lista por comprensión puede tener varios generadores, separados por coma.

$$> [(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$$
$$[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$$

- ¿Qué pasa cuando cambiamos el orden de los generadores?

$$> [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$$

- La expresión $x \leftarrow [1\,..\,5]$ es un *generador*, ya que dice como se generan los valores de x.
- Una lista por comprensión puede tener varios generadores, separados por coma.

$$> [\,(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$$
$$[\,(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)\,]$$

- ¿Qué pasa cuando cambiamos el orden de los generadores?

$$> [\,(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$$

- Los generadores posteriores cambian más rápidamente.

Generadores dependientes

- Un generador puede depender de un generador anterior

$$[(x, y) \mid x \leftarrow [1\,..\,3], y \leftarrow [x\,..\,3]]$$

Generadores dependientes

- Un generador puede depender de un generador anterior

  $$[(x, y) \mid x \leftarrow [1 \mathinner{..} 3], y \leftarrow [x \mathinner{..} 3]]$$

  Esto es la lista de todos los pares (x, y) tal que x, y están en $[1 \mathinner{..} 3]$ e $y \geqslant x$.

Generadores dependientes

- Un generador puede depender de un generador anterior

$$[(x, y) \mid x \leftarrow [1 \mathinner{\ldotp\ldotp} 3], y \leftarrow [x \mathinner{\ldotp\ldotp} 3]]$$

  Esto es la lista de todos los pares (x, y) tal que x, y están en $[1 \mathinner{\ldotp\ldotp} 3]$ e $y \geqslant x$.

- ¿Qué hace la siguiente función?

$$\begin{aligned} &concat && :: [[a]] \rightarrow [a] \\ &concat\ xss && = [x \mid xs \leftarrow xss, x \leftarrow xs] \end{aligned}$$

Generadores dependientes

- Un generador puede depender de un generador anterior

$$[(x, y) \mid x \leftarrow [1\,..\,3], y \leftarrow [x\,..\,3]]$$

Esto es la lista de todos los pares (x, y) tal que x, y están en $[1\,..\,3]$ e $y \geqslant x$.

- ¿Qué hace la siguiente función?

$$\begin{array}{ll} concat & :: [[a]] \rightarrow [a] \\ concat\ xss & = [x \mid xs \leftarrow xss, x \leftarrow xs] \end{array}$$

$$> concat\ [[1, 2, 3], [4, 5], [6]]$$
$$[1, 2, 3, 4, 5, 6]$$

Guardas

- Las listas por comprensión pueden usar guardas para restringir los valores producidos por generadores anteriores

$$[\,x \mid x \leftarrow [1\,..\,10], even\ x\,]$$

Guardas

- Las listas por comprensión pueden usar guardas para restringir los valores producidos por generadores anteriores

$$[x \mid x \leftarrow [1\,..\,10], \mathit{even}\ x]$$

- ¿Qué hace la siguiente función?

$$\begin{aligned} &\mathit{factors} &&:: \mathsf{Int} \rightarrow [\mathsf{Int}] \\ &\mathit{factors}\ n &&= [x \mid x \leftarrow [1\,..\,n], n\ \text{`}\mathit{mod}\text{`}\ x \equiv 0] \end{aligned}$$

Guardas

- Las listas por comprensión pueden usar guardas para restringir los valores producidos por generadores anteriores

$$[x \mid x \leftarrow [1 \mathinner{..} 10], even\ x]$$

- ¿Qué hace la siguiente función?

$$\begin{aligned} &factors \quad :: \mathsf{Int} \rightarrow [\mathsf{Int}] \\ &factors\ n = [x \mid x \leftarrow [1 \mathinner{..} n], n\ `mod`\ x \equiv 0] \end{aligned}$$

- Como un número n es primo iff sus únicos factores son 1 y n, podemos definir

$$\begin{aligned} &prime :: \mathsf{Int} \rightarrow \mathsf{Bool} \\ &prime\ n = factors\ n \equiv [1, n] \end{aligned}$$

$$\begin{aligned} &primes :: \mathsf{Int} \rightarrow [\mathsf{Int}] \\ &primes\ n = [x \mid x \leftarrow [2 \mathinner{..} n], prime\ x] \end{aligned}$$

Cadenas

- Una String es una lista de caracteres.
- `"Hola"` :: String
- `"Hola"` $=$ [`'H'`, `'o'`, `'l'`, `'a'`]
- Todas las funciones sobre listas son aplicables a String, y las listas por comprensión pueden ser aplicadas a Strings.

$$\begin{aligned} &cantminusc && :: \mathsf{String} \rightarrow \mathsf{Int} \\ &cantminusc\ xs && = length\ [x \mid x \leftarrow xs, isLower\ x] \end{aligned}$$

Zip

- La función zip, mapea dos listas a una lista con los pares de elementos correspondientes

$$zip :: [\,a\,] \rightarrow [\,b\,] \rightarrow [\,(a, b)\,]$$

$$> zip\ [\,\texttt{'a'}, \texttt{'b'}, \texttt{'c'}\,]\ [1, 2, 3, 4]$$

$$[\,(\texttt{'a'}, 1), (\texttt{'b'}, 2), (\texttt{'c'}, 3)\,]$$

Zip

- La función zip, mapea dos listas a una lista con los pares de elementos correspondientes

$$zip :: [a] \to [b] \to [(a, b)]$$

$$> zip \; [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}] \; [1, 2, 3, 4]$$
$$[(\texttt{'a'}, 1), (\texttt{'b'}, 2), (\texttt{'c'}, 3)]$$

- Ejemplo: Lista de pares de elementos adyacentes:

$$\begin{aligned} &pairs &&:: [a] \to [(a, a)] \\ &pairs \; xs &&= zip \; xs \; (tail \; xs) \end{aligned}$$

Zip

- La función zip, mapea dos listas a una lista con los pares de elementos correspondientes

$$
\begin{array}{l}
zip :: [a] \to [b] \to [(a,b)] \\
> zip\ [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}]\ [1,2,3,4] \\
[(\texttt{'a'},1),(\texttt{'b'},2),(\texttt{'c'},3)]
\end{array}
$$

- Ejemplo: Lista de pares de elementos adyacentes:

$$
\begin{array}{ll}
pairs & :: [a] \to [(a,a)] \\
pairs\ xs & = zip\ xs\ (tail\ xs)
\end{array}
$$

- ¿Está una lista ordenada?

$$
\begin{array}{ll}
sorted & :: \mathsf{Ord}\ a \Rightarrow [a] \to \mathsf{Bool} \\
sorted\ xs & = and\ [x \leqslant y \mid (x,y) \leftarrow pairs\ xs]
\end{array}
$$

Ejemplo *zip*: pares índice/valor

- Podemos usar *zip* para generar índices

$$
\begin{aligned}
&\mathit{rangeof} && :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow [\,a\,] \rightarrow [\,a\,] \\
&\mathit{rangeof}\ \mathit{low}\ \mathit{hi}\ \mathit{xs} && = [\,x \mid (x, i) \leftarrow \mathit{zip}\ \mathit{xs}\ [0\,..],\\
& && \qquad\qquad i \geqslant \mathit{low},\\
& && \qquad\qquad i \leqslant \mathit{hi}\,]
\end{aligned}
$$

Ejemplo zip: pares índice/valor

- Podemos usar zip para generar índices

$$
\begin{aligned}
&rangeof && :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow [\,a\,] \rightarrow [\,a\,] \\
&rangeof\ low\ hi\ xs && = [x \mid (x, i) \leftarrow zip\ xs\ [0\,..],\\
& && \qquad\qquad i \geqslant low,\\
& && \qquad\qquad i \leqslant hi]
\end{aligned}
$$

$$> [x \uparrow 2 \mid x \leftarrow [1\,..\,10]]$$
$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$
$$> rangeof\ 3\ 7\ [x \uparrow 2 \mid x \leftarrow [1\,..\,10]]$$
$$[16, 25, 36, 49, 64]$$

Recursión

- En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

Recursión

- En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

$$
\begin{aligned}
&factorial &&:: \mathsf{Int} \rightarrow \mathsf{Int}\\
&factorial\ 0 &&= 1\\
&factorial\ n &&= n * factorial\ (n-1)
\end{aligned}
$$

Recursión

- En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

$$\begin{aligned} &factorial && :: \mathsf{Int} \rightarrow \mathsf{Int} \\ &factorial\ 0 && = 1 \\ &factorial\ n && = n * factorial\ (n-1) \end{aligned}$$

- ¿Qué sucede con $factorial\ n$ cuando $n < 0$?

Recursión

- En los lenguajes funcionales, la recursión es uno de los principales recursos para definir funciones.

$$\begin{aligned}
&factorial && :: \mathsf{Int} \rightarrow \mathsf{Int} \\
&factorial\ 0 && = 1 \\
&factorial\ n && = n * factorial\ (n - 1)
\end{aligned}$$

- ¿Qué sucede con $factorial\ n$ cuando $n < 0$?
- Recursión sobre listas

$$\begin{aligned}
&length && :: [\,a\,] \rightarrow \mathsf{Int} \\
&length\ [\,] && = 0 \\
&length\ (x : xs) && = 1 + length\ xs
\end{aligned}$$

Recursión Mutua

- No hace falta ninguna sintaxis especial para la recursión mutua.

$$
\begin{aligned}
&zigzag :: [(a, a)] \rightarrow [a] \\
&zigzag = zig \\
&zig\ [\,] && = [\,] \\
&zig\ ((x, \_) : xs) && = x : zag\ xs \\
&zag\ [\,] && = [\,] \\
&zag\ ((\_, y) : xs) && = y : zig\ xs
\end{aligned}
$$

Recursión Mutua

- No hace falta ninguna sintaxis especial para la recursión mutua.

$$
\begin{aligned}
&zigzag :: [(a, a)] \to [a] \\
&zigzag = zig \\
&zig\ [\,] && = [\,] \\
&zig\ ((x, \_) : xs) && = x : zag\ xs \\
&zag\ [\,] && = [\,] \\
&zag\ ((\_, y) : xs) && = y : zig\ xs
\end{aligned}
$$

- $> zigzag\ [(1,2),(3,4),(5,6),(7,8)]$
  $[1,4,5,8]$

Quicksort

- El algoritmo de ordenación Quicksort:
  - La lista vacía está ordenada
  - Las listas no vacías pueden ser ordenadas, ordenando los valores de la cola $\leqslant$ que la cabeza, ordenando los valores $>$ que la cabeza y rearmando el resultado con las listas resultantes a ambos lados de la cabeza.

Quicksort

- El algoritmo de ordenación Quicksort:
  - La lista vacía está ordenada
  - Las listas no vacías pueden ser ordenadas, ordenando los valores de la cola $\leqslant$ que la cabeza, ordenando los valores $>$ que la cabeza y rearmando el resultado con las listas resultantes a ambos lados de la cabeza.
- Su implementación:

$$
\begin{array}{lll}
qsort & :: & \mathsf{Ord}\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
qsort\ [\,] & = & [\,] \\
qsort\ (x : xs) & = & qsort\ chicos \mathbin{+\!\!+} [\,x\,] \mathbin{+\!\!+} qsort\ grandes \\
\quad \textbf{where}\ chicos & = & [\,a \mid a \leftarrow xs, a \leqslant x\,] \\
\qquad\qquad grandes & = & [\,b \mid b \leftarrow xs, b > x\,]
\end{array}
$$

Sinónimos de tipos

- Vamos a definir un nuevo nombre para un tipo existente usando una declaración **type**.

  $$\textbf{type}\ \mathsf{String} = [\mathsf{Char}]$$

  String es un sinónimo del tipo [Char].

Sinónimos de tipos

- Vamos a definir un nuevo nombre para un tipo existente usando una declaración **type**.

  type String = [Char]

  String es un sinónimo del tipo [Char].

- Los sinónimos de tipo hace que ciertas declaraciones de tipos sean más fáciles de leer.

  $$\begin{array}{ll}
  \textbf{type}\ \mathsf{Pos} & = (\mathsf{Int}, \mathsf{Int}) \\
  origen & :: \mathsf{Pos} \\
  origen & = (0, 0) \\
  izq & :: \mathsf{Pos} \rightarrow \mathsf{Pos} \\
  izq\ (x, y) & = (x - 1, y)
  \end{array}$$

- Los sinónimos de tipo pueden tener parámetros

$$\textbf{type}\ \mathsf{Par}\ a = (a, a)$$

$$\begin{aligned} &copiar\ \ :: a \rightarrow \mathsf{Par}\ a \\ &copiar\ x = (x, x) \end{aligned}$$

Sinónimos de Tipos

- Los sinónimos de tipo pueden tener parámetros

$$\textbf{type}\ \mathsf{Par}\ a = (a, a)$$
$$copiar\ :: a \to \mathsf{Par}\ a$$
$$copiar\ x = (x, x)$$

- Los sinónimos de tipo pueden anidarse

$$\textbf{type}\ \mathsf{Punto} = (\mathsf{Int}, \mathsf{Int})$$
$$\textbf{type}\ \mathsf{Trans} = \mathsf{Punto} \to \mathsf{Punto}$$

- Los sinónimos de tipo pueden tener parámetros

$$\textbf{type}\ \mathsf{Par}\ a = (a, a)$$

$$\begin{aligned} &copiar &&:: a \to \mathsf{Par}\ a \\ &copiar\ x = (x, x) \end{aligned}$$

- Los sinónimos de tipo pueden anidarse

$$\begin{aligned} &\textbf{type}\ \mathsf{Punto} &&= (\mathsf{Int}, \mathsf{Int}) \\ &\textbf{type}\ \mathsf{Trans} &&= \mathsf{Punto} \to \mathsf{Punto} \end{aligned}$$

pero no pueden ser recursivos

$$\textbf{type}\ \mathsf{Tree} = (\mathsf{Int}, [\mathsf{Tree}])$$

Declaraciones **data**

- los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

  data Bool = False | True

  declara un nuevo tipo Bool con dos nuevos valores False y True.

Declaraciones **data**

- los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

  data Bool = False | True

  declara un nuevo tipo Bool con dos nuevos valores False y True.
- True y False son los *constructores* del tipo Bool
- Los nombres de los constructores deben empezar con mayúsculas.

Declaraciones **data**

- los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

  data Bool = False | True

  declara un nuevo tipo Bool con dos nuevos valores False y True.
- True y False son los *constructores* del tipo Bool
- Los nombres de los constructores deben empezar con mayúsculas.
- Dos constructores diferentes siempre construyen diferentes valores del tipo.

Usando **data**

- Los valores de un nuevo tipo se usan igual que los predefinidos

$$\textbf{data}\ \text{Respuesta} = \text{Si} \mid \text{No} \mid \text{Desconocida}$$

$$\begin{aligned}
&respuestas :: [\text{Respuesta}]\\
&respuestas = [\text{Si}, \text{No}, \text{Desconocida}]
\end{aligned}$$

$$\begin{aligned}
&invertir && :: \text{Respuesta} \to \text{Respuesta}\\
&invertir\ \text{Si} && = \text{No}\\
&invertir\ \text{No} && = \text{Si}\\
&invertir\ \text{Desconocida} && = \text{Desconocida}
\end{aligned}$$

- Ejemplo en que los constructores tienen parámetros

$$\mathbf{data}\ \mathsf{Shape} = \mathsf{Circle\ Float} \mid \mathsf{Rect\ Float\ Float}$$

$$\begin{aligned} &square \quad :: \mathsf{Float} \to \mathsf{Shape} \\ &square\ n = \mathsf{Rect}\ n\ n \end{aligned}$$

$$\begin{aligned} &area &&:: \mathsf{Shape} \to \mathsf{Float} \\ &area\ (\mathsf{Circle}\ r) &&= \pi * r \uparrow 2 \\ &area\ (\mathsf{Rect}\ x\ y) &&= x * y \end{aligned}$$

Usando **data**

- Ejemplo en que los constructores tienen parámetros

$$\textbf{data}\ \mathsf{Shape} = \mathsf{Circle}\ \mathsf{Float} \mid \mathsf{Rect}\ \mathsf{Float}\ \mathsf{Float}$$

$$\begin{aligned} &square :: \mathsf{Float} \to \mathsf{Shape} \\ &square\ n = \mathsf{Rect}\ n\ n \end{aligned}$$

$$\begin{aligned} &area :: \mathsf{Shape} \to \mathsf{Float} \\ &area\ (\mathsf{Circle}\ r) = \pi * r \uparrow 2 \\ &area\ (\mathsf{Rect}\ x\ y) = x * y \end{aligned}$$

- Los constructores son funciones

$$\begin{aligned} &> :t\ \mathsf{Circle} \\ &\mathsf{Circle} :: \mathsf{Float} \to \mathsf{Shape} \\ &> :t\ \mathsf{Rect} \\ &\mathsf{Rect} :: \mathsf{Float} \to \mathsf{Float} \to \mathsf{Shape} \end{aligned}$$

- Las declaraciones **data** pueden tener parámetros de tipos.

$$\textbf{data}\ \mathsf{Maybe}\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$$

$$\begin{aligned}
&safehead :: [\,a\,] \rightarrow \mathsf{Maybe}\ a\\
&safehead\ [\,] = \mathsf{Nothing}\\
&safehead\ xs = \mathsf{Just}\ (head\ xs)
\end{aligned}$$

Constructores de Tipos

- Las declaraciones **data** pueden tener parámetros de tipos.

$$\textbf{data}\ \mathsf{Maybe}\ a = \mathsf{Nothing} \mid \mathsf{Just}\ a$$

$$\begin{aligned} &safehead :: [a] \to \mathsf{Maybe}\ a \\ &safehead\ [\,] = \mathsf{Nothing} \\ &safehead\ xs = \mathsf{Just}\ (head\ xs) \end{aligned}$$

- Maybe es un constructor de tipos ya que dado un tipo a, construye el tipo Maybe a.

Tipos Recursivos

- Las declaraciones **data** pueden ser recursivos

$$\textbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat}$$

$$\begin{aligned} add\ n\ \mathsf{Zero} &= n \\ add\ n\ (\mathsf{Succ}\ m) &= \mathsf{Succ}\ (add\ m\ n) \end{aligned}$$

Tipos Recursivos

- Las declaraciones **data** pueden ser recursivos

$$\textbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat}$$

$$\begin{aligned} add\ n\ \mathsf{Zero} &= n \\ add\ n\ (\mathsf{Succ}\ m) &= \mathsf{Succ}\ (add\ m\ n) \end{aligned}$$

- Ejercicio: definir la multiplicación para Nat
- Ejercicio: definir la exponenciación para Nat.

- Además de pattern matching en el lado izq. de una definición, podemos usar una expresión **case**

$$
\begin{array}{l}
esCero :: \mathsf{Nat} \rightarrow \mathsf{Bool} \\
esCero\ n = \textbf{case}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathsf{Zero} \rightarrow \mathsf{True} \\
\qquad\qquad\qquad\qquad \_ \qquad \rightarrow \mathsf{False}
\end{array}
$$

Expresiones **case**

- Además de pattern matching en el lado izq. de una definición, podemos usar una expresión **case**

$$
\begin{aligned}
&esCero :: \mathsf{Nat} \rightarrow \mathsf{Bool} \\
&esCero\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\qquad\qquad \mathsf{Zero} \rightarrow \mathsf{True} \\
&\qquad\qquad\qquad \_ \qquad\ \rightarrow \mathsf{False}
\end{aligned}
$$

- Los patrones de los diferentes casos son intentados en orden

Expresiones **case**

- Además de pattern matching en el lado izq. de una definición, podemos usar una expresión **case**

$$
\begin{array}{l}
esCero :: \mathsf{Nat} \rightarrow \mathsf{Bool} \\
esCero\ n = \textbf{case}\ n\ \textbf{of} \\
\qquad\qquad\qquad \mathsf{Zero} \rightarrow \mathsf{True} \\
\qquad\qquad\qquad \_ \qquad \rightarrow \mathsf{False}
\end{array}
$$

- Los patrones de los diferentes casos son intentados en orden
- Se usa la indentación para marcar un bloque de casos