Programación Funcional con Haskell

Mauro Jaskelioff

28/03/2022

Hay dos maneras de construir un diseño de software:

Una manera es hacerlo de manera tan simple que sea obvio que no hay deficiencias.

La otra es hacerlo de manera tan complicada que no haya deficiencias obvias.

Tony Hoare, 1980

Programación funcional

- La programación funcional es un estilo de programación moderno.

Programación funcional

- La programación funcional es un estilo de programación moderno.
- No usa un modelo de computación basado en máquinas sino en un lenguaje simple y elegante (el λ-cálculo).

Programación funcional

- La programación funcional es un estilo de programación moderno.
- No usa un modelo de computación basado en máquinas sino en un lenguaje simple y elegante (el λ-cálculo).
- Su simpleza y versatilidad lo hacen ideal para aprender conceptos de
  - programación,
  - lenguajes de programación,
  - computabilidad,
  - semántica,
  - verificación, derivación, testing.

¿Qué es la programación funcional?

Hay diferentes opiniones.

¿Qué es la programación funcional?

Hay diferentes opiniones.

- Es un *estilo* de programación en el cual el método básico de computar es aplicar funciones a argumentos

¿Qué es la programación funcional?

Hay diferentes opiniones.

- Es un *estilo* de programación en el cual el método básico de computar es aplicar funciones a argumentos
- Un lenguaje de programación funcional es un lenguaje que *permite* y *alienta* el uso de un estilo funcional.

Haskell

- Haskell posee varias ventajas.
  - Programas Concisos
  - Sistemas de Tipos Poderoso
  - Funciones Recursivas
  - Fácilidad para probar propiedades de programas.
  - Funciones de Alto Orden
  - Evaluación Perezosa
  - Facilidad para definir DSLs
  - Efectos Monádicos

Haskell

- Haskell posee varias ventajas.
  - Programas Concisos
  - Sistemas de Tipos Poderoso
  - Funciones Recursivas
  - Fácilidad para probar propiedades de programas.
  - Funciones de Alto Orden
  - Evaluación Perezosa
  - Facilidad para definir DSLs
  - Efectos Monádicos
- Haremos un repaso por la sintaxis básica de Haskell.
- También introduciremos algunos conceptos nuevos.

Preludio

- Muchas funciones de uso común están definidas en el Preludio (Prelude.hs)
- Además de las operaciones aritméticas usuales se definen muchas funciones que operan sobre *listas*.
  - *head*, *tail*, (!!), *take*, *drop*, *length*, *sum*, *product*, $(+\!\!+)$, *reverse*
- Leer el código del preludio puede ser muy instructivo (y sorprendente!).

GHC

- Usaremos GHC, un compilador (*ghc*) e intérprete (ghci) de Haskell, que también soporta varias extensiones del mismo.
- URL: `http://www.haskell.org/ghc`

La aplicación de funciones

- en Haskell la aplicación se denota con un espacio y asocia a la izquierda.

Matemáticas	Haskell
$f\ (x)$	$f\ x$
$f\ (x,y)$	$f\ x\ y$
$f\ (g\ (x))$	$f\ (g\ x)$
$f\ (x,(g\ (y))$	$f\ x\ (g\ y)$
$f\ (x)\ g\ (y)$	$f\ x * g\ y$

La aplicación de funciones

- en Haskell la aplicación se denota con un espacio y asocia a la izquierda.

Matemáticas	Haskell
$f\ (x)$	$f\ x$
$f\ (x, y)$	$f\ x\ y$
$f\ (g\ (x))$	$f\ (g\ x)$
$f\ (x, (g\ (y)))$	$f\ x\ (g\ y)$
$f\ (x)\ g\ (y)$	$f\ x * g\ y$

- 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.

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 class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where

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 class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where
- Los comentarios se escriben

```
-- comentario que finaliza junto con la línea

{- bloque de comentarios, útil para comentarios
que no entran en una sola línea. -}
```

Offside Rule

- En una serie de definiciones, cada definición debe empezar en la misma columna.

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

Offside Rule

- En una serie de definiciones, cada definición debe empezar en la misma columna.

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

- Gracias a esta regla no hace falta un sintaxis *explícita* para agrupar definiciones.

Operadores infijos

- Los operadores infijos son funciones como cualquier otra.

Operadores infijos

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

$$10 \; `\mathit{div}` \; 4 = \mathit{div} \; 10 \; 4$$

Operadores infijos

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

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

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

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

Operadores infijos

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

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

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

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

- La asociatividad y precedencia se indica usando **infixr** (asociativad der.), **infixl** (asociatividad izq.), o **infix** (si los paréntesis deben ser obligatorios)

$$\textbf{infixr}\ 6\ (**)$$

Tipos

- Un tipo es un nombre para una colección de valores
  - Ej: *Bool* contiene los valores *True* y *False*.
  - Escribimos $True :: Bool$ y $False :: Bool$.

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

$$e :: t$$

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 Haskell, toda expresión válida tiene un tipo**

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 Haskell, toda expresión válida tiene un tipo**
- El tipo de cada expresión es calculado previo a su evaluación mediante la *inferencia de tipos*.

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 Haskell, toda expresión válida tiene un tipo**
- El tipo de cada expresión es calculado previo a su evaluación mediante la *inferencia de tipos*.
- Si no es posible encontrar un tipo (por ejemplo $(True + 4)$) el compilador/intérprete protestará con un *error de tipo*.

Tipos básicos

Alguno de los tipos básicos de Haskell 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

Listas

- Una lista es una secuencia de valores del *mismo* tipo
  - $[True, True, False, True] :: [Bool]$
  - `['h', 'o', 'l', 'a']` :: $[Char]$

Listas

- Una lista es una secuencia de valores del *mismo* tipo
  - $[\mathit{True}, \mathit{True}, \mathit{False}, \mathit{True}] :: [\mathit{Bool}]$
  - $[\texttt{'h'}, \texttt{'o'}, \texttt{'l'}, \texttt{'a'}] :: [\mathit{Char}]$

- En general, $[t]$ es una lista con elementos de tipo t.

Listas

- Una lista es una secuencia de valores del *mismo* tipo
  - $[\mathit{True}, \mathit{True}, \mathit{False}, \mathit{True}] :: [\mathit{Bool}]$
  - $[\texttt{'h'}, \texttt{'o'}, \texttt{'l'}, \texttt{'a'}] :: [\mathit{Char}]$
- En general, $[t]$ es una lista con elementos de tipo t.
- t, puede ser *cualquier* tipo válido.
  - $[[\texttt{'a'}], [\texttt{'b'}, \texttt{'c'}], []] :: [[\mathit{Char}]]$

Listas

- Una lista es una secuencia de valores del *mismo* tipo
  - $[True, True, False, True] :: [Bool]$
  - `['h','o','l','a']` $:: [Char]$
- En general, $[t]$ es una lista con elementos de tipo t.
- t, puede ser *cualquier* tipo válido.
  - `[['a'],['b','c'],[]]` $:: [[Char]]$
- No hay restricción con respecto a la longitud de las listas.

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True,$ `'a'`, `'b'`$) :: (Bool, Char, Char)$

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True,$ `'a'`, `'b'`$) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 \, . . \, n$.

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True,$ `'a'`, `'b'`$) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 \,..\, n$.
- A diferencia de las listas, las tuplas tienen explicitado en su tipo la *cantidad* de elementos que almacenan.

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True,$ `'a'`, `'b'`$) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 .. n$.
- 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'`$))$

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(\mathit{True}, \mathit{True}) :: (\mathit{Bool}, \mathit{Bool})$
  - $(\mathit{True}, \texttt{'a'}, \texttt{'b'}) :: (\mathit{Bool}, \mathit{Char}, \mathit{Char})$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 \mathinner{\ldotp\ldotp} n$.
- 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
  - $(\texttt{'a'}, (\mathit{True}, \texttt{'c'})) :: (\mathit{Char}, (\mathit{Bool}, \mathit{Char}))$

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True,$ `'a'`, `'b'`$) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 \mathinner{..} n$.
- 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'`$)) :: (Char, (Bool, Char))$
  - (([`'a'`, `'b'`], `'a'`), `'b'`)

Tuplas

- Una tupla es una secuencia finita de valores de tipos (posiblemente) distintos.
  - $(True, True) :: (Bool, Bool)$
  - $(True, \texttt{'a'}, \texttt{'b'}) :: (Bool, Char, Char)$
- En general, $(t_1, t_2, ..., t_n)$ es el tipo de una n-tupla cuyas componente i tiene tipo t_i, para i en $1 \,..\, n$.
- 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
  - $(\texttt{'a'}, (True, \texttt{'c'})) :: (Char, (Bool, Char))$
  - $(([\texttt{'a'}, \texttt{'b'}], \texttt{'a'}), \texttt{'b'}) :: (([Char], Char), Char)$

Funciones

- Una función mapea valores de un tipo en valores de otro
  - $not :: Bool \rightarrow Bool$
  - $isDigit :: Char \rightarrow Bool$

Funciones

- Una función mapea valores de un tipo en valores de otro
  - $not :: Bool \rightarrow Bool$
  - $isDigit :: Char \rightarrow Bool$

- En general, Un tipo función $t_1 \rightarrow t_2$ mapea valores de t_1 en valores de t_2.

Funciones

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

$$
\begin{aligned}
&add && :: (Int, Int) \rightarrow Int \\
&add\ (x, y) && = x + y \\
&deceroa && :: Int \rightarrow [Int] \\
&deceroa\ n && = [0 \mathinner{..} n]
\end{aligned}
$$

Currificación y aplicación parcial

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

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

Currificación y aplicación parcial

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

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

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

Currificación y aplicación parcial

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

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

- 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*:

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

Currificación

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

$$mult :: Int \to (Int \to (Int \to Int))$$
$$mult\ x\ y\ z = x * y * z$$

Currificación

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

$$mult :: Int \to (Int \to (Int \to Int))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención el constructor de tipos $\to$ asocia a la derecha.
  $mult :: Int \to Int \to Int \to Int$

Currificación

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

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

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

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

- Para evitar escribir muchos paréntesis, por convención el constructor de tipos $\rightarrow$ asocia a la derecha.
$mult :: Int \rightarrow Int \rightarrow Int \rightarrow Int$
- Notar que esta convención es consistente con la aplicación asociando a la izquierda.
- En Haskell todas las funciones están currificadas (salvo algunos casos particulares).

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.
  $length :: [a] \rightarrow Int$
  Para cualquier tipo a la función *length* es la misma.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo.
  $length :: [a] \to Int$
  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.
  $length :: [a] \rightarrow Int$
  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

$$
\begin{array}{ll}
length\ [False, True] & \leftarrow a = Bool \\
length\ [\texttt{'a'}, \texttt{'b'}] & \leftarrow a = Char
\end{array}
$$

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo.
  $length :: [a] \rightarrow Int$
  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

$$\begin{array}{ll} length\ [False, True] & \leftarrow a = Bool \\ length\ [\texttt{'a'}, \texttt{'b'}] & \leftarrow a = Char \end{array}$$

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

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas
- Esta sobrecarga se realiza mediante *clases de tipo*.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas
- Esta sobrecarga se realiza mediante *clases de tipo*.
- El operador $(+)$ tiene tipo:

$$(+) :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas
- Esta sobrecarga se realiza mediante *clases de tipo*.
- El operador $(+)$ tiene tipo:

$$(+) :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

- $(+)$ está definido para cualquier tipo que sea una *instancia* de la clase *Num*.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas
- Esta sobrecarga se realiza mediante *clases de tipo*.
- El operador $(+)$ tiene tipo:

$$(+) :: \mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

- $(+)$ está definido para cualquier tipo que sea una *instancia* de la clase *Num*.
- A diferencia del polimorfismo paramétrico, hay una definición distinta de $(+)$ para cada instancia.

Clases de tipo

- Haskell provee una serie de clases básicas:
- *Eq* son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$$
\begin{array}{ll}
(==) & :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool \\
(/=) & :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool
\end{array}
$$

¿Qué tipo no puede ser instancia de esta clase?

Clases de tipo

- Haskell provee una serie de clases básicas:
- *Eq* son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$$
\begin{aligned}
(==) &:: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool \\
(/=) &:: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool
\end{aligned}
$$

  ¿Qué tipo no puede ser instancia de esta clase?
- *Ord* son los tipos que además de ser instancias de *Eq* poseen un orden total.

$$
\begin{aligned}
(<), (\leqslant), (>), (\geqslant) &:: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool \\
min, max &:: Ord\ a \Rightarrow a \rightarrow a \rightarrow a
\end{aligned}
$$

Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

$$show :: Show\ a \Rightarrow a \rightarrow String$$

Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

$$show :: Show\ a \Rightarrow a \rightarrow String$$

- *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

$$read :: Read\ a \Rightarrow String \rightarrow a$$

Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

  $$show :: Show\ a \Rightarrow a \rightarrow String$$

- *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

  $$read :: Read\ a \Rightarrow String \rightarrow a$$

- ¿A qué tipo corresponde la instancia que leerá
  - *not* (*read* `"False"`)?

Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

  $$show :: Show\ a \Rightarrow a \rightarrow String$$

- *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

  $$read :: Read\ a \Rightarrow String \rightarrow a$$

- ¿A qué tipo corresponde la instancia que leerá
  - *not* (*read* `"False"`)?
  - *read* `"3"`?

Clases de tipo

- *Show* son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

$$show :: Show\ a \Rightarrow a \to String$$

- *Read* es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

$$read :: Read\ a \Rightarrow String \to a$$

- ¿A qué tipo corresponde la instancia que leerá
  - *not* (*read* `"False"`)?
  - *read* `"3"`?
  - *read* `"3"` :: *Int*?

Clases de Tipo

- *Num* son los tipos numéricos

Clases de Tipo

- *Num* son los tipos numéricos
- Sus instancias deben implementar

$$
\begin{aligned}
(+), (-), (*) &:: Num\ a \Rightarrow a \to a \to a \\
negate, abs, signum &:: Num\ a \Rightarrow a \to a
\end{aligned}
$$

Clases de Tipo

- *Num* son los tipos numéricos
- Sus instancias deben implementar

$$\begin{aligned}
(+), (-), (*) &:: Num\ a \Rightarrow a \rightarrow a \rightarrow a \\
negate, abs, signum &:: Num\ a \Rightarrow a \rightarrow a
\end{aligned}$$

¿Y la división?

Clases de Tipo

- *Num* son los tipos numéricos
- Sus instancias deben implementar

$$\begin{aligned}(+),(-),(*) &:: Num\ a \Rightarrow a \rightarrow a \rightarrow a\\ negate, abs, signum &:: Num\ a \Rightarrow a \rightarrow a\end{aligned}$$

  ¿Y la división?
- *Integral* son los tipos que son *Num* e implementan

$$div, mod :: Integral\ a \Rightarrow a \rightarrow a \rightarrow a$$

Clases de Tipo

- *Num* son los tipos numéricos
- Sus instancias deben implementar

$$\begin{aligned} (+), (-), (*) &:: Num\ a \Rightarrow a \rightarrow a \rightarrow a \\ negate, abs, signum &:: Num\ a \Rightarrow a \rightarrow a \end{aligned}$$

¿Y la división?

- *Integral* son los tipos que son *Num* e implementan

$$div, mod :: Integral\ a \Rightarrow a \rightarrow a \rightarrow a$$

- *Fractional* son los tipos que son *Num* e implementan

$$\begin{aligned} (/) &:: Fractional\ a \Rightarrow a \rightarrow a \rightarrow a \\ recip &:: Fractional\ a \Rightarrow a \rightarrow a \end{aligned}$$

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$\begin{aligned} abs \quad &:: Int \to Int \\ abs\ n &= \textbf{if}\ n \geqslant 0\ \textbf{then}\ n\ \textbf{else} - n \end{aligned}$$

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

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

- 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

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

- 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**"

Expresiones Condicionales

- Las funciones pueden ser definidas usando expresiones condicionales

$$\begin{aligned} &abs &&:: Int \to Int \\ &abs\ n &&= \mathbf{if}\ n \geqslant 0\ \mathbf{then}\ n\ \mathbf{else} - n \end{aligned}$$

- 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:

$$\begin{aligned} &signum &&:: Int \to Int \\ &signum\ n &&= \mathbf{if}\ n < 0\ \mathbf{then} - 1\ \mathbf{else} \\ & && \qquad \mathbf{if}\ n == 0\ \mathbf{then}\ 0\ \mathbf{else}\ 1 \end{aligned}$$

Ecuaciones con Guardas

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

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

Ecuaciones con Guardas

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

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

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

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

Ecuaciones con Guardas

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

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

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

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

- La condición *otherwise* se define en el preludio como

$$\mathit{otherwise} = \mathit{True}$$

Pattern Matching

- Muchas funciones se definen más claramente usando *pattern matching*.

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

Pattern Matching

- Muchas funciones se definen más claramente usando *pattern matching*.

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

- Los patrones se componen de constructores de datos y variables

Pattern Matching

- Muchas funciones se definen más claramente usando *pattern matching*.

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

- Los patrones se componen de constructores de datos y variables (salvo los patrones 0 y $n + 1$) .

Pattern Matching

- Muchas funciones se definen más claramente usando *pattern matching*.

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

- 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.

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

Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

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

Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

$$
\begin{array}{llll}
(\wedge) & & & :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} & \wedge & \mathit{True} & = \mathit{True} \\
\mathit{True} & \wedge & \mathit{False} & = \mathit{False} \\
\mathit{False} & \wedge & \mathit{True} & = \mathit{False} \\
\mathit{False} & \wedge & \mathit{False} & = \mathit{False}
\end{array}
$$

puede ser escrita en forma compacta como

$$
\begin{array}{llll}
(\wedge) & & & :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} & \wedge & \mathit{True} & = \mathit{True} \\
\_ & \wedge & \_ & = \mathit{False}
\end{array}
$$

Pattern Matching

- Usando el ingenio se pueden obtener definiciones concisas.

$$
\begin{array}{l}
(\wedge) \quad :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} \wedge \mathit{True} = \mathit{True} \\
\mathit{True} \wedge \mathit{False} = \mathit{False} \\
\mathit{False} \wedge \mathit{True} = \mathit{False} \\
\mathit{False} \wedge \mathit{False} = \mathit{False}
\end{array}
$$

puede ser escrita en forma compacta como

$$
\begin{array}{l}
(\wedge) \quad :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{True} \wedge \mathit{True} = \mathit{True} \\
\_ \wedge \_ = \mathit{False}
\end{array}
$$

- Notar la importancia del orden de las ecuaciones.

Patrones de tuplas

- Una tupla de patrones es un patrón.

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

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

Patrones de tuplas

- Una tupla de patrones es un patrón.

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

- ¿Qué hace la siguiente función?

$$f\ (x, (y, z)) = ((x, y), z)$$

Patrones de tuplas

- Una tupla de patrones es un patrón.

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

- ¿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

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

Patrones de Listas

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

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

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

$$\begin{aligned} &\mathit{head} && :: [a] \to a \\ &\mathit{head}\ (x{:}\ \_) && = x \end{aligned}$$

$$\begin{aligned} &\mathit{tail} && :: [a] \to [a] \\ &\mathit{tail}\ (\_ : xs) && = xs \end{aligned}$$

Patrones de Listas

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

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

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

$$\begin{aligned} &head && :: [a] \to a \\ &head\ (x : \_) && = x \\ &tail && :: [a] \to [a] \\ &tail\ (\_ : xs) && = xs \end{aligned}$$

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

Ejemplo usando patrones de listas

- Devolver el prefijo más grande que cumple un predicado.

$$
\begin{array}{ll}
\mathit{takeWhile} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] & \\
\mathit{takeWhile}\ p\ [\,] & = [\,] \\
\mathit{takeWhile}\ p\ (x : xs) \mid p\ x & = x : \mathit{takeWhile}\ p\ xs \\
\qquad\qquad\qquad\qquad \mid \mathit{otherwise} & = [\,]
\end{array}
$$

Ejemplo usando patrones de listas

- Devolver el prefijo más grande que cumple un predicado.

$$
\begin{aligned}
&\mathit{takeWhile} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a] \\
&\mathit{takeWhile}\ p\ [\,] && = [\,] \\
&\mathit{takeWhile}\ p\ (x : \mathit{xs}) \mid p\ x && = x : \mathit{takeWhile}\ p\ \mathit{xs} \\
&\qquad\qquad\qquad\quad\ \mid \mathit{otherwise} && = [\,]
\end{aligned}
$$

- Ejercicio: Definir la función *dropWhile* que elimina el prefijo más grande que cumple un predicado.

$$\mathit{dropWhile} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a]$$

- Definir la función *span* que divide una lista entre el prefijo más grande que cumple un predicado y el resto de la lista.

$$\mathit{span} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow ([a], [a])$$

Expresiones Lambda

- Se pueden construir funciones sin darles nombres usando *expresiones lambda*.

$$\lambda x \to x + x$$

- Estas funciones se llaman *funciones anónimas*.

Expresiones Lambda

- Se pueden construir funciones sin darles nombres usando *expresiones lambda*.

$$\lambda x \rightarrow x + x$$

- Estas funciones se llaman *funciones anónimas*.
- En Haskell escribimos $\backslash$ en lugar de la letra griega lambda λ

Expresiones Lambda

- Se pueden construir funciones sin darles nombres usando *expresiones lambda*.

$$\lambda x \to x + x$$

- Estas funciones se llaman *funciones anónimas*.
- En Haskell escribimos $\backslash$ en lugar de la letra griega lambda λ
- La notación proviene del cálculo lambda, en el cual se basa Haskell, y que estudiaremos en detalle más adelante.

Utilidad de las expresiones lambda

- Usando lambdas puedo explicitar que las funciones son simplemente valores:

$$add\ x\ y = x + y$$
$$add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

Utilidad de las expresiones lambda

- Usando lambdas puedo explicitar que las funciones son simplemente valores:

$$add\ x\ y = x + y$$
$$add = \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

- Evito tener que darle un nombre a una función que se usa una sola vez.

$$\begin{aligned} &impares\ n = map\ f\ [0 \mathinner{..} n - 1] \\ &\qquad\qquad \mathbf{where}\ f\ x = x * 2 + 1 \end{aligned}$$

Utilidad de las expresiones lambda

- Usando lambdas puedo explicitar que las funciones son simplemente valores:

$$
\begin{aligned}
&add\ x\ y = x + y \\
&add = \lambda x \to (\lambda y \to x + y)
\end{aligned}
$$

- Evito tener que darle un nombre a una función que se usa una sola vez.

$$
\begin{aligned}
&impares\ n = map\ f\ [0\,..\,n-1] \\
&\qquad\qquad\quad \textbf{where}\ f\ x = x * 2 + 1
\end{aligned}
$$

$$odds\ n = map\ (\lambda x \to x * 2 + 1)\ [0\,..\,n-1]$$

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:

$$
\begin{array}{l}
> (+)\ 1\ 2 \\
3
\end{array}
$$

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

$$
\begin{array}{l}
> (1+)\ 2 \\
3 \\
> (+2)\ 1 \\
3
\end{array}
$$

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

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

Conjuntos por comprensión

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

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

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 $\{1 \ldots 5\}$

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

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

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

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?

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

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?

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

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

Guardas

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

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

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?

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

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?

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

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

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

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

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 *String*s.

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

Zip

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

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

Zip

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

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

- Ejemplo: Lista de pares de elementos adyacentes:

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

Zip

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

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

- Ejemplo: Lista de pares de elementos adyacentes:

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

- ¿Está una lista ordenada?

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

Ejemplo *zip*: pares índice/valor

- Podemos usar *zip* para generar índices

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

Ejemplo *zip*: pares índice/valor

- Podemos usar *zip* para generar índices

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

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

$> \mathit{rangeof}\ 3\ 7\ [x \uparrow 2 \mid x \leftarrow [1\,..\,10]]$
$[16, 25, 36, 49, 64]$

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.

$$
\begin{aligned}
&\mathit{factorial} \quad :: \mathit{Int} \to \mathit{Int} \\
&\mathit{factorial}\ 0 = 1 \\
&\mathit{factorial}\ n = n * \mathit{factorial}\ (n-1)
\end{aligned}
$$

Recursión

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

$$\begin{array}{l} \mathit{factorial} \quad :: \mathit{Int} \to \mathit{Int} \\ \mathit{factorial}\ 0 = 1 \\ \mathit{factorial}\ n = n * \mathit{factorial}\ (n - 1) \end{array}$$

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

Recursión

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

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

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

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

Recursión Mutua

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

$$
\begin{aligned}
&\mathit{zigzag} :: [(a, a)] \to [a] \\
&\mathit{zigzag} = \mathit{zig}
\end{aligned}
$$

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

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

Recursión Mutua

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

$$
\begin{array}{l}
\mathit{zigzag} :: [(a, a)] \to [a] \\
\mathit{zigzag} = \mathit{zig}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{zig}\ [\,] & = [\,] \\
\mathit{zig}\ ((x, \_) : \mathit{xs}) & = x : \mathit{zag}\ \mathit{xs}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{zag}\ [\,] & = [\,] \\
\mathit{zag}\ ((\_, y) : \mathit{xs}) & = y : \mathit{zig}\ \mathit{xs}
\end{array}
$$

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

Ejemplo: Notación Polaca (NP)

- Forma de escribir operaciones aritméticas
- Los operadores se escriben en forma prefija

$$+\ 2\ 3 \text{ en lugar de } 2+3$$

- Inventada por el lógico Jan Łukasiewicz ($\sim$ 1920).

Ejemplo: Notación Polaca (NP)

- Forma de escribir operaciones aritméticas
- Los operadores se escriben en forma prefija

$$+\ 2\ 3 \text{ en lugar de } 2+3$$

- Inventada por el lógico Jan Łukasiewicz ($\sim$ 1920).
- Ventaja: No requiere paréntesis:

$$(6+5)*(4-2) \quad \text{se escribe} \quad * \;\; + \;\; 6\ 5 \;\; - \;\; 4\ 2$$

Ejemplo: Notación Polaca (NP)

- Forma de escribir operaciones aritméticas
- Los operadores se escriben en forma prefija

$$+\ 2\ 3 \text{ en lugar de } 2+3$$

- Inventada por el lógico Jan Łukasiewicz ($\sim$ 1920).
- Ventaja: No requiere paréntesis:

$$(6+5)*(4-2) \quad \text{se escribe} \quad * \;\; + \;\; 6\ 5 \;\; - \;\; 4\ 2$$

- Ej: ¿Cómo se escribe en NP la siguiente expresión?

$$(2+(3*5))*4$$

Evaluador de NP

$$
\begin{aligned}
&np :: String \to Integer \\
&np\ (\text{'+'} : xs) = ?
\end{aligned}
$$

Luego de un operador binario tenemos que leer dos expresiones enteras.

Evaluador de NP (2^{do} intento)

$$
\begin{array}{l}
np :: String \to Integer \\
np\ xs = \mathbf{let}\ (r, rest) = np_{aux}\ xs \\
\qquad\qquad \mathbf{in\ if}\ null\ rest \\
\qquad\qquad\quad \mathbf{then}\ r \\
\qquad\qquad\quad \mathbf{else}\ error\ (\texttt{"Error: Basura al final: "} + rest)
\end{array}
$$

La función auxiliar np_{aux} calcula una expresión entera y devuelve el resto de la cadena sin procesar

$$np_{aux} :: String \to (Integer, String)$$

Evaluador de NP (cont.)

La función auxiliar np_{aux} calcula un entero del prefijo y devuelve el resto de la cadena sin procesar

$$
\begin{array}{l}
np_{aux} :: String \quad \rightarrow (Integer, String) \\
np_{aux}\ (\texttt{'+'} : xs) = \textbf{let}\ (m, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\quad\ \ (n, ys) \quad = np_{aux}\ xs' \\
\qquad\qquad\qquad\qquad \textbf{in}\ (m + n, ys)
\end{array}
$$

Evaluador de NP (cont.)

La función auxiliar np_{aux} calcula un entero del prefijo y devuelve el resto de la cadena sin procesar

$$
\begin{array}{l}
np_{aux} :: String \quad \to (Integer, String) \\
np_{aux}\ (\texttt{'+'} : xs) = \textbf{let}\ (m, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\quad (n, ys) \quad = np_{aux}\ xs' \\
\qquad\qquad\qquad\quad \textbf{in}\ (m + n, ys) \\
np_{aux}\ (\texttt{'-'} : xs) = \textbf{let}\ (m, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\quad (n, ys) \quad = np_{aux}\ xs' \\
\qquad\qquad\qquad\quad \textbf{in}\ (m - n, ys) \\
np_{aux}\ (\texttt{'*'} : xs) = \textbf{let}\ (m, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\quad (n, ys) \quad = np_{aux}\ xs' \\
\qquad\qquad\qquad\quad \textbf{in}\ (m * n, ys) \\
np_{aux}\ (\texttt{'/'} : xs) = \textbf{let}\ (m, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\quad (n, ys) \quad = np_{aux}\ xs' \\
\qquad\qquad\qquad\quad \textbf{in}\ (div\ m\ n, ys) \\
\qquad\qquad\qquad\quad \vdots
\end{array}
$$

Eliminando redundancia

Código repetido huele a código mal escrito.

```
isOperator :: Char -> Bool
isOperator '+' = True
isOperator '-' = True
isOperator '*' = True
isOperator '/' = True
isOperator _   = False

operator :: Char -> Integer -> Integer -> Integer
operator '+' = (+)
operator '-' = (-)
operator '*' = (*)
operator '/' = div
operator _   = error "not an operator"
```

Terminamos el evaluador de NP

$$
\begin{array}{l}
np_{aux} :: String \to (Integer, String) \\
np_{aux}\ [\,] = error\ \texttt{"Error: esperando mas caracteres"} \\
\qquad \texttt{--}\ (0,\text{""}) \\
np_{aux}\ (x : xs) \mid isOperator\ x = \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad (n, xs') = np_{aux}\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad (m, ys) = np_{aux}\ xs' \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (operator\ x\ n\ m, ys) \\
\qquad\qquad\qquad \mid isDigit\ x \quad = \textbf{let} \\
\qquad\qquad\qquad\qquad\qquad\qquad (ns, xs') = span\ isDigit\ (x : xs) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (read\ ns, xs') \\
\qquad\qquad\qquad \mid isSpace\ x \quad = np_{aux}\ xs \\
\qquad\qquad\qquad \mid otherwise \quad = error\ (\texttt{"Error: encontre un "} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad + [x])
\end{array}
$$

Terminamos el evaluador de NP

$$
\begin{array}{l}
np_{aux} :: String \rightarrow (Integer, String) \\
np_{aux}\ [\,] = error\ \texttt{"Error: esperando mas caracteres"} \\
\quad\quad \texttt{--}\ (0,\text{""}) \\
np_{aux}\ (x : xs) \mid isOperator\ x = \textbf{let} \\
\quad\quad\quad\quad (n, xs') = np_{aux}\ xs \\
\quad\quad\quad\quad (m, ys) = np_{aux}\ xs' \\
\quad\quad\quad\quad \textbf{in}\ (operator\ x\ n\ m, ys) \\
\quad\quad \mid isDigit\ x \quad = \textbf{let} \\
\quad\quad\quad\quad (ns, xs') = span\ isDigit\ (x : xs) \\
\quad\quad\quad\quad \textbf{in}\ (read\ ns, xs') \\
\quad\quad \mid isSpace\ x \quad = np_{aux}\ xs \\
\quad\quad \mid otherwise \quad = error\ (\texttt{"Error: encontre un "} \\
\quad\quad\quad\quad\quad +\!\!+\ [x])
\end{array}
$$

$> np$ `"*+2*3 5 4"`

68

Referencias

- *Programming in Haskell*. Graham Hutton, (1er ed) CUP 2007, (2 ed) CUP 2016.
- *Introducción a la Programación Funcional con Haskell*. Richard Bird, Prentice Hall 1997.
- *Thinking Functionally with Haskell*. Richard Bird, CUP 2014.
- *"Can Programming Be Liberated from the von Neumann Style?"*. John Backus, Turing Award Lecture. 1977
- *Why Functional Programming Matters*. John Hughes, Computing Journal. Vol 32. 1989.
- *A History of Haskell: Being Lazy With Class*. Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages. 2007.