

RAZONANDO CON HASKELL

Un curso sobre
programación funcional

Blas C. Ruiz
Francisco Gutiérrez
Pablo Guerrero
José E. Gallardo

RAZONANDO CON HASKELL

**Un curso sobre
programación funcional**

Índice de fórmulas

Prólogo

Comunicado de prensa

1. Introducción al lenguaje Haskell. 98

2. Programación funcional básica con Haskell. 98

3. Programación funcional avanzada con Haskell. 98

Blas Carlos Ruiz Jiménez

Francisco Gutiérrez López

José Enrique Gallardo Ruiz

DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

Pablo Guerrero García

DEPARTAMENTO DE MATEMÁTICA APLICADA

E.T.S.I. INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

THOMSON

Australia • Canadá • España • Estados Unidos • México • Reino Unido • Singapur

ÍNDICE GENERAL

Índice de figuras

XIII

Prólogo

XVII

Convenios

XXIII

I Programación funcional básica con HASKELL 98

1

1. Programación funcional

1.1. Funciones	3
1.2. Sesiones y declaraciones	3
1.3. Reducción de expresiones	4
1.3.1. Órdenes de reducción aplicativo y normal	6
1.3.2. Evaluación perezosa	9
1.4. Sobre HASKELL	9

2. Introducción a HASKELL

2.1. El lenguaje HASKELL	13
2.2. Tipos simples predefinidos	13
2.2.1. El tipo <i>Bool</i>	15
2.2.2. El tipo <i>Int</i>	15
2.2.3. El tipo <i>Integer</i>	16
2.2.4. El tipo <i>Float</i>	16
2.2.5. El tipo <i>Double</i>	17
2.2.6. El tipo <i>Char</i>	18
2.2.7. Operadores de igualdad y orden	18
2.3. Constructores de tipos predefinidos	19
2.3.1. Tuplas	20
2.3.2. Listas	20
2.3.3. El constructor de tipo (\rightarrow)	21
2.4. Comentarios	22
2.5. Operadores	24
2.5.1. Operadores frente a funciones	24
2.6. Comparación de patrones	27
2.6.1. Patrones constantes	28

Índice general

2.6.2.	Patrones para listas	32
2.6.3.	Patrones para tuplas	33
2.6.4.	Patrones aritméticos	34
2.6.5.	Patrones nombrados o seudónimos	35
2.6.6.	El patrón subrayado	36
2.6.7.	Errores comunes	36
2.6.8.	Patrones y evaluación perezosa	37
2.7.	Expresiones <i>case</i>	38
2.8.	La función <i>error</i>	38
2.9.	Funciones a trozos	39
2.10.	Expresiones condicionales	40
2.11.	Definiciones locales	41
2.12.	Expresiones lambda	42
2.13.	Sangrado	43
2.14.	Ámbitos y módulos	45
2.15.	Ejercicios	46
3.	Funciones de orden superior y polimorfismo	49
3.1.	Parcialización	49
3.1.1.	Aplicación parcial	51
3.1.2.	Secciones	54
3.1.3.	Funciones de orden superior	56
3.1.4.	Una función de orden superior sobre naturales	58
3.2.	Polimorfismo	60
3.2.1.	La composición de funciones	62
3.2.2.	Otras funciones polimórficas	65
3.2.3.	Polimorfismo en listas	67
3.2.4.	Polimorfismo en tuplas	69
3.2.5.	Un iterador polimórfico sobre los naturales	70
4.	Definición de tipos	71
4.1.	Sinónimos de tipo	71
4.2.	Definición de tipos de datos	71
4.2.1.	Tipos enumerados	72
4.2.2.	Uniones	73
4.2.3.	Productos	74
4.2.4.	Tipos recursivos	78
4.2.5.	Tipos parametrizados (o polimórficos)	87
4.2.6.	Definiciones <i>newtype</i>	89
4.3.	Propiedades de funciones	90
4.3.1.	La propiedad universal de <i>foldNat</i>	97
4.4.	Sobrecarga y polimorfismo restringido	99
4.4.1.	Un ejemplo de sobrecarga	101
4.5.	Ejercicios	103

Índice general

5.	El sistema de clases de HASKELL	105
5.1.	Tipos y clases de tipos. Jerarquía de clases	105
5.1.1.	El sistema de clases	105
5.1.2.	Declaración de clase	109
5.1.3.	La clase <i>Eq</i> de PRELUDE	111
5.2.	Contextos	112
5.2.1.	Instancias paramétricas	115
5.3.	Subclases. La clase <i>Ord</i> de PRELUDE	115
5.3.1.	Un ejemplo: los enteros módulo n	118
5.3.2.	Intersección de clases	118
5.4.	Visualizando y leyendo datos. <i>Read</i> y <i>Show</i>	118
5.5.	Las clases <i>Num</i> , <i>Integral</i> y <i>Fractional</i> de PRELUDE	119
5.5.1.	Los tipos numéricos de HASKELL	122
5.5.2.	Ambigüedad en las constantes numéricas	123
5.5.3.	Promoción numérica	125
5.5.4.	Ejemplo: los racionales como instancias genéricas	126
5.6.	Ejercicios	128
6.	Programación con listas	131
6.1.	El tipo lista	131
6.1.1.	Secuencias aritméticas. La clase <i>Enum</i>	132
6.2.	Concatenación de listas	134
6.3.	Inducción sobre listas	136
6.4.	Selectores	137
6.5.	Emparejando listas	140
6.6.	Aplicando una función a los elementos de una lista	141
6.7.	Filtros	142
6.8.	Listas por comprensión	144
6.8.1.	Semántica de listas por comprensión	147
6.9.	Plegado de listas	148
6.9.1.	<i>foldr</i>	148
6.9.2.	La propiedad universal de <i>foldr</i>	151
6.9.3.	<i>foldl</i>	153
6.10.	Ordenación de listas	154
6.10.1.	Ordenación por inserción	155
6.10.2.	Ordenación por mezcla	156
6.10.3.	Ordenación rápida	158
6.11.	Problemas combinatorios	159
6.11.1.	Los segmentos iniciales de una lista	159
6.11.2.	Los segmentos consecutivos de una lista	160
6.11.3.	Permutaciones de una lista	161
6.12.	Otras funciones predefinidas	162
6.13.	Ejercicios	164

7. Entrada y salida	169
7.1. Operaciones de entrada y salida	169
7.1.1. El problema de la entrada y salida	169
7.1.2. El tipo <i>IO</i>	170
7.1.3. Excepciones	172
7.2. Un formateador de textos	174
7.2.1. Una implementación inefficiente	175
7.2.2. Una implementación eficiente	177
7.2.3. Utilidades para el manejo de documentos	179
7.2.4. Una clase de tipos documentables	179
7.2.5. Ejemplos	180
II Programación avanzada	183
8. Evaluación perezosa. Redes de procesos	185
8.1. Evaluación perezosa	185
8.1.1. Argumentos estrictos	185
8.1.2. Procesando estructuras infinitas en forma perezosa	186
8.2. Listas parciales y listas infinitas	189
8.2.1. Aproximaciones o listas parciales	189
8.2.2. Inducción sobre listas parciales	190
8.3. Redes finitas de procesos	191
8.3.1. La criba de Eratóstenes	193
8.3.2. El triángulo de Pascal	195
8.3.3. Procesos con varias entradas	196
8.3.4. La lista de factoriales	196
8.3.5. Los números de Fibonacci	198
8.3.6. Sucesiones genéricas	199
8.3.7. Los números de Hamming	202
8.4. Sucesiones contadoras	203
8.5. Ejercicios	207
9. Programación con árboles y grafos	211
9.1. Árboles	211
9.1.1. Funciones de orden superior sobre árboles	213
9.2. Árboles binarios	215
9.2.1. Árboles binarios de búsqueda	215
9.2.2. Funciones de orden superior para árboles binarios	219
9.2.3. Inducción para árboles binarios	220
9.3. Arrays	222
9.3.1. Una implementación inefficiente	222
9.3.2. Una implementación eficiente	224
9.4. Grafos y búsqueda en grafos	226
9.4.1. Búsqueda en anchura y en profundidad	227
9.4.2. Los grafos como instancias de una clase uniparamétrica	229

9.5. Grafos con pesos	232
9.5.1. Grafos con pesos como instancias de clases biparamétricas	232
9.5.2. Una clase HASKELL para grafos con pesos	234
9.6. Ejercicios	236
10. Programación modular y tipos abstractos de datos	243
10.1. Módulos	243
10.2. Bibliotecas estandarizadas	243
10.3. Declaraciones de módulos	243
10.4. Importación	245
10.4.1. Importación cualificada	246
10.5. Tipos abstractos de datos	247
10.6. Representación	247
10.6.1. Representación con una interfaz no sobrecargada	247
10.6.2. Representación con una interfaz sobrecargada	249
10.7. El TAD Conjunto (<i>Conjunto</i>)	253
10.8. El TAD Lista Ordenada (<i>OrdList</i>)	254
10.9. El TAD Diccionario (<i>Diccionario</i>)	255
10.10. Un índice KWIC (KeyWord In Context)	257
10.10.1. Datos del problema	258
10.10.2. La función <i>kwic</i>	259
10.10.3. Algunas funciones y sinónimos de tipos útiles	259
10.10.4. La función <i>creaNoClaves</i>	261
10.10.5. Modelo de datos para la resolución	261
10.10.6. La función <i>kwic'</i>	262
10.11. Ejercicios	264
11. Programación con mónadas	265
11.1. Concepto de mónada	265
11.2. Clases de constructores de tipos	266
11.2.1. La clase <i>Functor</i>	266
11.2.2. La clase <i>Monad</i>	269
11.2.3. La clase <i>MonadPlus</i>	272
11.3. Interpretación de las propiedades	273
11.3.1. Interpretación del operador (<i>>>=</i>)	276
11.4. Relación entre <i>functor</i> y mónada	277
11.5. La notación <i>do</i>	280
11.6. Ejemplos de mónadas	283
11.6.1. La mónada identidad	283
11.6.2. La mónada escritora	284
11.6.3. La mónada lectora	286
11.6.4. La mónada de transformadores de estado	288
11.6.5. La lista como mónada indeterminista	291
11.6.6. La mónada error	292
11.7. Operaciones con mónadas	296
11.7.1. Combinando mónadas	297

Algunos de los convenios usados en los diálogos con HUGS son los siguientes:

- Se usa PRELUDE para indicar que no hay ningún fichero de programa cargado en el intérprete en el momento de evaluar la expresión.
- Se usa MAIN para indicar que hay un fichero de programa cargado en el momento de evaluar la expresión. A veces aparecerá un nombre de módulo lo cual indica que dicho módulo se encuentra cargado.
- Se usa \$\$ para indicar la expresión que se evaluó previamente. Por ejemplo,

```
PRELUDE> [1..3] ++ [4..6]
[1,2,3,4,5,6] :: [Integer]
```

```
PRELUDE> length $$
```

```
6 :: Integer
```

En el segundo diálogo se calcula la longitud de la lista devuelta por el primer diálogo.

- Se usa el comando :t para que el intérprete muestre el tipo de una expresión (sin evaluarla):

```
:t "hola" ++ "adios"
"hola" ++ "adios" :: [Char]
```

- Se usa {Interrupted!} para indicar que se interrumpió la evaluación de una expresión pulsando una combinación especial de teclas (normalmente las teclas control y pausa simultáneamente):

```
PRELUDE> [1..]
[1,2,3,4{Interrupted!}]
```

Algunos de los convenios usados en las demostraciones y reducciones son:

- Se usa is para denotar que se ha usado la propiedad sustitutiva de la igualdad.
- Se usa HI para denotar que se ha usado la hipótesis de inducción.
- Una reducción como:

```
[] ++ [3,4,5]
→ { 1 } (++)
[3,4,5]
```

indica que se pasó de la primera expresión a la segunda usando la primera ecuación de la función (++) .

Algunos de los convenios usados para demostrar la igualdad entre expresiones:

```
(+) is (+)
(+) HI (+)
```

Probar si una expresión es válida (que satisfaga la lista de condiciones). El intérprete usa como prueba general el comando is. La otra forma es el comando HI. También se considera la posibilidad de probar si una expresión es válida usando la función prove. Esta función sirve para probar si una expresión es válida o no. Para ello se introducen de modo de uso organizaciones pertenecientes a la clase prove que implementan el comando que se utilizan en el código el comando prove se ejecuta.

```
prove (x) ==> is (x)
```

Algunas veces, el lenguaje Haskell no tiene la capacidad de demostrar la validez de una expresión porque no tiene la información suficiente. En estos casos, se considera la posibilidad de probar si una expresión es válida. Si esto es así, se considera el comando prove. Siempre que sea posible, el resultado es una expresión válida.

PROGRAMACIÓN FUNCIONAL BÁSICA CON HASKELL 98

El lenguaje Haskell 98 es un lenguaje de programación funcional. Es un lenguaje de programación declarativo que se basa en la evaluación normal. Los tipos de datos y las operaciones están definidos por el lenguaje. Los tipos de datos incluyen tipos de datos simples y tipos de datos compuestos. Los tipos de datos simples incluyen tipos de datos numéricos, tipos de datos de texto y tipos de datos booleanos. Los tipos de datos compuestos incluyen tipos de datos de lista y tipos de datos de tupla.

La sintaxis del lenguaje es muy similar a la de otros lenguajes de programación. Los operadores y las funciones tienen la misma sintaxis que en otros lenguajes de programación. Los tipos de datos y las operaciones están definidos por el lenguaje.

El lenguaje Haskell 98 es un lenguaje de programación declarativo que se basa en la evaluación normal. Los tipos de datos y las operaciones están definidos por el lenguaje.

Algunos de los lenguajes de programación declarativos más populares son el LISP y el Prolog. El lenguaje Haskell 98 es similar a estos lenguajes en su estructura y funcionamiento. Sin embargo, el lenguaje Haskell 98 es más fácil de aprender y más fácil de usar que el LISP y el Prolog.

El lenguaje Haskell 98 es un lenguaje de programación declarativo que se basa en la evaluación normal. Los tipos de datos y las operaciones están definidos por el lenguaje.

El lenguaje Haskell 98 es un lenguaje de programación declarativo que se basa en la evaluación normal. Los tipos de datos y las operaciones están definidos por el lenguaje.

1

PROGRAMACIÓN FUNCIONAL

1.1. FUNCIONES

En matemáticas, una *función* f entre dos conjuntos A y B , llamados *conjunto inicial* y *final*, es una correspondencia que a cada elemento de un subconjunto de A , llamado el *dominio* de f , le hace corresponder uno y sólo uno de un subconjunto de B llamado *imagen* de f . La notación utilizada suele ser:

$$\begin{aligned} f : A &\rightarrow B \\ f(x) &\mapsto \dots \end{aligned}$$

Un ejemplo simple es la función *sucesor* con dominio e imagen el conjunto de los números enteros, que asocia a cada entero su siguiente:

$$\begin{aligned} \text{sucesor} : \mathbb{Z} &\rightarrow \mathbb{Z} \\ \text{sucesor}(x) &\mapsto x + 1 \end{aligned}$$

También existen funciones de varias variables. Por ejemplo, la función:

$$\begin{aligned} \text{sumaCuadrados} : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z} \\ \text{sumaCuadrados}(x, y) &\mapsto x^2 + y^2 \end{aligned}$$

es de dos variables. Incluso es habitual definir funciones de cero variables denominadas *funciones constantes* o simplemente *constantes*, como:

$$\begin{aligned} \pi : &\mathbb{R} \\ \pi &\mapsto 3.141592\dots \end{aligned}$$

Cuando definimos una función, suele interesarnos evaluar la función para ciertos valores de la variable. Por ejemplo, escribimos:

$$\text{sucesor}(1) \implies 2$$

para denotar que el valor de la función *sucesor* para el valor 1 es 2. Se dice también que el resultado obtenido al *aplicar* la función *sucesor* al valor 1 es 2. Cuando la función es de varias variables, se aplicará al número correspondiente de argumentos. Por ejemplo, $\text{sumaCuadrados}(2, 3) \implies 13$.

1.2. SESIONES Y DECLARACIONES

Programar consiste en especificar al ordenador los pasos que debe seguir para resolver un problema. La solución a un problema es un valor que se calcula a partir de ciertos datos, por lo que un modo natural de describir programas es mediante funciones.

Desde el punto de vista de la programación funcional, el computador actúa como una calculadora o *evaluador* que calcula el resultado de las expresiones que introducimos. Para ello, muestra una línea como la siguiente solicitando la expresión a evaluar:

PRELUDE>

El programador dispone, en principio, de un conjunto de valores, funciones y operadores (llamados *predefinidos*) cuyo significado es conocido por el evaluador y que se encuentran definidos en lo que se conoce como el PRELUDE. Por ejemplo, el evaluador puede trabajar con valores numéricos enteros:

PRELUDE> 1 + 2
3 :: Integer

Las líneas anteriores representan una *sesión* (o *diálogo*) con el evaluador. La interpretación de las líneas anteriores es la siguiente: se solicitó al evaluador que calculase el valor de la expresión $1 + 2$ y éste respondió 3. El evaluador también mostró el tipo del resultado. Nuestro lenguaje de programación (que se llama HASKELL [Peyton Jones, 2003]) permite trabajar con distintas categorías de datos (o *tipos de datos*). Cada valor tiene asociado un tipo. El valor 1 tiene asociado el tipo *Integer*, que denota valores enteros en HASKELL. Otros tipos de datos predefinidos son los valores numéricos reales, los caracteres (letras y otros símbolos) y las listas (secuencias de datos del mismo tipo).

El siguiente ejemplo utiliza valores reales:

PRELUDE> cos pi
-1.0 :: Double

En la expresión anterior, *cos* es el *identificador* de la función coseno, y *pi* el de la constante π . El tipo *Double* se corresponde con valores numéricos reales. El evaluador contestó, en este caso, que el coseno de π es el valor real -1.0. La función *cos* tiene un único argumento mientras que *pi* es una función de cero argumentos (o *constante*). Obsérvese que la notación es distinta a la habitual en matemáticas. El argumento de la función no va entre paréntesis. Al evaluar una función, los argumentos se escriben tras el nombre de ésta, usando como separador uno o varios espacios en blanco. Sólo los argumentos que son expresiones compuestas van entre paréntesis, como en:

PRELUDE> cos (2 * pi)
1.0 :: Double

que calcula el coseno de 2π . El símbolo * es el operador producto y, a diferencia de lo que ocurre en matemáticas, no puede ser omitido. Para aplicar una función *f* a un argumento *z* se escribe *f z*. Esta notación, denominada *currificada*, separa funciones y argumentos por espacios y además de hacer las expresiones más breves, permite utilizar las funciones de un modo más versátil, como veremos posteriormente.

El siguiente ejemplo es un poco más elaborado:

PRELUDE> [1..5]
[1, 2, 3, 4, 5] :: [Integer]

PRELUDE> sum [1..10]
55 :: Integer

Primero se pidió al evaluador que calculase la lista de enteros consecutivos que recorre desde el uno al cinco. *[Integer]* indica que [1, 2, 3, 4, 5] es una lista de valores enteros. También se calculó la suma de los primeros diez valores enteros. *sum* es la función que toma como argumento una lista de valores numéricos y los suma. Para ilustrar funciones de más de un argumento podemos usar *mod*, una función de dos argumentos que devuelve el resto que se obtiene al dividir el primero entre el segundo:

PRELUDE> mod 10 3
1 :: Integer

PRELUDE> mod 10 (3 + 1)
2 :: Integer

Como veímos, el lenguaje HASKELL proporciona un rico conjunto de elementos predefinidos que pueden ser usados por el programador. Sin embargo, la característica más interesante es que este conjunto es ampliable. En efecto, el programador puede definir nuevas funciones, nuevos operadores e incluso nuevos tipos de datos.

Como primer ejemplo podemos definir una función que calcule el sucesor de un número entero. La definición de esta función en HASKELL es:

sucesor :: Integer → Integer
sucesor x = *x* + 1

En esta definición, *sucesor* es el nombre de la nueva función. El programador puede elegir cualquier nombre para las funciones que defina, aunque es conveniente que el nombre refleje el comportamiento de ésta. La primera línea es una *declaración de tipo*, e indica que *sucesor* es una función de enteros en enteros, es decir, una función que toma un único argumento entero y devuelve como resultado un valor entero. HASKELL es un lenguaje *fuertemente tipificado*, lo cual significa que cada elemento tiene asociado un tipo. Como veremos posteriormente, las declaraciones de tipo son opcionales, ya que el evaluador puede *inferir* el tipo de cualquier elemento a partir de su definición.

La segunda línea es una *ecuación* y proporciona el *método de cálculo* de la función. *x* es el *párametro formal* de la función (representa el argumento de ésta). Podría haberse usado otro nombre para el parámetro formal, por ejemplo *y*, y definir así la función:

sucesor :: Integer → Integer
sucesor y = *y* + 1

La expresión tras el símbolo = indica cómo evaluar la función a partir de su argumento.

Tras proporcionar la declaración de función anterior al evaluador, éste puede reducir expresiones en las que intervenga la nueva función. Una posible sesión es:

MAIN> sucesor 3
4 :: Integer

MAIN> 10 * sucesor 3
40 :: Integer

La siguiente declaración muestra la definición de una función de dos argumentos:

sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados *x y* = *x * x + y * y*

La función toma dos argumentos enteros y calcula la suma de sus cuadrados:

MAIN> *sumaCuadrados* 2 3 MAIN> *sumaCuadrados* (2 + 2) 3
13 :: Integer 25 :: Integer

La declaración de tipo indica que la función toma dos enteros y devuelve un entero. En general, los tipos de los distintos argumentos de la función aparecen separados por el *constructor de tipo* →. El último tipo que aparece en una declaración de tipo es el del resultado de la función definida. El cuerpo de la función consta de la parte izquierda y la parte derecha, separadas por el símbolo =. La parte izquierda consta del nombre de la función definida y los parámetros formales para cada argumento, separados por espacios. La parte derecha es una expresión que indica el valor devuelto por la función.

1.3. REDUCCIÓN DE EXPRESIONES

La labor del evaluador es calcular el resultado que se obtiene al evaluar una expresión utilizando las definiciones de las funciones involucradas. Para ello, el evaluador *simplifica* la expresión original todo lo posible y muestra el resultado. La simplificación se produce, en general, tras varios pasos. Por ejemplo, dada la siguiente declaración de función que calcula el doble de su argumento:

doble :: Integer → Integer
doble *x* = *x + x*

podemos calcular el valor de la expresión 5 * *doble* 3 del siguiente modo:

5 * *doble* 3
⇒ { por la definición de *doble* }
5 * (3 + 3)
⇒ { por el operador (+) }
5 * 6
⇒ { por el operador (*) }
30

Llamaremos a cada uno de los pasos anteriores una *reducción*, que denotaremos con el símbolo ⇒. Como vemos, en cada paso de reducción el evaluador busca una parte de la expresión que sea simplificable (que llamaremos *redex*) y la simplifica. Hemos subrayado el *redex* a reducir en cada paso del ejemplo. Así, (3 + 3) es un *redex* que se simplifica a 6 en la segunda reducción. Obsérvese que una función seguida de sus parámetros (i.e., una *aplicación de función*) se reduce sustituyendo, en la parte derecha de la ecuación de la función, los *parámetros formales* o *argumentos* por los que aparecen en la llamada (también llamados *parámetros reales* o simplemente *parámetros*). Cuando una expresión no pueda reducirse más se dice que está en *forma normal*.

1.3 - Reducción de expresiones

La labor del evaluador consiste entonces en buscar un *redex* en la expresión, reducirlo y repetir este proceso hasta que la expresión esté en forma normal. Una vez alcanzada la forma normal, el evaluador muestra el resultado.

Sin embargo, esta definición del comportamiento del evaluador es ambigua. En algunas expresiones existe más de un *redex*. Por ejemplo, en la expresión *doble* (*doble* 3) existen dos *redexes*. Podemos reducir la expresión desde dentro hacia fuera, es decir, reducir primero los *redexes* internos (los más anidados).

doble (*doble* 3)
⇒ { por la definición de *doble* }
doble (3 + 3)
⇒ { por el operador (+) }
doble 6
⇒ { por la definición de *doble* }
6 + 6
⇒ { por el operador (+) }
12

Ésta es la forma en la que la mayoría de nosotros, probablemente, habríamos reducido la expresión anterior. Sin embargo, como veremos a continuación, esta estrategia presenta algunos problemas. Una estrategia mejor consiste en reducir la expresión desde fuera hacia dentro, es decir, reducir primero los *redexes* externos (los menos anidados). Para ello hemos de observar que la ecuación de la función *doble* puede ser vista como una *regla de reescritura* que indica que podemos reemplazar una aparición de esta función seguida de un argumento por dos copias del argumento separadas por el operador (+):

doble [x] ⇒ [x] + [x]

Es decir, no es necesario evaluar previamente el parámetro para aplicar la definición de la función *doble*. Se pueden pasar los parámetros a las funciones como expresiones sin reducir, no necesariamente como valores. Utilizando esta estrategia, la reducción es:

doble (*doble* 3)
⇒ { por la definición de *doble* }
doble 3 + *doble* 3
⇒ { por la definición de *doble* }
(3 + 3) + *doble* 3
⇒ { por el operador (+) }
6 + *doble* 3
⇒ { por la definición de *doble* }
6 + (3 + 3)
⇒ { por el operador (+) }
6 + 6
⇒ { por el operador (+) }
12

Obsérvese qué, tras la primera reducción, la expresión $(doble\ 3) + (doble\ 3)$ no es un *redex* ya que el operador (+), al igual que los demás operadores aritméticos, sólo puede ser reducido cuando sus dos parámetros sean valores en forma normal (es decir, cuando sean números). Para reducir la expresión $e_1 + e_2$ hay que reducir previamente e_1 y e_2 . Este tipo de funciones que para ser evaluadas necesitan que sus parámetros estén en forma normal se llaman *estrictas*. Hay entonces dos *redexes* más externos: cada una de las apariciones de la expresión *doble 3*. Cuando esto ocurra, reduciremos el *redex* que aparezca más a la izquierda en la expresión.

Podemos observar que, sea cual sea la estrategia seguida en las reducciones, el resultado final (el valor 12) es el mismo. Esto es consecuencia de una propiedad conocida como *transparencia referencial* que establece que una misma expresión denota siempre el mismo valor. La reducción cambia la forma de una expresión pero no su valor.

Puede entonces parecer que si aparecen varios *redexes*, podemos elegir cualquiera ya que el resultado final no variará. Sin embargo, la reducción de un *redex* equivocado puede que no conduzca a la forma normal de una expresión. A modo de ejemplo, consideremos las siguientes definiciones de funciones:

$$\begin{aligned} \text{infinito} &:: \text{Integer} \\ \text{infinito} &= 1 + \text{infinito} \\ \text{cero} &:: \text{Integer} \rightarrow \text{Integer} \\ \text{cero } x &= 0 \end{aligned}$$

La función *cero* devuelve el valor 0 sea cual sea su argumento, por lo que tendremos $\forall n :: \text{Integer} \cdot \text{cero } n \Rightarrow 0$. En particular, $\text{cero infinito} \Rightarrow 0$. Si en cada momento de la reducción elegimos el *redex* más interno (el más anidado) obtenemos la siguiente reducción:

$$\begin{aligned} \text{cero infinito} &\Rightarrow \{\text{por definición de infinito}\} \\ &\Rightarrow \{\text{cero } (1 + \text{infinito})\} \\ &\Rightarrow \{\text{por definición de infinito}\} \\ &\Rightarrow \{\text{cero } (1 + (1 + \text{infinito}))\} \\ &\Rightarrow \{\text{por definición de infinito}\} \\ &\dots \end{aligned}$$

y la evaluación no terminaría nunca, por lo que no obtendríamos ningún resultado. Sin embargo, si en cada paso elegimos el *redex* más externo, se alcanza la forma normal en un solo paso:

$$\begin{aligned} \text{cero infinito} &\Rightarrow \{\text{por definición de cero}\} \\ &\Rightarrow 0 \end{aligned}$$

Como vemos, la estrategia utilizada para seleccionar el *redex* es crucial, ya que puede hacer que se obtenga o no la forma normal de la expresión.

1.3.1. ÓRDENES DE REDUCCIÓN APlicativo Y NORMAL

Un *orden de reducción* es una estrategia que indica qué *redex* hay que seleccionar en cada paso de la reducción. Existen varios órdenes de reducción: dos de los más interesantes son el *orden aplicativo* y el *orden normal*.

El *orden aplicativo* es el utilizado en la primera reducción del ejemplo anterior y consiste en seleccionar siempre el *redex* más interno (el más anidado en la expresión). En caso de que existan varios *redexes* que cumplan la condición anterior, se selecciona el que aparece más a la izquierda en la expresión. Esta estrategia de reducción es conocida como *paso de parámetros por valor* (*call by value*). Esto se debe a que, ante una aplicación de función, se reducen primero los parámetros de la función. Una vez conocidos los valores de los parámetros, se sustituyen en la ecuación de la función. A los evaluadores que utilizan este orden se les llama *estrictos* o *impacientes*. Uno de los problemas del *paso por valor* es que, a veces, se efectúan reducciones que no son necesarias para calcular el valor de la expresión. Por ejemplo, la siguiente reducción usa este orden para calcular el valor de la expresión *cero* ($10 * 4$):

$$\begin{aligned} \text{cero } (10 * 4) &\Rightarrow \{\text{por el operador } (*)\} \\ &\Rightarrow \text{cero } 40 \\ &\Rightarrow \{\text{por definición de cero}\} \\ &\Rightarrow 0 \end{aligned}$$

La expresión $10 * 4$ fue reducida, aunque dicha reducción no es necesaria para calcular el resultado, ya que la función *cero* no necesita conocer el valor de su argumento. El ejemplo puede parecer artificial, pero en la práctica es habitual definir funciones que no necesitan evaluar todos sus parámetros. La estrategia de *paso por valor* tiene un problema aún más grave: puede no conducir a la forma normal de la expresión. Éste es el caso de la expresión *cero infinito*.

El *orden de reducción normal* consiste en seleccionar el *redex* más externo (menos anidado) y, en caso de conflicto, de entre los más externos el que aparece más a la izquierda de la expresión. Esta estrategia de reducción se conoce como *paso de parámetros por nombre* (*call by name*), ya que se pasan como parámetro expresiones en vez de valores. Este orden fue el utilizado en la segunda reducción de la expresión *cero infinito*. A los evaluadores que utilizan el orden normal se les llama *no estrictos*. Una característica muy interesante del *paso por nombre* es que es *normalizante*: si la expresión considerada tiene forma normal, una reducción que use este orden la alcanza. Este resultado es conocido como el *teorema de estandarización*. Además, un evaluador no estricto sólo reducirá aquellos *redexes* que son necesarios para calcular el resultado final. Esta característica hace posible trabajar con estructuras de datos infinitas en los programas, aun siendo la memoria del ordenador finita. Estudiaremos esta posibilidad en capítulos posteriores.

1.3.2. EVALUACIÓN PEREZOSA

Una reducción con *paso por nombre* puede hacer que ciertas expresiones se reduzcan varias veces. Podemos observar que en la reducción no estricta de la expresión *doble* (*doble 3*) la expresión $(3 + 3)$ es calculada dos veces. Esto no ocurre en la

reducción que usa *paso por valor* para la misma expresión. La estrategia de *paso de parámetros por necesidad* (*call by need*), también conocida como *evaluación perezosa*, soluciona este problema. La evaluación perezosa consiste en utilizar *paso por nombre* y recordar los valores de los argumentos ya calculados para evitar que el cálculo se repita.

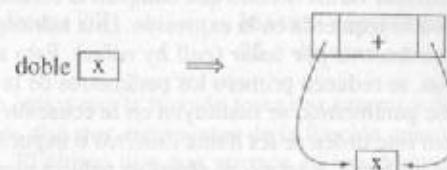


Figura 1.1: Representación perezosa de la función *doble*.

Para ello cada expresión se representa mediante un *grafo*. Por ejemplo, la Figura 1.1 muestra la reducción de grafos correspondiente a la definición de la función *doble*. Puede observarse que en vez de crear dos copias del argumento de la función, se crean dos referencias al argumento original.

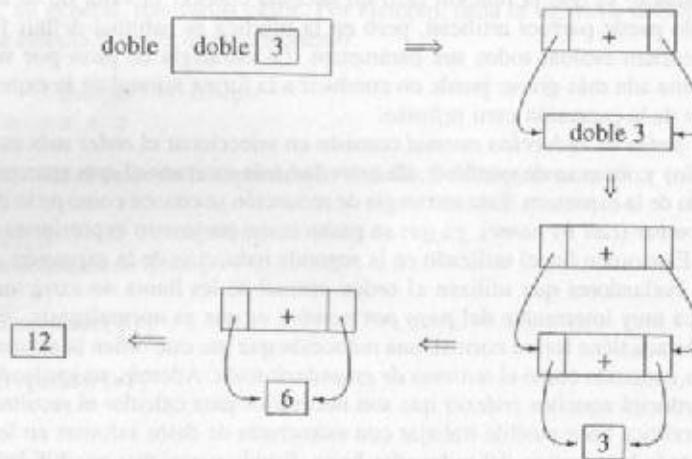


Figura 1.2: Reducción perezosa de *doble (doble 3)*.

La Figura 1.2 muestra la reducción de la expresión *doble (doble 3)* utilizando evaluación perezosa. Con objeto de poder expresar textualmente reducciones perezosas, nombraremos las expresiones compartidas. Así, la reducción de la Figura 1.2 la escribiríamos como:

1.4 - Sobre HASKELL

doble (doble 3)
 $\Rightarrow \{ \text{por la definición de } \textit{doble} \}$
 $a + a \text{ donde } a = \textit{doble 3}$
 $\Rightarrow \{ \text{por la definición de } \textit{doble} \}$
 $a + a \text{ donde } a = b + b \text{ donde } b = 3$
 $\Rightarrow \{ \text{por el operador } (+) \}$
 $a + a \text{ donde } a = 6$
 $\Rightarrow \{ \text{por el operador } (+) \}$

12

Obsérvese que la evaluación de las sumas no se ha repetido y que sólo fueron necesarias 4 reducciones para alcanzar la forma normal. Esto es un resultado general, ya que utilizando evaluación perezosa no se realizarán más reducciones que utilizando *paso por valor*. Como vemos, la evaluación perezosa posee las ventajas del *paso por nombre* y no es menos eficiente que el *paso por valor*. Es por ello que la mayoría de las implementaciones de HASKELL utilizan un evaluador perezoso.

1.4. SOBRE HASKELL

El lenguaje de programación HASKELL surge tras una reunión celebrada en Septiembre de 1987 en la conferencia *Functional Programming Languages and Computer Architecture* (FPCA '87). En dicha reunión se forma un comité cuyo objetivo es diseñar un lenguaje funcional puro y no estricto que unifique las ideas de distintos lenguajes funcionales existentes en dicho momento, con idea de constituir un estándar tanto para la comunidad educativa como para la investigadora. El lenguaje ha ido pasando por distintas versiones hasta alcanzar una versión estable que se denomina HASKELL 98 [Peyton Jones, 2003; Peyton Jones y Hughes, 2002].

Existen varias implementaciones disponibles para HASKELL 98. Una de las más populares es HUGS 98, una implementación interpretada de HASKELL 98 desarrollada por Mark Jones. Este software es de libre distribución y está disponible para los sistemas más utilizados actualmente (Unix, Windows y Macintosh). Nosotros hemos probado los ejemplos que aparecen en este libro usando Winhugs, una versión para Windows de dicho intérprete desarrollada por uno de los autores del libro. Los lectores interesados pueden descargar una copia del intérprete en la dirección <http://haskell.org/hugs>.

El lector interesado en encontrar más información sobre HASKELL puede acceder a la página web del mismo en <http://haskell.org>. Allí encontrará enlaces al software, ejemplos, documentación y otros contenidos relacionados con el lenguaje.

2

INTRODUCCIÓN A HASKELL

2.1. EL LENGUAJE HASKELL

HASKELL es un lenguaje funcional *puro, no estricto y fuertemente tipificado*. La primera de estas características indica que las funciones definibles en HASKELL cumplen la propiedad de *transparencia referencial*:

Una misma expresión denota siempre el mismo valor, sea cual sea el punto del programa en que aparezca.

La pureza del lenguaje hace que el razonamiento ecuacional utilizado en matemáticas, que nos permite sustituir expresiones por otras equivalentes, sea aplicable a los programas. Esta propiedad hace viable una serie de técnicas que permiten, desde comprobar la corrección de los programas desarrollados (que realmente realizan la labor para la que fueron diseñados) hasta transformar programas en otros equivalentes más eficientes (que realizan la misma labor utilizando menos recursos como pueden ser el tiempo o la cantidad de memoria utilizada). Estudiaremos estas técnicas en un capítulo posterior.

El término no estricto indica que el orden utilizado para reducir expresiones no es el orden aplicativo. Las implementaciones de HASKELL suelen usar evaluación perezosa. Como vimos en el Capítulo 1, ésta posee varias ventajas, como son:

- Es normalizante, es decir, siempre se encuentra la forma normal de aquellas expresiones que la poseen.
- Se realiza el trabajo mínimo a la hora de evaluar una expresión, ya que sólo se reducen aquellos *redexes* necesarios para calcular el valor normal de la expresión.
- Es razonablemente eficiente, ya que no se realizan más reducciones que utilizando paso de parámetros por valor.
- Es posible trabajar con estructuras de datos infinitas aun siendo la memoria del computador finita.

La *tipificación fuerte* indica que los elementos del lenguaje utilizables están clasificados en distintas categorías o tipos. Cada objeto del lenguaje (valores, funciones, operadores, etc.) tiene un tipo. Esta información permite comprobar que se hace un uso consistente de los distintos elementos del lenguaje. El sistema de tipos de HASKELL es *estático*. Esto significa que todas las comprobaciones de tipos se realizan durante la compilación del

programa, antes de la ejecución de éste. Los programas con errores de tipo son rechazados por el compilador (o intérprete) y no pueden ser ejecutados. Todo programa que pase la comprobación de tipos no producirá un error debido al uso inconsistente de algún elemento del lenguaje durante su ejecución. La comprobación de tipos permite detectar un gran número de errores lógicos en el programa en sus fases iniciales de desarrollo.

La mayoría de los lenguajes tipificados son menos flexibles que los no tipificados. En lenguajes tipificados como PASCAL es necesario escribir distintas funciones para trabajar con estructuras de datos idénticas si éstas almacenan datos con tipos distintos, aún cuando los cálculos realizados no dependan de dichos tipos. Por ejemplo, es necesario escribir dos funciones distintas para calcular la longitud de una lista de valores enteros y la longitud de una lista de valores reales. En lenguajes no tipificados, como PROLOG o LISP, una única función es válida para listas de cualquier tipo. El problema de los lenguajes no tipificados es que muchos de los errores en los programas no pueden ser detectados por el compilador por lo que se deja esta responsabilidad al programador, lo cual hace que los programas desarrollados sean menos seguros.

El sistema de tipos de HASKELL es uno de los más sofisticados hoy en día. Se trata de un sistema *polimórfico* que permite recuperar gran parte de la flexibilidad de los lenguajes no tipificados, pero que a la vez mantiene la corrección de los programas. El sistema de tipos también permite la *sobrecarga*: asignar un mismo nombre de función a un conjunto de funciones distintas y seleccionar la versión adecuada a utilizar en una expresión a partir de los tipos de los argumentos.

A diferencia de la mayoría de los lenguajes de programación, HASKELL usa un sistema de *inferencia de tipos* que hace que las anotaciones de tipo en un programa sean opcionales. Esto es posible ya que siempre se puede calcular el tipo de un elemento a partir de su definición. Sin embargo, el tipo de una función es una de las documentaciones más importantes que se puede aportar sobre ella (indica el tipo de argumentos que ésta espera y el tipo del resultado que produce), por lo que es buena práctica de programación acompañar cualquier definición de función con su tipo al declararla. Nosotros intentaremos seguir este criterio a lo largo del libro.

Un programa en HASKELL consiste en una o más definiciones de funciones. Para introducir una función hay que:

- Declararla: se trata de indicar el tipo de los argumentos que toma y del resultado que produce.
- Definirla: se trata de dar el método para computar el valor de dicha función.

Aunque el programador declare un tipo, el compilador lo infiere y comprueba que éste sea válido. En otro caso avisa con un error.

La declaración consiste en el nombre de la función, el signo :: y el tipo de ésta (:: se lee "tiene el tipo").

La definición de una función consta del nombre de la función, seguida de los nombres asociados a cada parámetro formal separados por espacios en blanco, el signo igual (=) y la expresión que constituye el cuerpo de la función.

Los principales tipos de datos básicos predefinidos en HASKELL son: *Char*, *Int*, *Integer*, *Float*, *Double* y *Bool*. Nótese que los *identificadores* (o nombres) de tipo siempre comienzan con una letra mayúscula en HASKELL. Por otro lado, los identificadores de funciones y variables comenzarán siempre por una letra minúscula.

2.2 - Tipos simples predefinidos

Ejemplo 2.1 Un fichero de texto con el siguiente contenido es un programa HASKELL que define dos funciones:

-- Un ejemplo de fichero Haskell

-- Calcula el siguiente entero al argumento
 $\text{sucesor} :: \text{Integer} \rightarrow \text{Integer}$
 $\text{sucesor } x = x + 1$

-- Calcula la suma de los cuadrados de sus dos argumentos
 $\text{sumaCuadrados} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$
 $\text{sumaCuadrados } x y = x * x + y * y$

Las líneas que comienzan por dos guiones son comentarios (no forman parte del programa, sólo lo documentan).

Ejemplo

2.2. TIPOS SIMPLES PREDEFINIDOS

Existe una colección de tipos de datos, funciones y operadores que pueden usarse en cualquier programa. Estos son los elementos del lenguaje *predefinidos*¹. A continuación estudiamos los principales tipos de datos predefinidos y las funciones asociadas a éstos.

2.2.1. EL TIPO *Bool*

Los valores con este tipo representan expresiones lógicas cuyo resultado puede ser verdadero o falso. Sólo hay dos valores constantes para este tipo: *True* y *False* (han de escribirse así exactamente), que representan los dos resultados posibles.

FUNCIONES Y OPERADORES

Los siguientes operadores y funciones predefinidos operan con valores booleanos:

- $(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$. Conjunction lógica.
- $(||) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$. Disyunción lógica.
- $\text{not} :: \text{Bool} \rightarrow \text{Bool}$. Negación lógica.
- $\text{otherwise} :: \text{Bool}$. Función constante que devuelve el valor *True*.

El comportamiento de estas funciones puede verse en la sesión y tablas siguientes:

PRELUDE> *True* && *False*
False :: *Bool*

PRELUDE> *not* (*True* && *False*)
True :: *Bool*

<i>v1</i>	<i>v2</i>	<i>v1 && v2</i>	<i>v1 v2</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>

<i>v</i>	<i>not v</i>
<i>True</i>	<i>False</i>
<i>False</i>	<i>True</i>

¹Tal colección de objetos aparece definida en un fichero de programa especial llamado PRELUDE.

2.2.2. EL TIPO *Int*

Los valores de este tipo son números enteros de precisión limitada que cubren al menos el intervalo $[-2^{29}, 2^{29} - 1]$. Pueden usarse las expresiones *minBound :: Int* y *maxBound :: Int* para determinar exactamente este intervalo. Así, en HUGS:

```
PRELUDE> minBound :: Int
- 2147483648 :: Int
```

```
PRELUDE> maxBound :: Int
2147483647 :: Int
```

por lo que para esta implementación del lenguaje el intervalo es $[-2^{31}, 2^{31} - 1]$. Los valores constantes de este tipo se escriben con la notación científica habitual.

FUNCIONES Y OPERADORES

Algunas de las funciones y operadores definidos para este tipo son:

- $(+), (-), (*) :: Int \rightarrow Int \rightarrow Int$. Suma, resta y producto de enteros.
- $(\wedge) :: Int \rightarrow Int \rightarrow Int$. Operador potencia; el exponente debe ser natural.
- *div, mod :: Int \rightarrow Int \rightarrow Int*. Cociente y resto de dividir dos enteros.
- *abs :: Int \rightarrow Int*. Valor absoluto.
- *signum :: Int \rightarrow Int*. Devuelve 1, -1 o 0, según el signo del argumento entero.
- *negate :: Int \rightarrow Int*. Invierte el signo de su argumento; también puede usarse para este propósito un uso prefijo (antes de la expresión a negar) del signo menos.
- *even, odd :: Int \rightarrow Bool*. Comprueban la naturaleza par o impar de un número.

Ejemplo 2.2 Podemos definir una función que a partir de la relación:

$$\text{máximo } (x, y) = \frac{(x+y) + |x-y|}{2}$$

calcule el máximo de sus argumentos:

```
máximo :: Int → Int → Int
máximo x y = div ((x + y) + abs(x - y)) 2
```

Example

2.2.3. EL TIPO *Integer*

Los valores de este tipo son números enteros de precisión ilimitada. Se usan cuando los números tratados se salen del rango de los valores del tipo *Int*. Para los valores del tipo *Integer* están disponibles las mismas operaciones que para el tipo *Int*.

Los cálculos con datos de tipo *Integer* son menos eficientes que con datos de tipo *Int*. El siguiente ejemplo calcula una potencia de valores de tipo *Integer*:

```
PRELUDE> 2 ↑ 100
1267650600228229401496703205376 :: Integer
```

2.2 - Tipos simples predefinidos

2.2.4. EL TIPO *Float*

Los valores de este tipo representan números reales (realmente se trata de un subconjunto de los números reales de cierto intervalo). Hay dos modos de escribir valores reales:

- La notación habitual: por ejemplo, 1.35, -15.345, 1.0 o 1.
- La notación científica: por ejemplo, $1.5e7$ (que denota el valor 1.5×10^7) o $1.5e-17$ (que denota el valor 1.5×10^{-17}).

FUNCIONES Y OPERADORES

Algunas de las funciones y operadores definidos para este tipo son:

- $(+), (*), (-), (/) :: Float \rightarrow Float \rightarrow Float$. Suma, producto, resta y división real.
- $(\wedge) :: Float \rightarrow Int \rightarrow Float$. Potencia de base real, pero exponente natural.
- $(**) :: Float \rightarrow Float \rightarrow Float$. Potencia de base y exponente real.
- *abs :: Float \rightarrow Float*. Valor absoluto.
- *signum :: Float \rightarrow Float*. Devuelve -1.0, 0.0 o 1.0 según el signo del argumento real.
- *negate :: Float \rightarrow Float*. Devuelve el valor del argumento real negado; puede usarse también el signo menos prefijo.
- *sin, asin, cos, acos, tan, atan :: Float \rightarrow Float*. Funciones trigonométricas (trabajan con radianes).
- *atan2 :: Float \rightarrow Float \rightarrow Float*, *atan2 x y* devuelve la arcotangente de $\frac{x}{y}$.
- *log, exp :: Float \rightarrow Float*. Funciones logarítmicas y exponenciales.
- *sqrt :: Float \rightarrow Float*. Raíz cuadrada.
- *pi :: Float*. El valor del número π .
- *truncate, round, floor y ceiling :: Float \rightarrow Integer o Float \rightarrow Int*. Funciones de redondeo; *truncate* elimina la parte decimal, *round* redondea al entero más próximo, *floor* devuelve el entero inferior y *ceiling* el superior.
- *fromInt :: Int \rightarrow Float* y *fromInteger :: Integer \rightarrow Float*. Funciones de conversión de tipo.

2.2.5. EL TIPO Double

Los valores de este tipo son números reales (realmente se trata de un subconjunto de los números reales de cierto intervalo). El subconjunto es mayor que el correspondiente al tipo *Float*, y las aproximaciones obtenidas con ellos son por ello más precisas. Todas las operaciones disponibles para el tipo *Float* están también disponibles para el tipo *Double*. Las primeras se denominan en precisión simple, las segundas, en precisión doble, y de ahí es de donde procede el nombre.

2.2.6. EL TIPO Char

Un valor de tipo *Char* representa un carácter (una letra, un dígito, un signo de puntuación, etc.). Un valor constante de tipo carácter se escribe entre comillas simples (se obtiene pulsando la tecla a la derecha del cero en el teclado español). Por ejemplo,

- 'a' denota la letra *a* minúscula.
- '1' denota el carácter correspondiente al dígito uno.
- '?' denota el carácter correspondiente al cierre de interrogación.

Algunos caracteres especiales se escriben precedidos por el carácter \:

- '\n' es el carácter de salto de línea.
- '\t' es el carácter tabulador.
- '\"' es el carácter comilla.
- '\"'' es el carácter comilla doble.
- '\\' es el carácter \.

FUNCIONES

Algunas de las funciones definidas² para este tipo son:

- *ord* :: *Char* → *Int*. Devuelve el código ASCII correspondiente al carácter argumento. El código ASCII hace corresponder a cada carácter un número único.
- *chr* :: *Int* → *Char*. Función inversa a *ord*.
- *isUpper*, *isLower*, *isDigit*, *isAlpha* :: *Char* → *Bool*. Comprueban si el carácter argumento es una letra mayúscula, minúscula, un dígito o una letra, respectivamente.
- *toUpper*, *toLower* :: *Char* → *Char*. Convierten la letra que toman como argumento en mayúscula o minúscula, respectivamente.

²Para usar estas funciones es necesario importar el módulo de biblioteca *Char*.

2.2 · Tipos simples predefinidos

El siguiente diálogo ilustra algunas de las funciones anteriores:

PRELUDE> *toUpper* 'a'
'A' :: *Char*

PRELUDE> *isUpper* 'a'
False :: *Bool*

PRELUDE> *ord* 'a'
97 :: *Int*

PRELUDE> *chr* 98
'b' :: *Char*

2.2.7. OPERADORES DE IGUALDAD Y ORDEN

Para todos los tipos básicos comentados están definidos los siguientes *operadores binarios*, que devuelven un valor booleano³:

(>)	mayor que	(≥)	mayor o igual que
(<)	menor que	(≤)	menor o igual que
(==)	igual a	(≠)	distinto de

El tipo de los dos argumentos para cualquier aplicación de los operadores anteriores debe ser el mismo (no se pueden comparar valores de tipos distintos). El siguiente diálogo muestra el comportamiento de estos operadores:

PRELUDE> *10 ≤ 15*
True :: *Bool*

PRELUDE> 'x' == 'y'
False :: *Bool*

PRELUDE> 'x' ≠ 'y'
True :: *Bool*

PRELUDE> *True* < 'a'
ERROR : Type error in application
*** Expression : *True* < 'a'
*** Term : *True*
*** Type : *Bool*
*** Does not match : *Char*

PRELUDE> 'b' > 'a'
True :: *Bool*

PRELUDE> *False* < *True*
True :: *Bool*

PRELUDE> (*1 < 5*) && (*10 > 9*)
True :: *Bool*

Para el tipo *Char* el orden viene dado por el código ASCII del carácter. Las letras mayúsculas tienen un código ASCII menor que las minúsculas. Dentro de cada grupo se respeta el orden alfabético. Las vocales acentuadas y la letra ñ aparecen fuera del orden habitual. En el caso del tipo *Bool*, el valor *False* se considera menor que *True*. La última línea del diálogo muestra que se produce un error de tipo si se intentan comparar dos valores con tipos distintos. Nótese que el operador de igualdad se escribe con dos caracteres (=); el símbolo = se usa para definir funciones.

Para todos los tipos que definen un orden están definidas las funciones *min* y *max* que devuelven el mínimo o máximo de dos valores respectivamente:

³Consultese la tabla inicial de este libro para conocer la sintaxis ASCII usada realmente en HASKELL para estos operadores.

```
PRELUDE> min 10 15
10 :: Integer
```

```
PRELUDE> max 10 15
15 :: Integer
```

Ejemplo 2.3 La siguiente función comprueba si su argumento está entre cero y nueve:

```
entre0y9 :: Integer → Bool
entre0y9 x = (0 ≤ x) && (x ≤ 9)
```

Ejemplo

2.3. CONSTRUCTORES DE TIPOS PREDEFINIDOS

Además de los tipos simples comentados, HASKELL define *tipos estructurados* que permiten representar colecciones de objetos.

2.3.1. TUPLAS

Una *tupla* es un dato compuesto donde el tipo de cada componente puede ser distinto. Los valores de tipo tupla se escriben separando con comas las distintas componentes y encerrando todas entre paréntesis. Una tupla tiene un tipo asociado; el tipo se escribe separando los tipos de las distintas componentes con comas y entre paréntesis:

Tuplas

Si v_1, v_2, \dots, v_n son valores con tipo t_1, t_2, \dots, t_n
entonces (v_1, v_2, \dots, v_n) es una tupla con tipo (t_1, t_2, \dots, t_n)

El siguiente diálogo muestra algunos valores con tipo tupla:

```
PRELUDE> ()
```

 $() :: ()$

```
PRELUDE> ('a', True)
('a', True) :: (Char, Bool)
```

```
PRELUDE> ('a', True, 1.5)
```

 $('a', True, 1.5) :: (Char, Bool, Double)$

Obsérvese que existe una tupla sin elementos, y que su tipo correspondiente se escribe de modo idéntico al valor.

Ejemplo 2.4 Las tuplas son útiles cuando una función ha de devolver más de un valor. La siguiente función devuelve el predecesor y el sucesor del argumento como una tupla:

```
predSuc :: Integer → (Integer, Integer)
predSuc x = (x - 1, x + 1)
```

Ejemplo

2.3 - Constructores de tipos predefinidos

2.3.2. LISTAS

Una *lista* es una colección de cero o más elementos todos del mismo tipo. Hay dos constructores (operadores que permiten construir valores) para listas:

- [] Representa la lista vacía (una lista sin elementos).
- (:) Permite añadir un elemento al principio de una lista. Si xs es una lista con n elementos, y x es un elemento, entonces $x : xs$ es una lista con $n + 1$ elementos. Recuérdese que todos los elementos de una lista han de ser del mismo tipo, por lo que se produce un error de tipo si el tipo de x no coincide con el de los elementos de la lista xs .

Si el tipo de todos los elementos de una lista es t , entonces el tipo de la lista se escribe $[t]$:

Listas

Si v_1, v_2, \dots, v_n son valores con tipo t
entonces $v_1 : (v_2 : (\dots (v_{n-1} : (v_n : [])) \dots))$ es una lista con tipo $[t]$

Algunos ejemplos de listas son:

- $1 : []$. Una lista que almacena un único entero; tiene tipo $[Integer]$.
- $3 : (1 : [])$. Una lista que almacena dos enteros; el valor 3 ocupa la primera posición dentro de la lista, el valor 1 la segunda.
- $'a' : (1 : [])$. Es una expresión errónea (produce un error de tipo) ya que todas las componentes de una lista han de tener el mismo tipo.

El constructor $(:)$ es asociativo a la derecha por lo que, en ausencia de paréntesis, el significado de la expresión es el mismo que si éstos aparecieran anidados a la derecha. Esto permite una notación más simple para representar listas no vacías⁴:

Asociatividad derecha de $(:)$

$x_1 : x_2 : \dots : x_{n-1} : x_n : [] \Rightarrow x_1 : (x_2 : (\dots (x_{n-1} : (x_n : [])) \dots))$

Aun así, la notación sigue siendo engorrosa, por lo que HASKELL permite una sintaxis para listas más cómoda: escribir los elementos entre corchetes y separados por comas:

⁴Para muchas construcciones del lenguaje, HASKELL proporciona notaciones sintácticas alternativas que lo único que introducen es un modo más cómodo de escribir la expresión. En los textos anglosajones esto se conoce como *syntactic sugar*. La forma más cómoda es traducida a la otra durante la compilación del programa, por lo que el uso de estas características no produce reducciones adicionales durante la ejecución del programa. Nosotros notaremos que una forma e_1 es interpretada como otra e_2 escribiendo $e_1 \Rightarrow e_2$.

Sintaxis para listas

$$[x_1, x_2, \dots, x_{n-1}, x_n] \Rightarrow x_1 : (x_2 : (\dots (x_{n-1} : (x_n : [])) \dots))$$

El siguiente diálogo muestra tres modos de escribir la misma lista. Puede observarse que HUGS siempre usa la última sintaxis al mostrar listas.

```
PRELUDE> 1 : (2 : (3 : []))
[1, 2, 3] :: [Integer]
```

```
PRELUDE> 1 : 2 : 3 : []
[1, 2, 3] :: [Integer]
```

CADENAS DE CARACTERES

Una *cadena de caracteres* es una secuencia de cero o más caracteres. Este tipo de datos es muy útil ya que permite trabajar con textos. En HASKELL, las cadenas de caracteres son listas de caracteres. El tipo asociado a las cadenas de caracteres es *String*. Este nombre de tipo es tan sólo un modo equivalente de escribir el tipo *[Char]*.

Por ejemplo, el valor `'h', 'o', 'l', 'a'` es una cadena de cuatro caracteres y tiene el tipo *[Char]* (o también *String*, que es lo mismo). Se permite una sintaxis más cómoda para escribir cadenas de caracteres: escribir el texto entre comillas dobles.

Cadenas de caracteres

$$"x_1 x_2 \dots x_{n-1} x_n" \Rightarrow ['x_1', 'x_2', \dots, 'x_{n-1}', 'x_n']$$

El siguiente diálogo muestra tres modos de escribir la misma cadena de caracteres:

```
PRELUDE> ['U', 'n', ' ', 'C', 'o', 'c', 'h', 'e']
"Un Coche" :: [Char]
```

```
PRELUDE> "U" : "n" : " " : "C" : "o" : "c" : "h" : "e" : []
"Un Coche" :: [Char]
```

2.3.3. EL CONSTRUCTOR DE TIPO (\rightarrow)

En un lenguaje funcional, como es de esperar, es posible declarar el tipo correspondiente a las distintas funciones. Para ello disponemos de un único constructor: (\rightarrow) .

Si $t_1, t_2, \dots, t_n, t_r$ son tipos válidos, entonces $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_r$ es el tipo de las funciones que toman n argumentos (cada uno con tipo t_i , $1 \leq i \leq n$) y devuelven un resultado con tipo t_r :

Tipos funcionales

Si $t_1, t_2, \dots, t_n, t_r$ son tipos válidos entonces $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_r$ es el tipo de una función con n argumentos

2.3 - Constructores de tipos predefinidos

Ejemplo 2.5 La función *inc* toma un argumento de tipo entero y devuelve el valor que se obtiene al incrementarlo:

```
inc :: Integer → Integer
inc x = x + 1
```

Ejemplo

Ejemplo 2.6 La siguiente función comprueba si su argumento entero es un cero:

```
esCero :: Integer → Bool
esCero x = x == 0
```

Ejemplo

Ejemplo 2.7 La función *sumaCuadrados* suma los cuadrados de sus dos argumentos:

```
sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados x y = x ↑ 2 + y ↑ 2
```

Ejemplo

Ejemplo 2.8 El argumento o resultado de una función puede ser otra función, dando lugar a lo que se denomina *funciones de orden superior* (véase el Capítulo 3):

```
componer :: (Integer → Integer) → (Integer → Integer)
           → Integer → Integer
componer g f x = g (f x)
```

Es importante resaltar que los paréntesis no pueden ser eliminados en el tipo de *componer*. Un ejemplo de uso de *componer* es el siguiente⁵:

```
MAIN> componer inc inc 10
12 :: Integer
```

que se reduce del siguiente modo:

```
componer inc inc 10
⇒ { sustituyendo en la definición de componer }
inc (inc 10)
⇒ { por definición de inc }
(inc 10) + 1
⇒ { por definición de inc }
(10 + 1) + 1
⇒ { por definición de (+) }
11 + 1
⇒ { por definición de (+) }
12
```

Ejemplo

⁵El operador predefinido `(.)` permite componer funciones de un modo más general que la función definida en el ejemplo.

2.4. COMENTARIOS

Se considera buena práctica de programación incluir *comentarios* a la hora de escribir un programa. Un comentario es un texto que acompaña al programa pero que es ignorado por el evaluador.

Hay dos modos de incluir comentarios en un programa:

- Comentarios de una sola línea: comienzan por dos guiones consecutivos (--) y abarcan hasta el final de la línea actual:

```
f :: Integer → Integer
f x = x + 1 -- Esto es un comentario
```

- Comentarios que abarcan más de una línea: Comienzan por los caracteres {– y acaban con –}. Pueden abarcar varias líneas y anidarse:

```
{- Esto es un comentario
de más de una línea -}
g :: Integer → Integer
g x = x - 1
```

2.5. OPERADORES

En HASKELL podemos definir, además de funciones, *operadores*. Se trata de funciones con dos argumentos cuyo nombre es simbólico (una cadena de signos, no letras) y que pueden ser invocados de forma *infija* (entre sus dos argumentos). Ya hemos visto algunos de los operadores predefinidos como &&, ||, +, -, *, / y !.

El programador puede definir sus propios operadores. Como identificador puede utilizar uno o más de los siguientes símbolos:

```
: ! # $ % & * + . / < = > ? @ \ | - ~
```

Los operadores que comienzan por el carácter dos puntos (:) tienen un significado especial: son *constructores infijos de datos*. Un constructor de datos es usado para construir valores de un tipo. Estudiaremos esta característica en el Capítulo 4.

Algunos ejemplos de operadores válidos son⁶:

```
+ ++ && || ≤ = ≠ . // $
% @@ + \ / \> ?
```

Los que aparecen en la primera línea están predefinidos. Algunas combinaciones de símbolos son *claves* (o *reservados*) del lenguaje y no pueden ser usados por el programador como nombres de nuevos operadores; éstos son

⁶ Consultese la tabla inicial de este libro para conocer la sintaxis ASCII de estos operadores.

: → ⇒ : .. = ⊗ \ | ← ~

A la hora de definir un operador podemos indicar su *prioridad* (un número entero entre 0 y 9; 10 es la prioridad máxima pero está reservada para la aplicación de funciones) y su *asociatividad*. Esto se consigue con las cláusulas infix:

infix [prioridad] [identificador operador] (define un operador no asociativo)
 infixl [prioridad] [identificador operador] (define un operador asociativo a la izquierda)
 infixr [prioridad] [identificador operador] (define un operador asociativo a la derecha)

Para declarar el tipo de un operador hay que escribir el identificador de éste entre paréntesis. En la parte izquierda de la definición del cuerpo del operador se puede usar notación infija (el operador aparece entre sus dos argumentos).

Ejemplo 2.9 Podemos definir un operador (~=) para ver si dos valores reales son aproximadamente iguales⁷:

```
infix 4 ~=
(~=) :: Float → Float → Bool
x ~= y = abs(x - y) < 0.0001
```

El operador ha sido declarado no asociativo y con prioridad 4. Como muestra el ejemplo, es obligatorio escribir el operador entre paréntesis en la declaración de tipo correspondiente. Además, como muestra la siguiente sesión con HUGS:

```
MAIN> 1.0 ~= 2.0
False :: Bool
```

```
MAIN> 1.0 ~= 1.00001
True :: Bool
```

el operador se puede usar de modo infijo.

Ejemplo

Obsérvese que el programador no puede definir operadores unarios (de sólo un argumento). El único operador unario en HASKELL es el cambio de signo o negación representado con el signo – prefijo:

```
PRELUDE> -(7 + 2)
-9 :: Integer
```

Las prioridades y asociatividades de los operadores predefinidos aparecen en la Figura 2.1, que está extraída del PRELUDE.

Si en una expresión aparecen operadores con prioridades distintas, se agrupan primero aquéllos con mayor prioridad, en ausencia de paréntesis. Por ejemplo, la expresión 1 + 2 * 4 es interpretada como 1 + (2 * 4) ya que el operador (*) tiene mayor prioridad (7) que el operador (+) (6).

⁷ En otros lenguajes de programación como MATLAB, este operador es el predefinido para la desigualdad.

```

infixr 9 .
infixl 9 !!
infixr 8 !!, !!, ++
infixl 7 *, /, `quot`, `rem`, `div`, `mod`
infixl 6 +, -
infixr 5 :, +
infix 4 ==, !=, <, ≤, ≥, >, `elem`, `notElem`
infixr 3 &&
infixr 2 ||
infixl 1 >>, >>=
infixr 1 =<=
infixr 0 $, $!, `seq`
```

Figura 2.1: Asociatividad y prioridad de los operadores predefinidos.

Se pueden utilizar paréntesis si la interpretación deseada es otra: $(1 + 2) * 4$, o para aclarar el significado. La llamada a una función (aplicación de uno o más argumentos a una función) tiene prioridad máxima (10), por lo que la expresión *cuadrado* $3 + 4$ donde *cuadrado* es:

```

cuadrado :: Integer → Integer
cuadrado x = x * x
```

es interpretada como $(\text{cuadrado } 3) + 4$. Si lo que queremos es calcular el *cuadrado* de $3+4$ hemos de usar paréntesis: *cuadrado* $(3 + 4)$. La regla general es muy simple: sólo los argumentos compuestos (los que son expresiones, no valores) han de ir entre paréntesis. Los paréntesis también son obligatorios al pasar operadores como argumentos a funciones de orden superior.

La prioridad no aclara qué ocurre cuando un mismo operador aparece varias veces en una expresión. ¿Cuál es el valor de la expresión $8 - 5 - 1$? Puede ser $((8 - 5) - 1)$ o $(8 - (5 - 1))$. La asociatividad aclara el significado de la expresión en este caso. Dado que la diferencia ($-$) está definida como asociativa a la izquierda, la interpretación correcta es la primera.

Si \otimes es un operador asociativo a la izquierda:

$$x_1 \otimes x_2 \otimes \cdots \otimes x_{n-1} \otimes x_n \Rightarrow (\dots (x_1 \otimes x_2) \otimes \cdots \otimes x_{n-1}) \otimes x_n$$

Si \otimes es un operador asociativo a la derecha:

$$x_1 \otimes x_2 \otimes \cdots \otimes x_{n-1} \otimes x_n \Rightarrow x_1 \otimes (x_2 \otimes \cdots \otimes (x_{n-1} \otimes x_n) \dots)$$

Si un operador es no asociativo no puede ser utilizado más de una vez en una expresión sin aclarar el significado con paréntesis. Por ejemplo, si definimos el siguiente operador de división como no asociativo:

2.5 - Operadores

```

infix 7 /\
(/\) :: Integer → Integer → Integer
x /\ y = div x y
```

obtenemos el siguiente diálogo:

```

MAIN> 7 /\ 4 /\ 2
ERROR: Ambiguous use of operator "/\" with "/\"
```

```

MAIN> (7 /\ 4) /\ 2
0 :: Integer
```

```

MAIN> 1 < 2 == True
ERROR: Ambiguous use of operator "<" with "=="
```

En HASKELL, todo operador toma por defecto (si no se usan las cláusulas *infixl* o *infixr* en la declaración correspondiente) prioridad 9 y asociatividad izquierda. Sin embargo, es recomendable incluir la cláusula siempre que se declare un operador.

2.5.1. OPERADORES FRENTE A FUNCIONES

La principal diferencia entre un operador y una función de dos argumentos es que los primeros se usan de modo infijo (entre sus argumentos) mientras que las funciones se usan de modo prefijo (preceden a sus argumentos). Los dos conceptos son tan parecidos que HASKELL permite convertir cualquier operador en una función de dos argumentos. Para ello basta con escribir el nombre del operador entre paréntesis. Así, el operador definido previamente puede ser usado como una función del siguiente modo:

```

MAIN> (~=) 1.0 2.0
False :: Bool
```

```

MAIN> (~=) 1.0 1.00001
True :: Bool
```

Los paréntesis son obligatorios en los ejemplos anteriores. También se puede usar notación prefija al declarar el operador. La definición del operador $(\sim=)$ puede realizarse del siguiente modo:

```

infix 4 ~=
(~=) :: Float → Float → Bool
(~=) x y = abs(x - y) < 0.0001
```

Cualquier operador puede usarse tanto de modo prefijo como infijo, independientemente del modo en que esté declarado. Por ejemplo, la suma puede también ser usada de modo prefijo:

```

PRELUDE> (+) 3 4
7 :: Integer
```

Al igual que un operador puede convertirse en una función, una función de dos argumentos puede convertirse en un operador si se escribe entre acentos franceses (el carácter ‘`’ en la tecla a la derecha de la tecla P en el teclado español). Por ejemplo, la función:

```
suma :: Integer → Integer → Integer
suma x y = x + y
```

puede también ser declarada como:

```
suma :: Integer → Integer → Integer
x `suma` y = x + y
```

En cualquiera de los dos casos se puede usar de modo prefijo o infijo:

```
MAIN> suma 1 2
3 :: Integer
```

```
MAIN> 1 `suma` 2
3 :: Integer
```

También se puede dar una prioridad y asociatividad al uso infijo de cualquier función:

```
infixl 6 `suma`
```

Esta declaración indica que la función *suma* tendrá prioridad 6 y asociatividad izquierda cuando se use de modo infijo.

Algunos detalles importantes referentes a la sintaxis de operadores y funciones son los siguientes:

- En una declaración de tipo, lo que aparece antes del signo `::` debe ser el nombre de una función. Por ello, al declarar el tipo de un operador es necesario escribir el nombre de éste entre paréntesis para convertirlo en una función.
- Las cláusulas `infix`, `infixl` y `infixr` van seguidas de un número de prioridad y un nombre de operador. Por ello, cuando se declara la asociatividad de una función de dos argumentos es necesario convertir ésta en operador escribiendo su nombre entre acentos franceses. Los nombres de operadores no pueden ir entre paréntesis ya que serían funciones.
- Una función puede aceptar como argumento otra función. Este tipo de funciones se llaman *de orden superior* (ver el Capítulo 3). Es posible pasar un operador como parámetro, pero hay que convertirlo previamente en función escribiéndolo entre paréntesis.

2.6. COMPARACIÓN DE PATRONES

En la mayoría de los lenguajes de programación, los parámetros formales de una función sólo pueden ser nombres de variables. En HASKELL hay además otras posibilidades: un argumento puede ser un *patrón*.

Es posible definir una función dando más de una ecuación para ésta. Cada ecuación define el comportamiento de la función para distintas formas del argumento, y los patrones permiten capturar dicha forma. Al aplicar la función a un parámetro concreto la *comparación de patrones* determina la ecuación a utilizar. Por tanto, el uso de patrones permite modelar cómo se evaluará una llamada a una función a partir de la forma de sus argumentos.

2.6.1. PATRONES CONSTANTES

Un *patrón constante* puede ser un número, un carácter o un constructor de datos (ver Sección 4.2). Con un patrón constante sólo *unifica* (o concuerda) un argumento que coincida con dicha constante.

Ejemplo 2.10 La siguiente función:

```
f :: Integer → Bool
f 1 = True
f 2 = False
```

toma un valor entero y devuelve cierto cuando el argumento es 1 o falso si el argumento es 2. Como puede apreciarse en la siguiente sesión:

```
MAIN> f 1
True :: Bool
```

```
MAIN> f 3
Program error : {f 3}
```

```
MAIN> f 2
False :: Bool
```

la función está parcialmente definida (su conjunto inicial es *Integer*, pero el dominio es tan sólo {1,2}). Para cualquier otro entero la función no está definida (HttoS lo indica con un error). Si queremos definir totalmente la función podemos añadir otra ecuación, por ejemplo:

```
f :: Integer → Bool
f 1 = True
f 2 = False
f x = True
```

El resultado es ahora cierto para cualquier valor distinto a 2:

```
MAIN> f 3
True :: Bool
```

Para reducir una aplicación de una función definida mediante patrones se van probando las ecuaciones en el orden en que aparecen en la definición, hasta encontrar una que unique con el argumento, y su cuerpo es el que se utiliza para reducir la expresión. Como podemos ver, el orden de las ecuaciones es importante. En una definición como:

```
f :: Integer → Bool
f 1 = True
f 1 = False
f x = True
```

la expresión *f 1* siempre será reducida a *True*. Si permitiésemos que la elección fuese arbitraria se perdería la transparencia referencial, ya que una misma expresión (*f 1*) podría tener dos valores distintos.

Ejemplo 2.11 La definición de la conjunción y disyunción de valores lógicos usa patrones constantes (*True* y *False* son dos constructores de datos para el tipo *Bool*):

```
infixr 3 &&
(&&) :: Bool -> Bool -> Bool
False && x = False
True && x = x

infixr 2 ||
(||) :: Bool -> Bool -> Bool
False || x = x
True || x = True
```

Ejemplo

Cuando una función tiene más de un argumento:

- Se comprueban los patrones correspondientes a las distintas ecuaciones en el orden dado por el programa, hasta que se encuentre una que unifique.
- Dentro de una misma ecuación se intentan unificar los patrones correspondientes a los argumentos de izquierda a derecha.
- En cuanto un patrón falla para un argumento, se pasa a la siguiente ecuación.

Por ejemplo, la expresión *True && True* es reducida al valor *True*, ya que se selecciona la segunda ecuación de la definición de *&&*.

La función que se está definiendo puede aparecer como parte de la expresión resultado. A esto se llama *recursividad*. El uso de patrones y de la recursividad permite dar definiciones de funciones muy compactas y elegantes.

Ejemplo 2.12 La función *fact* calcula recursivamente el factorial de un número natural:

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

Así, la expresión *fact 3* se reduce del siguiente modo⁸:

```
fact 3
=> { segunda ecuación de fact (n -> 3) }
  3 * fact (3 - 1)
=> { definición de (-) }
  3 * fact 2
=> { segunda ecuación de fact (n -> 2) }
  3 * (2 * fact (2 - 1))
=> { definición de (-) }
```

⁸ Cuando sea conveniente, indicaremos que una variable *v* que aparece en un patrón queda ligada a una expresión *e* con *v -> e*.

2.6 - Comparación de patrones

```
3 * (2 * fact 1)
=> { segunda ecuación de fact (n -> 1) }
  3 * (2 * (1 * fact (1 - 1)))
=> { definición de (-) }
  3 * (2 * (1 * fact 0))
=> { primera ecuación de fact }
  3 * (2 * (1 * 1))
=> { definición de (+) }
  3 * (2 * 1)
=> { definición de (+) }
  3 * 2
=> { definición de (+) }
  6
```

Ejemplo

Obsérvese que en todas las reducciones de *fact* excepto en la última se usa la segunda ecuación de la definición de *fact*. El mecanismo de unificación de patrones es simple: en la primera reducción se unifican las expresiones *fact 3* y *fact n*, de donde se obtiene la sustitución *{n -> 3}*, que al ser aplicada al cuerpo de la segunda ecuación de *fact* produce el resultado de la primera reducción: *3 * fact (3 - 1)*. Nótese también que es necesario realizar las diferencias para ver con qué ecuación de *fact* unifica el argumento; se dice en este caso que la función *fact* es *estricta* en su argumento, ya que éste debe estar en forma normal.

Para que una definición recursiva sea efectiva hay que garantizar que:

- Exista, al menos, un caso no recursivo. Es lo que se denomina un *caso base*. El caso base para la función *fact* es cuando el argumento es cero.
- En cada llamada recursiva, el valor del argumento ha de "aproximarse" al caso base. Esta condición se cumple en la función *fact* si es utilizada con argumentos naturales, ya que *n - 1* es un valor más próximo a cero que *n*. Sin embargo, la condición no se cumple para argumentos negativos, por lo que en ese caso no sería posible garantizar que la reducción terminase.
- Hay que asegurar que el caso base se alcance, lo cual garantiza que la reducción acabará. Esta condición se cumple en la definición de *fact*, ya que si decrementamos repetidamente un número natural, acabamos alcanzando el valor cero, siempre que partamos de un número positivo.

Si una de las condiciones anteriores falla la reducción no acabará. Por ejemplo, si hubiésemos definido la función como:

```
fact' :: Integer -> Integer
fact' 0 = 1
fact' n = n * (n - 1) * fact' (n - 2)
```

la tercera condición no se cumpliría y la reducción no acabaría para argumentos impares:

```
fact' 3
=> { segunda ecuación de fact' y (-) }
```

```

3 * 2 * fact' 1
=> { segunda ecuación de fact' y (-) }
  3 * 2 * (1 * 0 * fact' (-1))
=> { segunda ecuación de fact' y (-) }
  3 * 2 * (1 * 0 * ((-1) * (-2) * fact' (-3)))
=>
...

```

2.6.2. PATRONES PARA LISTAS

Es posible utilizar patrones al definir funciones que trabajen con listas. Los patrones para listas toman las siguientes formas:

- `[]` sólo unifica con un argumento que sea una lista vacía.
- `[x]`, `[x, y]`, etc. sólo unifican con listas de uno, dos, etc. argumentos.
- `(x : xs)` unifica con listas con al menos un elemento; `x` queda ligada a la *cabeza* (primer elemento de la lista), `xs` queda ligada a la *cola* (la lista argumento sin el primer elemento). Así, el resultado de unificar la lista `[1, 2, 3]` con el patrón `(x : xs)` es el par de sustituciones `(x ← 1, xs ← [2, 3])`. También se puede usar `(x : y : xs)`, `(x : y : v : xs)`, etc. para listas de al menos dos, tres, etc. elementos.

Ejemplo 2.13 La siguiente función toma una lista de valores enteros y los suma:

```

suma      :: [Integer] → Integer
suma []   = 0
suma (x : xs) = x + suma xs

```

-- caso base
-- caso recursivo

Las dos ecuaciones hacen que la función esté definida para cualquier lista. La primera será utilizada si la lista está vacía, mientras que la segunda se usará en otro caso. Tenemos la siguiente reducción:

```

suma [1, 2, 3]
=> { sintaxis de listas }
  suma (1 : (2 : (3 : [])))
=> { segunda ecuación de suma (x ← 1, xs ← 2 : (3 : [])) }
  1 + suma (2 : (3 : []))
=> { segunda ecuación de suma (x ← 2, xs ← 3 : []) }
  1 + (2 + suma (3 : []))
=> { segunda ecuación de suma (x ← 3, xs ← []) }
  1 + (2 + (3 + suma []))
=> { primera ecuación de suma }
  1 + (2 + (3 + 0))
=> { definición de (+) tres veces }
  6

```

Obsérvese que cada vez que se aplica la segunda ecuación, la variable `x` queda ligada al primer elemento de la lista y `xs` al resto de ésta. Cuando la lista tiene un único elemento se aplica la segunda ecuación, aunque `xs` queda ligada a la lista vacía.

Tipos

Ejercicio 2.14 Define una función:

`aEntero :: [Integer] → Integer`

que transforme una lista de dígitos en el correspondiente valor entero:

```

MAIN> aEntero [2, 3, 4]
234 :: Integer

```

Defina la función recíproca `aLista`:

```

MAIN> aLista 234
[2, 3, 4] :: [Integer]

```

2.6.3. PATRONES PARA TUPLAS

También puede usarse la sintaxis de tuplas en patrones.

Ejemplo 2.15 Las siguientes funciones permiten seleccionar el primer elemento de tuplas con dos y tres componentes enteras:

```

primero2      :: (Integer, Integer) → Integer
primero2 (x, y) = x
primero3      :: (Integer, Integer, Integer) → Integer
primero3 (x, y, z) = x

```

El siguiente diálogo muestra el uso de estas funciones:

MAIN> primero2 (5, 8)	MAIN> primero2 (5, 8, 7)
5 :: Integer	ERROR : Type error in application
MAIN> primero3 (5, 8, 7)	*** Expression : primero2 (5, 8, 7)
5 :: Integer	*** Term : (5, 8, 7)
	*** Type : (a, b, c)
	*** Does not match : (Int, Int)

Obsérvese que en el último caso se ha producido un error de tipo ya que la función `primero2` está definida para pares y el argumento es una terma.

Tipos

Ejemplo 2.16 Los patrones pueden anidarse. La siguiente función suma todos los elementos almacenados en una lista de pares:

```

sumaPares      :: [(Integer, Integer)] → Integer
sumaPares []   = 0
sumaPares ((x, y) : xs) = x + y + sumaPares xs

```

Por ejemplo

```

sumaPares [(1, 2), (3, 4)]
⇒ { sintaxis de listas }
  sumaPares ((1, 2) : ((3, 4) : []))
⇒ { segunda ecuación de sumaPares (x ← 1, y ← 2, xs ← (3, 4) : []) }
    1 + 2 + sumaPares ((3, 4) : [])
⇒ { segunda ecuación de sumaPares (x ← 3, y ← 4, xs ← []) }
    1 + 2 + (3 + 4 + sumaPares [])
⇒ { primera ecuación de sumaPares }
    1 + 2 + (3 + 4 + 0)
⇒ { definición de (+) cuatro veces }
    10

```

El patrón $((x, y) : xs)$ concuerda con una lista de pares no vacía, unificando x con la primera componente del par en la cabeza de la lista, y con la segunda componente y xs con la lista de pares que se obtiene al eliminar el primer par.

Ejemplo

2.6.4. PATRONES ARITMÉTICOS

Un patrón de la forma $(n + k)$, donde k es una constante natural, sólo unifica con un argumento que sea un número entero mayor o igual que k y asocia a la variable n el valor de dicho número menos k .

Ejemplo 2.17 Podemos definir la función factorial del siguiente modo:

```

factorial      :: Integer → Integer
factorial 0     = 1
factorial (n + 1) = (n + 1) * factorial n

```

La reducción de $\text{factorial } 2$ es:

```

factorial 2
⇒ { segunda ecuación de factorial (n ← 1) }
  (1 + 1) * factorial 1
⇒ { definición de (+) }
  2 * factorial 1
⇒ { segunda ecuación de factorial (n ← 0) }
  2 * ((0 + 1) * factorial 0)
⇒ { definición de (+) }
  2 * (1 * factorial 0)
⇒ { primera ecuación de factorial }
  2 * (1 * 1)
⇒ { definición de (*) dos veces }
  2

```

Nótese que se aplica la segunda ecuación mientras el argumento sea mayor o igual que 1. Cada vez que se aplica dicha ecuación, n queda ligada al argumento menos 1.

Ejemplo

2.6 - Comparación de patrones

Hay una diferencia sutil que suele pasar inadvertida entre la definición anterior y la vista en el Ejemplo 2.12:

```

fact   :: Integer → Integer
fact 0  = 1
fact n  = n * fact (n - 1)

```

Si se invoca a la función fact con un argumento negativo, el cálculo de la expresión no acaba, ya que se produce una secuencia infinita de aplicaciones de la segunda ecuación. Sin embargo, la función factorial está parcialmente definida (no está definida para argumentos negativos), ya que ninguno de los dos patrones unifica con un número negativo, y da lugar al correspondiente error inmediatamente.

2.6.5. PATRONES NOMBRADOS O SEUDÓNIMOS

A veces un patrón compuesto que aparece en la parte izquierda de una ecuación debe volver a aparecer en la parte derecha de ésta. En ese caso es posible utilizar un seudónimo o *patrón alias* para nombrar dicho patrón, y utilizar el seudónimo en vez del patrón compuesto en la parte derecha de la definición.

Ejemplo 2.18 En la definición:

```

factorial      :: Integer → Integer
factorial 0     = 1
factorial (n + 1) = (n + 1) * factorial n

```

el patrón $(n + 1)$ vuelve a aparecer en la parte derecha de la segunda ecuación. Es posible utilizar un seudónimo del siguiente modo:

```

factorial''     :: Integer → Integer
factorial'' 0    = 1
factorial'' m@(n + 1) = m * factorial'' n

```

El cualificador $@$ en la segunda ecuación asigna el seudónimo m al patrón $(n + 1)$. Así, la variable m queda asociada con el valor del argumento, mientras que n lo hace con el valor del argumento menos uno. Por ejemplo, la reducción de $\text{factorial}'' 2$ es:

```

factorial'' 2
⇒ { segunda ecuación de factorial'' (n ← 1, m ← 2) }
  2 * factorial'' 1
⇒ { segunda ecuación de factorial'' (n ← 0, m ← 1) }
  2 * (1 * factorial'' 0)
⇒ { primera ecuación de factorial'' }
  2 * (1 * 1)
⇒ { definición de (*) dos veces }
  2

```

Ejemplo

Como puede observarse, los seudónimos pueden mejorar ligeramente la eficiencia de una función; se utiliza el valor del argumento directamente cada vez que se usa la segunda ecuación, lo cual evita recalcular el argumento en cada llamada recursiva (comárese con la reducción de *factorial 2* en la sección anterior). Además se simplifica la escritura, con lo que disminuye la probabilidad de que el programador cometa errores tipográficos.

2.6.6. EL PATRÓN SUBRAYADO

Un *patrón subrayado* (_) unifica con cualquier argumento pero no establece ligadura alguna. Puede usarse cuando parte del argumento no aparece en el cuerpo de la función.

Ejemplo 2.19 La función que calcula el número de elementos de una lista de enteros:

```
longitud      :: [Integer] → Integer
longitud []    = 0
longitud (x : xs) = 1 + longitud xs
```

puede ser escrita usando un patrón subrayado:

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

ya que la variable *x* no se usa en la parte izquierda de la ecuación.

Ejemplo

El significado de un patrón subrayado es idéntico al de una variable, aunque permite que no haya que nombrar argumentos que luego no se usan. Se considera buen estilo de programación utilizar un patrón subrayado cuando un argumento o parte de éste no se usa en el cuerpo de la función. Así también se evitan errores tipográficos.

2.6.7. ERRORES COMUNES

En HASKELL un nombre de variable no puede aparecer repetido en la parte izquierda (a la izquierda del signo igual) de una misma ecuación:

```
sonIguales :: Integer → Integer → Bool
sonIguales x x = True   -- No es correcto: x repetida
sonIguales x y = False
```

En su lugar hay que escribir⁹:

```
sonIguales :: Integer → Integer → Bool
sonIguales x y = (x == y)
```

No es un error repetir el patrón subrayado:

⁹ Esta advertencia es especial para aquellos que conocen el lenguaje PROLOG.

2.6 - Comparación de patrones

siempreVerdad :: Integer → Integer → Bool
siempreVerdad _ _ = True

El tipo de todas las ecuaciones correspondientes a la definición de una función debe ser el mismo (más exactamente deben ser tipos *unificables*, como veremos posteriormente). La siguiente definición de función no es válida:

f 0 = 0
f True = 2 -- Error: Bool e Integer son dos tipos distintos

ya que no sería posible asignar un tipo al argumento de *f*.

2.6.8. PATRONES Y EVALUACIÓN PEREZOSA

La unificación determina qué ecuación es seleccionada para reducir una expresión a partir de la forma de los argumentos. Por ejemplo, consideremos la siguiente función que permite comprobar si la lista que se le pasa como argumento es vacía:

```
esVacia      :: [a] → Bool
esVacia []    = True
esVacia (_ : _) = False
```

Para reducir una expresión como *esVacia ls* hay que saber si *ls* es una lista vacía o no. Podría pensarse en obtener primero la forma normal de *ls* y con este resultado determinar la ecuación apropiada de *esVacia*, pero sería poco consistente con el principio de evaluación perezosa, que intenta evaluar lo mínimo para obtener el resultado. Además, la evaluación del argumento ni siquiera acabaría si la lista *ls* fuese infinita como

```
infinita :: [Integer]
infinita = 1 : infinita
```

La solución adoptada en HASKELL consiste en evaluar un argumento hasta obtener un número de constructores suficiente para resolver el patrón, sin obtener necesariamente su forma normal. En este caso basta con obtener el constructor más externo ([] o (:)). Una vez conocido este constructor es posible determinar la ecuación a utilizar. Por ejemplo:

```
esVacia infinita
⇒ { definición de infinita }
esVacia (1 : infinita)
⇒ { segunda ecuación de esVacia }
False
```

Obsérvese que en un caso como:

```
noAcaba :: Integer
noAcaba = 1 + noAcaba
infinita' :: [Integer]
infinita' = noAcaba : infinita'
```

ni siquiera la cabeza de la lista tiene que ser evaluada:

```
esVacia infinita'
→ (definición de infinita')
esVacia (noAcaba : infinita')
→ [segunda ecuación de esVacia ]
False
```

2.7. EXPRESIONES case

Es posible realizar una comprobación de patrones en cualquier punto de una expresión. Para ello puede usarse la construcción `case`. La sintaxis de esta construcción es:

```
case expr of
  patrón1 → resultado1
  patrón2 → resultado2
  ...
  patrónn → resultadon
```

Todos los patrones han de ser del mismo tipo, el cual debe coincidir con el de `expr`. Todos los resultados deben ser del mismo tipo, que es el del resultado de la construcción.

Ejemplo 2.20 La siguiente función calcula la longitud de una lista de enteros usando una expresión `case`:

```
long :: [Integer] → Integer
long ls = case ls of
  [] → 0
  _ : xs → 1 + long xs
```

2.8. LA FUNCIÓN error

Si intentamos evaluar una función parcial en un punto en el cual no está definida, se produce un error y HUGS lo muestra:

```
cabeza :: [Integer] → Integer
cabeza (x : _) = x

MAIN> cabeza []
Program error : {cabeza []}
```

Existe una función predefinida que permite al programador terminar la evaluación de una expresión y mostrar un mensaje por pantalla. La función `error` toma como argumento una cadena de caracteres. Siempre que se intente reducir una aplicación de la función `error` la reducción terminará prematuramente y se mostrará la cadena por pantalla.

2.9 - FUNCIONES A TROZOS

Ejemplo 2.21 Definiendo la función `cabeza` del siguiente modo:

```
cabeza' :: [Integer] → Integer
cabeza' [] = error "cabeza' de lista vacía no definida"
cabeza' (x : _) = x
```

cuando se aplica esta función a una lista vacía:

MAIN> <code>cabeza'</code> [1, 2, 3]	MAIN> <code>cabeza'</code> []
<code>1 :: Integer</code>	<code>Program error : cabeza' de lista vacía no definida</code>

el intérprete muestra el mensaje “`cabeza'` de lista vacía no definida”.

■ Ejemplo

La función `error` permite controlar casos sin sentido al definir una función, emitiendo el correspondiente mensaje por pantalla si se alcanzan. Semánticamente se considera que el valor devuelto por la función es el valor especial \perp (leido bottom) que pertenece a cualquier tipo. Esto es posible ya que la función `error` es polimórfica. El valor \perp denota semánticamente otro tipo de errores, como la no terminación en la reducción de una expresión, luego es el valor de las expresiones que no tienen forma normal.

2.9. FUNCIONES A TROZOS

En las funciones *a trozos* o *por partes* el resultado depende de cierta condición, como

$$\text{absoluto} :: \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\text{absoluto}(x) = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{si } x < 0 \end{cases}$$

Ejemplo 2.22 En HASKELL la definición anterior puede escribirse del siguiente modo:

```
absoluto :: Integer → Integer
absoluto x
| x ≥ 0 = x
| x < 0 = -x
```

■ Ejemplo

El sangrado de la expresión anterior es obligatorio: las dos últimas líneas de la definición han de estar un poco más a la derecha y ambas han de estar a la misma altura. Esta función consta de una única ecuación. Cada una de las expresiones entre los símbolos `|` y `=` es una *guarda* y han de ser expresiones con tipo *Bool*. Al reducir una aplicación de esta función se evalúan las guardas en el orden textual hasta encontrar una cuya valor sea *True*, y se devuelve como resultado la expresión a la derecha de dicha guarda:

```
MAIN> absoluto (-10)
10 :: Integer
```

Ejemplo 2.23 La siguiente función con guardas devuelve -1, 0 o 1 dependiendo del signo de su argumento:

```
signo :: Integer → Integer
signo x
```

$$\begin{cases} x > 0 = 1 \\ x = 0 = 0 \\ x < 0 = -1 \end{cases}$$

Es posible utilizar como guarda la palabra *otherwise*, que es equivalente al valor *True*, y ésta suele utilizarse para devolver un resultado en caso de que no se cumplan ninguna de las guardas previas. La función anterior puede también ser definida como:

```
signo :: Integer → Integer
```

$$\begin{cases} signo x \\ | x > 0 = 1 \\ | x = 0 = 0 \\ | otherwise = -1 \end{cases}$$

Ejemplo

2.10. EXPRESIONES CONDICIONALES

Otro modo de escribir expresiones cuyo resultado dependa de cierta condición es:

```
if exprBool then exprSi else exprNo
```

Para evaluar esta *expresión condicional* se procede del siguiente modo:

1. Se evalúa *exprBool*, que debe ser una expresión de tipo booleano.
2. Si el valor de ésta es *True*, el valor de la expresión es el de *exprSi*.
3. En otro caso, el valor de la expresión es el de *exprNo*.

Los tipos de *exprSi* y *exprNo* pueden ser arbitrarios, pero deben coincidir. El tipo del resultado de la expresión completa es el de *exprSi* (el mismo que el de *exprNo*).

Ejemplo 2.24 Podemos usar una expresión condicional para definir una función:

```
maxEnt :: Integer → Integer → Integer
maxEnt x y = if x > y then x else y
```

que devuelve el máximo de dos números enteros.

Ejemplo

Al evaluar una expresión condicional, la expresión booleana siempre se evalúa. HASKELL es un lenguaje perezoso (sólo se evalúa lo necesario para conocer el resultado de una expresión): si se determina que el valor de la expresión booleana es *True* no se evaluará *exprNo*, y si su valor es *False* no se evaluará *exprSi*:

```
PRELUDE> if 5 > 2 then 10.0 else (10.0/0.0)
10.0 :: Double
```

```
PRELUDE> if 5 < 2 then 10.0 else (10.0/0.0)
Program error : {primDivFloat 10.0 0.0}
```

Obsérvese que la división por cero no se efectuó en el primer ejemplo. Se dice que la expresión condicional es *estricta* en su primer argumento (siempre se evalúa) pero no lo es en el segundo ni en el tercero (puede que no se evalúen).

Al ser el condicional una expresión, puede usarse como parte de otras expresiones:

```
PRELUDE> 2 + if 'a' < 'z' then 10 else 4
20 :: Integer
```

2.11. DEFINICIONES LOCALES

A menudo es conveniente dar un nombre a una subexpresión que aparece varias veces en otra y utilizar este nombre en lugar de la expresión.

Ejemplo 2.25 Considérese la siguiente función que calcula las raíces de una ecuación de segundo grado a partir de sus coeficientes, sólo si éstas son reales:

```
raíces :: Float → Float → Float → (Float, Float)
raíces a b c
| b ↑ 2 - 4 * a * c ≥ 0 = ((-b + sqrt(b ↑ 2 - 4 * a * c))/(2 * a),
| | | | (-b - sqrt(b ↑ 2 - 4 * a * c))/(2 * a))
| otherwise = error "raíces complejas"
```

Puede mejorarse la legibilidad de esta definición llamando *disc* al discriminante, *raízDisc* a la raíz cuadrada de éste y *denom* a la expresión $2 * a$. En HASKELL puede usarse para este propósito la palabra *where*, con lo que la función anterior quedaría:

```
raíces :: Float → Float → Float → (Float, Float)
raíces a b c
| disc ≥ 0 = ((-b + raízDisc)/denom, (-b - raízDisc)/denom)
| otherwise = error "raíces complejas"
| where
| | disc = b ↑ 2 - 4 * a * c
| | raízDisc = sqrt disc
| | denom = 2 * a
```

Se dice que *disc*, *raízDisc* y *denom* son *definiciones locales* de la función *raíces*, ya que, al igual que los argumentos de la función, sólo pueden utilizarse en el cuerpo de ésta (no son visibles fuera de la función en que se definen). El sangrado es obligatorio en las definiciones locales (todas las definiciones locales introducidas deben comenzar un poco más a la derecha que la función en la que se definen y a la misma altura).

Las definiciones locales son el mecanismo más básico que proporciona HASKELL para conseguir *encapsulamiento*. Para poder construir programas de cierta envergadura es fundamental poder definir entidades que sólo sean visibles desde ciertas partes del

programa. HASKELL proporciona un mecanismo más elaborado para este propósito: el sistema de módulos que estudiaremos en el Capítulo 10.

Con una definición local no sólo ganamos legibilidad sino además eficiencia. Las definiciones constantes locales son calculadas como máximo una vez (si son necesarias para calcular el resultado), y los resultados memorizados y utilizados posteriormente.

También es posible introducir definiciones locales que sean funciones de uno o más argumentos, aunque, en este caso, sólo se gana legibilidad:

```
función :: Integer → Integer
función x = f (3 * x) + f (5 * x)
  where
    f n = n ↑ 2 + 2
```

Las definiciones locales `where` sólo pueden aparecer al final de una definición de función, aunque es posible introducir definiciones locales en cualquier parte de una expresión usando las palabras `let` e `in`. El siguiente ejemplo define localmente la función `f` y luego la evalúa con argumento 100:

```
PRELUDE> let f n = n ↑ 2 + 2 in f 100
10002 :: Integer
```

En ambos casos las definiciones locales introducidas pueden ser recursivas, pueden referirse unas a otras, el orden en que aparezcan no es importante (una definición local puede usar otra definida posteriormente en la misma definición), y se ignoran las definiciones locales innecesarias para calcular el resultado, al ser HASKELL perezoso:

```
fun :: Integer → Integer
fun x = 2 * (x + y)
  where
    y = 3 + x
    fallo = x `div` 0
```

```
MAIN> fun 10
46 :: Integer
```

Como vemos, la definición local `fallo` fue ignorada (de no ser así se hubiese producido un error al dividir por cero) al no ser necesaria para calcular el resultado.

Ejercicio 2.26 Escriba una función:

`descomponer :: Integer → (Integer, Integer, Integer)`

que a partir de una cantidad de segundos, devuelva una terna con las horas, minutos y segundos equivalentes.

2.12. EXPRESIONES LAMBDA

Sería absurdo tener que nombrar cualquier expresión antes de usarla. Estamos acostumbrados a expresiones como `f (10 + 4)` en la que el parámetro es una expresión

2.13 - Sangrado

anónima (sin nombre). HASKELL permite definir funciones anónimas usando *expresiones lambda* (también llamadas λ -expresiones). Una expresión como $\lambda x \rightarrow x + 1$ denota en HASKELL la función que toma un argumento y lo devuelve incrementado¹⁰; x es el parámetro formal y el símbolo \rightarrow separa la parte izquierda (parámetros formales) de la parte derecha (cuerpo) de la función. El nombre del argumento no es importante, por lo que otro modo de escribir la función anterior es $\lambda n \rightarrow n + 1$.

```
PRELUDE> :t λx → isUpper x
λx → isUpper x :: Char → Bool
```

```
PRELUDE> (λx → isUpper x) 'A'
True :: Bool
```

La primera línea del diálogo muestra que la expresión lambda introducida es una función de caracteres en booleanos¹¹; la segunda, que al aplicar un parámetro a una expresión lambda se obtiene como resultado la evaluación de la función con dicho parámetro.

También podemos definir funciones de más de un argumento con la notación lambda:

```
PRELUDE> :t λx y → isUpper x && isUpper y
λx y → isUpper x && isUpper y :: Char → Char → Bool
PRELUDE> (λx y → isUpper x && isUpper y) 'A' 'z'
False :: Bool
```

Las expresiones lambda son útiles al manejar funciones de orden superior (las que toman como argumento otras funciones). Estudiaremos esta posibilidad en el Capítulo 3.

2.13. SANGRADO

Un fichero contiene una serie de definiciones. ¿Cómo sabe el compilador dónde acaba una y comienza otra? La mayoría de los lenguajes de programación usan *delimitadores* para este propósito. En HASKELL la disposición del texto del programa (el *sangrado*) delimita las distintas definiciones mediante la regla del fuera de juego (también llamada *layout o offside rule*):

Una definición acaba con el primer trozo de código con un margen izquierdo menor o igual que el del comienzo de la definición actual.

El primer carácter de una definición abre una caja que alberga dicha definición (ver Figura 2.2). La caja permanece abierta hasta que se encuentra algo a la misma altura o a la izquierda del comienzo de la definición actual. En este punto, la caja se cierra (ver Figura 2.3). La regla *layout* se aplica tras las palabras `let`, `where`, `do` y `of`, además de a las definiciones globales y guardas. Como consecuencia de esta regla:

- Todas las definiciones globales en un fichero deben tener el mismo sangrado (se recomienda que comiencen en la primera columna).
- Todas las definiciones locales introducidas por `where` o `let` deben tener el mismo sangrado.

¹⁰La letra λ no está disponible en el teclado, por lo que se usa realmente el carácter \.

¹¹El comando `:t e` hace que el intérprete muestre el tipo de la expresión `e`.

```
función1 x = x * x
    + 2
    * 3
```

Figura 2.2: Sangrando una declaración.

```
función1 x = x * x
    + 2
    * 3
```

```
función2 x = ...
```

Figura 2.3: Sangrando dos declaraciones.

Un sangrado “razonable” no da problemas. Como ejemplo de sangrado incorrecto,

```
f1 :: Integer → Integer
f1 x = z + y
where
  z = 3
  y = 4
```

En este caso se produce un error de compilación ya que se interpreta que la definición local de *y* es parte de la definición de *z*, es decir, se interpreta:

```
f1 :: Integer → Integer
f1 x = z + y
where
  z = 3 y = 4
```

Una sintaxis alternativa a la regla *layout* utiliza llaves y el carácter separador punto y coma en cualquiera de los puntos en los que se aplica dicha regla. Por ejemplo:

```
g :: Integer → Integer → Integer
g x y = doble x + triple y where {doble n = 2 * n; triple n = 3 * n}
h :: Integer → Integer → Integer
h x y = let {doble n = 2 * n; triple n = 3 * n} in doble x + triple y
```

De hecho, una fase previa a la compilación de un programa añade llaves y puntos y coma si no aparecen. Por ello, si la regla *layout* no se usa adecuadamente, el compilador puede mostrar errores del tipo “carácter '{' esperado” o “carácter ';' esperado”. Estos mensajes de error lo que indican realmente es que hay un problema de sangrado.

2.14. ÁMBITOS Y MÓDULOS

Las reglas de *ámbito* establecen desde qué puntos del programa son visibles los distintos identificadores. Así, el *ámbito* de un argumento en una definición de función es todo el cuerpo de ésta. Por ejemplo, si:

```
f :: Integer → Integer
f x = x * h 7
where
  h z = x + 5 + z ↑ 2
(+++) :: Integer → Integer → Integer
a +++ b = 2 * a + f b
```

el argumento *x* es visible en todo el cuerpo de la función *f*, incluso en *h* ya que ésta forma parte del cuerpo de *f*. Sin embargo, *x* no es visible desde el cuerpo del operador *(+++)*. Se dice que *x* es local a *f*. Lo mismo ocurre con la definición local *h*, que es visible tan sólo desde el cuerpo de *f*.

A las definiciones de funciones que aparecen en el nivel más externo del programa se les llama *globales*. La función *f* y el operador *(+++)* son globales. Esto significa que son visibles desde cualquier otra función. Por ejemplo, el operador *(+++)* puede usar *f* por ser global esta función. Como vemos, en el cuerpo de una definición pueden aparecer referencias a sus argumentos, a sus definiciones locales y a otras funciones globales.

HASKELL proporciona un conjunto de definiciones globales que pueden ser usadas por el programador sin necesidad de definirlas. Por ejemplo, los operadores aritméticos *(+)*, *(*)* y *(↑)* son utilizados en el cuerpo de *f* y *(+++)*. Estas definiciones aparecen agrupadas en *módulos de biblioteca* (conjuntos de definiciones relacionadas).

Ejemplo 2.27 Existe una biblioteca en la que se define el tipo *Rational* que representa números racionales. El nombre de esta biblioteca es *Ratio*. Para poder utilizar los elementos definidos dentro de una biblioteca es necesario *importarla*. La importación se consigue escribiendo la palabra *import* al inicio del programa. Por ejemplo, el siguiente programa utiliza la biblioteca de números racionales para definir una función que suma el cubo de sus dos argumentos racionales¹²:

¹²HUGS importa automáticamente el módulo *Ratio* pero esto no es una característica estándar.

```
import Ratio
sumaCubos :: Rational → Rational → Rational
sumaCubos ra rb = ra ^ 3 + rb ^ 3
```

El siguiente diálogo muestra el uso de esta función anterior, donde $a \% b$ denota el número racional $\frac{a}{b}$ en la biblioteca *Ratio*:

```
MAIN> sumaCubos (1 % 3) (2 % 7)
559 % 9261 :: Rational
```

Existe un módulo de biblioteca especial denominado PRELUDE. Este módulo lo importa automáticamente cualquier programa, por lo que sus elementos pueden usarse directamente. A los elementos definidos en este módulo se les llama *predefinidos*. En este módulo se definen, por ejemplo, los distintos tipos de datos estudiados en las Secciones 2.2 y 2.3 y sus operaciones asociadas.

Además de usar bibliotecas ya definidas, el programador puede ampliar el lenguaje definiendo sus propias bibliotecas. Estudiaremos en profundidad esta posibilidad en el Capítulo 10.

2.15. EJERCICIOS

2.28 Sea una lista de funciones de enteros en enteros:

$$[f_1, f_2, \dots, f_n] :: [Integer \rightarrow Integer]$$

Defina un operador ($|>$) de forma que se tenga:

$$[f_1, f_2, \dots, f_n] |> x \implies [f_1 x, f_2 x, \dots, f_n x]$$

2.29 Escriba una función que determine si un año es bisiesto. Un año es bisiesto si es múltiplo de 4 (por ejemplo 1984). Una excepción a la regla anterior es que los años múltiples de 100 sólo son bisiestos cuando a su vez son múltiplos de 400 (por ejemplo 1800 no es bisiesto, mientras que 2000 sí):

```
MAIN> esBisiesto 1984
True :: Bool
```

2.30 Escriba una función que calcule el número de días de un mes, dados los valores numéricos del mes y año. Nota: considere los años bisiestos para febrero.

2.31 Escriba una función que añada un dígito a la derecha de un número entero:

```
MAIN> 3 `aLaDerechaDe` 146
1463 :: Integer
```

2.15 - Ejercicios

2.32 Escriba una función recursiva que devuelva el resto de la división de dos enteros usando sustracciones.

2.33 Escriba una función recursiva que devuelva el cociente que se obtiene al dividir dos números enteros usando sumas y restas.

2.34 Escriba una función recursiva que devuelva el sumatorio desde un valor entero hasta otro:

$$\text{sumDesdeHasta } a b \implies a + (a+1) + (a+2) + \dots + (b-1) + b$$

2.35 Escriba una función recursiva que devuelva el producto desde un valor entero hasta otro:

$$\text{prodDesdeHasta } a b \implies a * (a+1) * (a+2) * \dots * (b-1) * b$$

2.36 Escriba una función variaciones que calcule el número de variaciones de m elementos tomados de n en n . Use para ello la siguiente relación:

$$\text{variaciones } m n = \frac{m!}{(m-n)!}$$

Escriba una versión alternativa que use esta otra:

$$\text{variaciones } m n = (m-n+1) * (m-n+2) * \dots * (m-1) * m$$

2.37 Escriba una función que calcule números combinatorios con la siguiente relación:

$$\binom{m}{n} = \frac{m!}{(m-n)! \cdot n!}$$

Escriba otra versión que use estas relaciones:

$$\binom{m}{0} = 1, \quad \binom{m}{m} = 1, \quad \binom{m}{n} = \binom{m-1}{n-1} + \binom{m-1}{n}$$

¿Puede garantizar que la última definición acaba?

2.38 Escriba una función que devuelva el i -ésimo número de la sucesión de Fibonacci. Esta sucesión tiene como primer término 0, como segunda 1, y cualquier otro término se obtiene sumando los dos que le preceden: 0, 1, 1, 2, 3, 5, 8, 13, ...

```
MAIN> fibonacci 0
0 :: Integer
```

```
MAIN> fibonacci 6
8 :: Integer
```

2.39 Escriba una función que determine el mayor de tres números enteros. Escriba otra para cuatro números.

2.40 Escriba una función que tome tres números enteros y devuelva una terna con los números ordenados en orden creciente:

```
MAIN> ordena3 10 4 7
(4, 7, 10) :: (Integer, Integer, Integer)
```

2.41 Escriba una función que determine si un número positivo de exactamente cuatro cifras es capicúa o no:

```
MAIN> esCapicua 1221
True :: Bool
```

```
MAIN> esCapicua 12
ERROR : número de cifras incorrecto
```

2.42 Escriba una función que calcule la suma de las cifras de un número natural:

```
MAIN> sumaCifras 123
6 :: Integer
```

2.43 Escriba una función que calcule el número de cifras de un número natural (sin ceros a la izquierda):

```
MAIN> numeroCifras 123
3 :: Integer
```

2.44 Escriba una función trocear que tome un número de n dígitos y que, usando sólo sumas y restas, devuelva un par donde el primer elemento corresponde a los $n - 1$ primeros dígitos del número y el segundo elemento sea el dígito n -ésimo. Por ejemplo:

```
PRELUDE> trocear 1234
(123, 4) :: (Integer, Integer)
```

2.45 Escriba una función concatenar que concatene los dígitos de dos números no nulos. Por ejemplo:

```
PRELUDE> concatenar 123 45
12345 :: Integer
```

```
PRELUDE> concatenar 123 0
123 :: Integer
```

Ejemplo 3.1. Una función:

3 FUNCIONES DE ORDEN SUPERIOR Y POLIMORFISMO

3.1. PARCIALIZACIÓN

En HASKELL las funciones de más de un argumento se pueden interpretar como funciones que toman un único argumento y devuelven como resultado otra función con un argumento menos¹. Este mecanismo que permite representar funciones de varios argumentos mediante funciones de un único argumento se denomina *parcialización* (*currying* en los textos anglosajones; el término tiene su origen en el apellido del lógico norteamericano Haskell B. Curry, que popularizó esta notación en los años 30).

Para explicar esta idea, consideremos la siguiente función:

```
inc :: Integer → Integer
inc x = x + 1
```

HASKELL, como la mayoría de los lenguajes funcionales, está basado en el λ-cálculo (Capítulo 17), en el cual el único modo de escribir funciones es utilizando λ-expresiones. En HASKELL, la ecuación $\text{inc } x = x + 1$ es una alternativa a esta otra:

```
inc = λ x → x + 1
```

Las dos definiciones son equivalentes, y la segunda nos ayudará a comprender la notación *parcializada*. A partir de esta definición, la expresión $\text{inc } 10$ se reduce del siguiente modo:

```
inc 10
⇒ { definición de inc }
(λ x → x + 1) 10
⇒ { sustituyendo x por 10 en el cuerpo de la λ-expresión }
(10 + 1)
⇒ { operador (+) }
11
```

Consideremos ahora la siguiente función de dos argumentos:

¹Es importante resaltar que una función puede devolver como resultado otra función. Esto es una novedad para aquellos lectores que hayan utilizado lenguajes imperativos tradicionales como ADA o MODULA-2, pero sin embargo es una característica natural de los lenguajes funcionales modernos.

sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados $x\ y = x * x + y * y$

De nuevo, la definición es una simplificación de esta otra:

sumaCuadrados :: Integer → Integer → Integer
sumaCuadrados = $\lambda x\ y \rightarrow x * x + y * y$

La λ -expresión $\lambda x\ y \rightarrow x * x + y * y$ es, a su vez, un modo más cómodo de escribir $\lambda x \rightarrow (\lambda y \rightarrow x * x + y * y)$. Tal equivalencia se generaliza en la forma:

Regla de λ -abstracciones

$$\lambda x_1 x_2 \dots x_n \rightarrow e \Leftrightarrow \lambda x_1 \rightarrow (\lambda x_2 \rightarrow (\dots \rightarrow (\lambda x_n \rightarrow e)))$$

Además, hemos de recordar que el constructor de tipo funcional (\rightarrow) es asociativo a la derecha, por lo que el tipo $\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$ denota realmente el tipo $\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$. En general:

Asociatividad a la derecha de (\rightarrow)

$$t_1 \rightarrow t_2 \rightarrow \dots t_n \Leftrightarrow (t_1 \rightarrow (t_2 \rightarrow (\dots \rightarrow t_n)))$$

Por todo esto, la definición original de la función *sumaCuadrados* es interpretada como:

sumaCuadrados :: Integer → (Integer → Integer)
sumaCuadrados = $\lambda x \rightarrow (\lambda y \rightarrow x * x + y * y)$

es decir, *sumaCuadrados* es una función que toma un único argumento de tipo *Integer* (el correspondiente a *x*) y devuelve como resultado la función $(\lambda y \rightarrow x * x + y * y)$ de tipo *Integer* → *Integer*; ésta toma un argumento (el correspondiente a *y*) computando la expresión $x * x + y * y$ de tipo *Integer*. Para que las expresiones como *sumaCuadrados* 2 3 tengan sentido, lo único que hace falta es que la aplicación de funciones se asocie a la izquierda, como de hecho ocurre en HASKELL. La expresión *sumaCuadrados* 2 3 denota realmente la expresión *(sumaCuadrados 2) 3*. En general:

Asociatividad izquierda de la aplicación de funciones

$$f a_1 a_2 \dots a_n \Leftrightarrow ((f\ a_1)\ a_2) \dots a_n$$

La reducción de la expresión *sumaCuadrados* 2 3 se produce del siguiente modo:

sumaCuadrados 2 3
 \Leftrightarrow [la aplicación es asociativa a la izquierda]
(sumaCuadrados 2) 3

3.1 - Parcialización

\Rightarrow { definición de *sumaCuadrados* }
 $((\lambda x \rightarrow (\lambda y \rightarrow x * x + y * y)) 2) 3$
 \Rightarrow { sustituyendo *x* por 2 en el cuerpo de la λ -expresión }
 $(\lambda y \rightarrow 2 * 2 + y * y) 3$
 \Rightarrow { sustituyendo *y* por 3 en el cuerpo de la λ -expresión }
 $2 * 2 + 3 * 3$
 \Rightarrow { operador (*) }
 $4 + 3 * 3$
 \Rightarrow { operador (*) }
 $4 + 9$
 \Rightarrow { operador (+) }
13

3.1.1. APLICACIÓN PARCIAL

Dado que las funciones de más de un argumento se representan como funciones con un único argumento que devuelven como resultado funciones, es posible aplicar a una función menos argumentos de los que realmente parece tener, y obtener de este modo una nueva función. A esta posibilidad se la denomina *aplicación parcial* o *parcialización* (no confundir con el concepto de función *parcialmente definida*).

Ejemplo 3.1 Dada la función:

multiploDe :: Integer → Integer → Bool
multiploDe $p\ n = n \bmod p == 0$

que comprueba si su segundo argumento es múltiplo del primero, podemos obtener una función que comprueba si un número es par, aplicando parcialmente el primer argumento:

esPar :: Integer → Bool
esPar = *multiploDe* 2

MAIN> *multiploDe* 3 15
True :: Bool

MAIN> *esPar* 18
True :: Bool

En efecto, tenemos:

\Rightarrow { definición de *esPar* }
multiploDe 2
 \Rightarrow { definición de *multiploDe* }
 $(\lambda p \rightarrow (\lambda n \rightarrow n \bmod p == 0)) 2$
 \Rightarrow { sustituyendo *p* por 2 en el cuerpo de la λ -expresión }
 $\lambda n \rightarrow n \bmod 2 == 0$

es decir, `esPar` es una función que toma un entero (n) y devuelve un booleano ($n \bmod 2 == 0$).

Ejemplo

Es necesario que el argumento aplicado a una función coincida en tipo con el primer argumento de ésta. Se obtiene como resultado una función con un argumento menos:

Tipificado de la aplicación de funciones

Si $f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n$ y $x_1 :: t_1$, entonces $f \ x_1 :: t_2 \rightarrow \dots \rightarrow t_n$

Ejemplo 3.2 A la siguiente función de tres argumentos:

```
fun      :: Int → Int → Int → Int
fun x y z = x * (2 * y + z)
```

es posible aplicarle uno, dos o tres argumentos. En cada caso obtenemos una función con un argumento menos (ver Figura 3.1).

Ejemplo

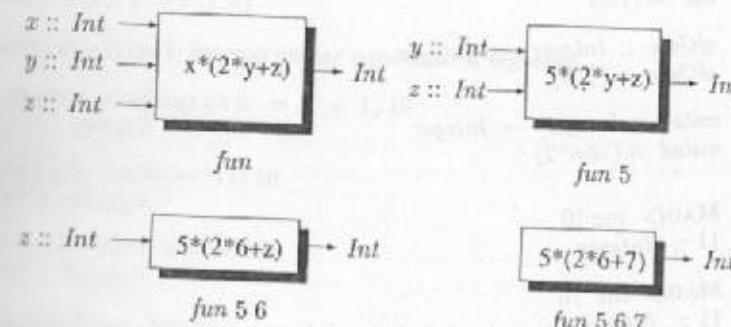


Figura 3.1: Aplicaciones parciales.

Como podemos observar, la parcialización permite un uso más versátil de las funciones, ya que a una función de n argumentos es posible aplicarle desde cero a n argumentos. La aplicación parcial también resulta muy útil al trabajar con funciones de orden superior, como veremos posteriormente. Obsérvese que sólo es posible ir aplicando parcialmente argumentos de izquierda a derecha. Si por ejemplo quisieramos aplicar parcialmente tan sólo el segundo argumento deberíamos usar una λ -expresión.

Ejemplo 3.3 Sea la función:

```
multiploDe'    :: (Integer, Integer) → Bool
multiploDe' (p, n) = n `mod` p == 0
```

3.1 - Parcialización

MAIN> `multiploDe' (3, 15)`

`True :: Bool`

En este caso, las aplicaciones parciales no son posibles: siempre hay que aplicar dos enteros (en forma de tupla) para usar la función `multiploDe'`.

Ejemplo

Por lo tanto hay dos modos de representar funciones de varios argumentos en HASKELL:

$f :: t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_r$
 $f \ x_1 \ x_2 \ \dots \ x_n = \dots$ -- Forma parcializada

$f' :: (t_1, t_2, \dots, t_n) \rightarrow t_r$
 $f' (x_1, x_2, \dots, x_n) = \dots$ -- Forma no parcializada

pero sólo la primera puede ser parcialmente aplicada:

Si $v_1 :: t_1$ entonces $f \ v_1 :: t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_r$
Si $v_1 :: t_1, v_2 :: t_2$ entonces $f \ v_1 \ v_2 :: t_3 \rightarrow \dots \rightarrow t_n \rightarrow t_r$
...
Si $v_1 :: t_1, v_2 :: t_2, \dots, v_n :: t_n$ entonces $f \ v_1 \ v_2 \ \dots \ v_n :: t_r$

Un concepto relacionado con la aplicación parcial es la η -reducción de expresiones:

$\lambda x \rightarrow \text{expr } x$	\equiv	expr	si x no aparece libre en expr
--	----------	---------------	--

Como consecuencia de esta regla, podemos cancelar el último argumento en la parte izquierda de una definición de función si éste es el último aplicado en la parte derecha, siempre que dicho argumento no quede libre (no siga apareciendo) tras la cancelación. La condición adicional es importante.

Ejemplo 3.4 La declaración:

$f \ x = g \ 5 \ x$

puede ser escrita como:

$f = g \ 5$

cancelando la x en la definición de f . Sin embargo, la declaración:

$f \ x = g \ (x + 1) \ x$

no, ya que la variable x en la expresión $(x + 1)$ quedaría libre.

Ejemplo

Las funciones de PRELUDE, *curry* y *uncurry* permiten convertir una función de dos argumentos no parcializada a la forma parcializada y viceversa:

$$\begin{aligned} \text{curry} &:: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \\ \text{curry } f \ x \ y &= f(x, y) \\ \text{uncurry} &:: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c) \\ \text{uncurry } f \ (x, y) &= f \ x \ y \end{aligned}$$

Por ejemplo:

```
esPar' :: Integer → Bool
esPar' = (curry múltiploDe') 2
        -- o también curry múltiploDe' 2
        -- ya que la aplicación asocia a
        -- la izquierda
```

```
MAIN> esPar' 18
True :: Bool
```

Las funciones *curry* y *uncurry* son ambas de orden superior (ya que toman como primer argumento una función) y *polimórficas* (ya que pueden actuar con distintos tipos).

3.1.2. SECCIONES

En HASKELL también es posible aplicar a un operador menos de dos argumentos (todos los operadores excepto el cambio de signo son binarios). A una expresión así se la denomina *sección*. Recordemos que es posible aplicar un operador de modo infijo:

```
PRELUDE> 5 ↑ 2
25 :: Integer
```

o prefijo, y en este caso debe ir entre paréntesis:

```
PRELUDE> (↑) 5 2
25 :: Integer
```

Las secciones permiten invocar un operador binario bien con tan sólo su primer argumento, bien con tan sólo el segundo, o sin ninguno de los dos. Para el operador potencia, tres posibles secciones son:

- (5 ↑) La función que toma un argumento y eleva el valor 5 a éste.
- (↑ 5) La función que toma un argumento y eleva éste a 5.
- (↑) La función que toma dos argumentos y eleva el primero al segundo.

Los paréntesis son obligatorios en las secciones de operadores. En general, si (\otimes) es un operador, tenemos las siguientes equivalencias:

3.1 - Parcialización

Secciones de operadores

$$\begin{aligned} (x \otimes) &\iff (\lambda y \rightarrow x \otimes y) \\ (\otimes y) &\iff (\lambda x \rightarrow x \otimes y) \\ (\otimes) &\iff (\lambda x \ y \rightarrow x \otimes y) \end{aligned}$$

Una excepción a la reglas de las secciones es el operador $(-)$. Una expresión con la forma $(- e)$ no es una sección sino que representa la negación de e . El operador $-$ es la única función simbólica unaria en HASKELL. El programador no puede definir nuevos operadores unarios, por lo que es la única excepción posible a la regla.

Algunos ejemplos de secciones son los siguientes:

```
inc :: Integer → Integer
inc = (+1)
```

```
inc' :: Integer → Integer
inc' = (1+)
```

```
alCubo :: Integer → Integer
alCubo = (↑ 3)
```

```
mitad :: Integer → Integer
mitad = (^div 2)
```

```
MAIN> inc 10
11 :: Integer
```

```
MAIN> inc' 10
11 :: Integer
```

```
MAIN> alCubo 5
125 :: Integer
```

```
MAIN> (↑ 3) 5
125 :: Integer
```

Por ejemplo,

```
alCubo
⇒ { definición de alCubo }
      (↑ 3)
⇒ { reglas de las secciones }
      (λ x → x ↑ 3)
```

Por lo que *alCubo* es la función que toma un argumento y lo eleva al cubo. Como muestra la función *mitad*, el operador literal *^div* puede ser *seccionado*. Las secciones resultan muy útiles al trabajar con funciones de orden superior.

3.1.3. FUNCIONES DE ORDEN SUPERIOR

Una de las principales características que diferencian a los lenguajes de programación funcionales es que consideran a las funciones datos de *primera clase*. Esto significa que una función puede aparecer en cualquier lugar donde pueda aparecer un dato de otro tipo. Por ejemplo, es posible construir una lista cuyos elementos sean funciones, siempre y cuando todas tengan el mismo tipo:

```
lista :: [Integer → Integer]
lista = [(\lambda x → x + 1), (+1), dec, (\↑ 2)]
where
  dec x = x - 1
```

Una consecuencia de esta propiedad es que una función, como cualquier otro tipo de dato, puede aparecer como argumento de otra función o como resultado de ésta. A las funciones que manejan funciones se las denomina *funciones de orden superior*. La siguiente función es un ejemplo de función de orden superior, ya que toma como primer argumento una función de enteros en enteros:

```
dosVeces :: (Integer → Integer) → Integer → Integer
dosVeces f x = f (f x)
```

que aplica dos veces la función primer argumento al segundo:

```
MAIN> dosVeces (\lambda x → x + 1) 10
12 :: Integer
MAIN> dosVeces (*2) 10
40 :: Integer
MAIN> dosVeces inc 10
12 :: Integer
```

Los paréntesis son obligatorios en el tipo de la función, ya que el tipo:

$$\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$$

es equivalente a:

$$\text{Integer} \rightarrow (\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer}))$$

Es decir, el de una función de tres argumentos enteros, pero *dosVeces* tiene sólo dos argumentos, y el primero no es un entero sino una función de enteros en enteros. El tipo de *dosVeces* refleja que si sólo se pasa el argumento *f*, *dosVeces f* devuelve una función:

```
MAIN> :t dosVeces inc
dosVeces inc :: Integer → Integer
```

Quizás esto se vea mejor si cambiamos la definición por su equivalente.

$$\text{dosVeces } f = \lambda x \rightarrow f (f x)$$

Ejemplo 3.5 Una función de orden superior que devuelve como resultado otra función es:

```
derivada :: (Float → Float) → Float → Float
derivada f x = (f (x + h) - f x) / h
where
  h = 0.0001
```

que devuelve una aproximación numérica de la derivada de una función real:

$$f'x = \lim_{h \rightarrow 0} \frac{f(x + h) - f x}{h}$$

Recuérdese que la ecuación anterior puede sustituirse por su equivalente:

```
derivada f = \lambda x \rightarrow (f (x + h) - f x) / h
where
  h = 0.0001
```

A partir de cualquiera de las dos definiciones obtenemos el siguiente diálogo:

```
MAIN> :t derivada sqrt
derivada sqrt :: Float → Float
MAIN> derivada sqrt 1.0
0.499487 :: Float
MAIN> derivada sin 0.0
1.0 :: Float
```

Ejemplo 3.6 Otra función de orden superior es la siguiente:

```
logEnBase :: Float → (Float → Float)
logEnBase b = \lambda x → log x / log b -- log computa el logaritmo neperiano
```

que toma un argumento real correspondiente a una base y devuelve la función logaritmo en dicha base, utilizando la fórmula:

$$\log_b x = \frac{\ln(x)}{\ln(b)}$$

```
MAIN> :t logEnBase 2
logEnBase 2 :: Float → Float
```

```
MAIN> (logEnBase 2) 16
4.0 :: Float
```

```
MAIN> logEnBase 2 16
4.0 :: Float
```

Obsérvese que las expresiones 2 y 16 están actuando en este caso con tipo *Float*. El tipo de una constante numérica se infiere del contexto en que se usa. También podríamos haber usado directamente las expresiones 2.0 y 16.0. Esta característica será estudiada con más detalle en el Capítulo 5.

Ejemplo

3.1.4. UNA FUNCIÓN DE ORDEN SUPERIOR SOBRE NATURALES

Muchas de las funciones recursivas que se definen sobre naturales siguen el siguiente esquema inductivo:

- *Caso Base*: Se define el resultado de la función cuando el argumento es cero.
- *Paso Inductivo*: Se define el resultado de la función cuando el argumento es $n + 1$ en función del resultado devuelto por la función cuando el argumento es n .

Este tipo de definiciones se llaman inductivas. La función *factorial* es un ejemplo²:

```
factorial      :: Integer → Integer
factorial 0     = 1
factorial m@(n + 1) = (*) m (factorial n)
```

Otro ejemplo es la función *sumatorio* que devuelve la suma de los primeros naturales:

```
sumatorio      :: Integer → Integer
sumatorio 0     = 0
sumatorio m@(n + 1) = (+) m (sumatorio n)
```

Podemos observar que ambas funciones siguen la siguiente plantilla:

```
fun      :: Integer → Integer
fun 0     = e
fun m@(n + 1) = op m (fun n)
```

Por ejemplo, la definición de *factorial* se obtiene sustituyendo *e* por 1 y *op* por $(*)$. Es posible definir una función de orden superior que tome como argumentos la operación a aplicar en el caso recursivo y el valor a devolver en el caso base:

```
iter      :: (Integer → Integer → Integer) →
            Integer →
            Integer →
            Integer
iter op e 0     = e
iter op e m@(n + 1) = op m (iter op e n)
```

²El operador $(+)$ se ha escrito de modo prefijo por conveniencia.

3.1 - Parcialización

que también puede ser definida utilizando definiciones locales:

```
iter      :: (Integer → Integer → Integer) → Integer →
            (Integer → Integer)
iter op e = fun
  where
    fun 0          = e
    fun m@(n + 1) = op m (fun n)
```

La definición anterior es lo que se denomina un *combinador* (una función de orden superior que captura un esquema de cómputo). El combinador *iter* captura las definiciones inductivas sobre los naturales. Se dice también que es un *iterador* para los valores naturales. La gran ventaja de definir combinadores es que, a partir de ellos, es fácil definir cualquier función que siga el esquema capturado. Para definir las funciones *factorial* y *sumatorio* basta con proporcionar los dos primeros argumentos adecuadamente:

```
factorial' :: Integer → Integer
factorial' = iter {(*)} 1
sumatorio' :: Integer → Integer
sumatorio' = iter {(+) 0}
```

(Los paréntesis son obligatorios cuando se pasa un operador simbólico como argumento.) Como vemos, las nuevas definiciones resultan más compactas y ocultan la recursividad aunque las definiciones siguen siendo recursivas. Por ejemplo, *factorial'* 2 se calcula del siguiente modo:

```
factorial' 2
⇒ {definición de factorial'}
iter {(*)} 1 2
⇒ {segunda ecuación de iter}
(*) 2 (iter {(*)} 1 1)
⇒ {segunda ecuación de iter}
(*) 2 ((*) 1 (iter {(*)} 1 0))
⇒ {primera ecuación de iter}
(*) 2 ((*) 1 1)
⇒ {(*)}
(*) 2 1
⇒ {(*)}
2
```

La mayoría de los programadores primerizos abusan de las definiciones recursivas. Se considera una buena práctica de programación definir combinadores que capturen patrones comunes de cómputo y utilizar éstos en lugar de las correspondientes definiciones recursivas. Ésta es una de las mayores ventajas que aportan las funciones de orden superior: permiten al programador ampliar el lenguaje de programación. En el PRELUDE de HASKELL se definen un número considerable de combinadores a partir de los cuales se pueden definir la mayoría de las funciones. Los estudiaremos a lo largo del libro.

Ejercicio 3.7 Escriba la siguiente función potencia a través del combinador `iter`:

```
potencia      :: Integer → Integer → Integer
potencia b 0  = 1
potencia b m@(n + 1) = b * potencia b n
```

3.2. POLIMORFISMO

Hasta ahora casi todas las funciones definidas han tenido como dominio y codominio tipos concretos (por ejemplo, `Int`, `Integer`, ...). Sin embargo, algunas funciones tienen sentido para más de un tipo. Este tipo de funciones se denominan *funciones polimórficas*.

El ejemplo más simple de función polimórfica es la función *identidad* `id`:

```
id x = x
```

de modo que:

```
MAIN> id 'd'
'd' :: Char
```

```
MAIN> id True
True :: Bool
```

```
MAIN> id toUpper 'a'
'A' :: Char
```

En el primer diálogo la función actúa con el tipo `Char → Char`, en el segundo con tipo `Bool → Bool`, y en el último con tipo: `(Char → Char) → (Char → Char)`. La función tiene sentido para argumentos de cualquier tipo, y el tipo del valor devuelto debe coincidir con el tipo del argumento. En HASKELL el tipo de la función identidad es:

```
id :: a → a
```

En esta declaración, *a* es una *variable de tipo* (es igualmente válido escribir *b* → *b*, por ejemplo). Las variables de tipo deben empezar por una letra minúscula. Una variable de tipo denota cualquier tipo. La declaración anterior indica que la función `id` puede actuar con cualquier tipo que se obtenga al sustituir la variable de tipo *a* por un tipo concreto. El tipo de la declaración debe leerse como `id :: ∀ a . a → a`. Cualquiera de los tipos anteriores (`Char → Char`, `Bool → Bool`, ...) es un tipo válido de `id`. Sin embargo, el uso de la función `id` en la expresión `(id 'd') + 3` es incorrecto, ya que para que la suma sea posible es necesario que el tipo de `id 'd'` sea `Integer`, y una variable de tipo sólo puede ser reemplazada con un único tipo en el mismo uso de la función.

Por otro lado, si hubiésemos declarado la función identidad como:

```
id :: a → b
id x = x
```

3.2 - Polimorfismo

habríamos obtenido un error de tipo, ya que el tipo declarado es *demasiado general*. El intérprete deduce del cuerpo de la función que el tipo del resultado coincide necesariamente con el tipo del argumento, y la declaración de tipo anterior indica que pueden ser distintos, lo cual es falso, y por ello la declaración no puede ser aceptada. En general, gracias al sistema de tipos, se puede inferir el tipo más general de una función a partir de sus ecuaciones. Para ello basta con que omitamos la declaración de tipos al definir la función. Para visualizar el tipo más general de una función definida de este modo, podemos usar el comando `:t` de HUGS seguido del nombre de la función. Por ejemplo, si incluimos esta función en un fichero:

```
unaVez f x = f x
```

y la cargamos en el intérprete, podemos obtener el tipo de la función tecleando:

```
MAIN> :t unaVez
unaVez :: (a → b) → a → b
```

A partir de la Figura 3.2 podemos observar que para esta función, el tipo del dominio de *f* ha de coincidir con el de *x*, pero no es necesario que el tipo del codominio coincida, y por ello se han utilizado dos variables *a* y *b* de tipo distintas. Un ejemplo de diálogo es:

```
MAIN> unaVez (+1) 7
8 :: Integer
```

```
MAIN> unaVez (== 0) 9
False :: Bool
```

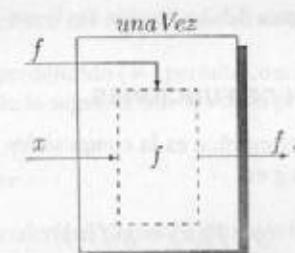


Figura 3.2: La función `unaVez`.

En el primer diálogo `unaVez` actúa con tipo `(Integer → Integer) → Integer → Integer`. En el segundo, el tipo es `(Integer → Bool) → Integer → Bool`. Un ejemplo de uso incorrecto de la función es:

```

MAIN> unaVez inc 'p'
ERROR      : Type error in application
*** Expression : unaVez inc 'p'
*** Term       : inc
*** Type       : Integer → Integer
*** Does not match : Char → Integer

```

ya que la variable (de tipo) *a* tendría que ser reemplazada simultáneamente por los tipos *Integer* y *Char*, y esto no es posible.

Ejercicio 3.8 ¿Cuál es el tipo más general de la función dosVeces (ver la Figura 3.3)?

$$\text{dosVeces } f \ x = f(f \ x)$$

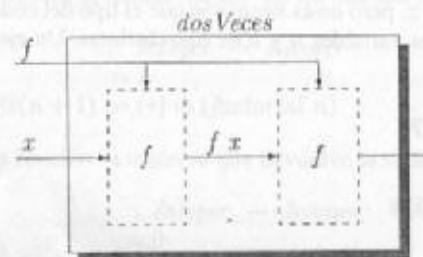


Figura 3.3: La función dosVeces.

3.2.1. LA COMPOSICIÓN DE FUNCIONES

Otro ejemplo de función polimórfica es la *composición de funciones*. La definición matemática de *f* compuesta con *g* es:

$$(g \circ f)(x) = g(f(x))$$

La composición es un operador que actúa sobre dos funciones y devuelve como resultado una nueva función (ver Figura 3.4). En HASKELL, la composición de funciones se denota mediante un punto, y está definida en el PRELUDE del siguiente modo:

```

infixr 9 .
(.) :: (b → c) → (a → b) → (a → c)
g . f = λ x → g(f x)

```

La Figura 3.4 muestra este operador gráficamente. Por ejemplo, la función:

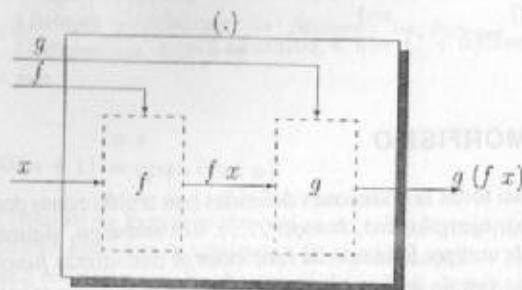


Figura 3.4: La composición de funciones.

```

sumayMult :: Integer → Integer
sumayMult = (*10) . (+1)

```

toma un entero, lo incrementa y después lo multiplica por diez:

```

MAIN> sumayMult 5
60 :: Integer

```

El tipo de la composición de funciones indica que no todo par de funciones puede ser compuesto. Tenemos las siguientes restricciones para *g.f*:

- El tipo del resultado de *f* debe coincidir con el tipo del argumento de *g*.
- El tipo del resultado de la composición (*g.f*) coincide con el del resultado de *g*.
- El tipo del argumento de (*g.f*) es el mismo que el tipo del argumento de *f*.

Si llamamos:

- *a* al tipo del argumento de *f*,
- *b* al tipo del argumento de *g*,
- *c* al tipo del resultado de *g*,

y tenemos en cuenta las restricciones anteriores, obtenemos el tipo de la composición:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

La composición de funciones es un patrón de cómputo muy habitual. Así, si la solución a un problema consta de varias etapas, podemos definir cada una de ellas como funciones independientes y componer todas para solucionar el problema.

Ejemplo 3.9 Usando la composición podemos definir la función *esImpar* a partir de la función *esPar* para comprobar la paridad de un número:

```
esPar, esImpar :: Integer → Bool
esPar x = (x `mod` 2 == 0)
esImpar = not . esPar
```

```
MAIN> esImpar 3
True :: Bool
```

donde *not* :: *Bool* → *Bool* es la negación lógica, y (==) es la comparación de igualdad para dos elementos.

Por la declaración infixr, el operador de composición de funciones es asociativo a la derecha, de donde la ecuación de la definición:

```
f :: Integer → Integer
f = (+1) . (*3) . (↑ 2)
```

es equivalente a:

```
f = (+1) . ((*3) . (↑ 2))
```

Es decir, primero se eleva el argumento al cuadrado, después se multiplica por tres y por último se le suma uno³.

El operador (\$) puede ser usado para evaluar una composición de funciones en un punto sin escribir ésta entre paréntesis, ya que tiene menor prioridad que la composición:

```
PRELUDE> ((+1) . (*3) . (↑ 2)) 10
301 :: Integer
```

```
PRELUDE> (+1) . (*3) . (↑ 2) $ 10
301 :: Integer
```

El operador de composición proporciona un gran nivel de abstracción, ya que permite construir funciones a partir de otras y olvidarnos de los identificadores de los argumentos. Por ejemplo, la función *dosVeces* puede ser definida como:

```
dosVeces f = f . f
```

A este estilo de definir funciones se lo denomina *sin argumentos* (*point-free*). Al estilo habitual (con argumentos) se lo denomina en los textos anglosajones *point-wise*. HASKELL permite utilizar cualquiera de los dos. La ventaja de usar el estilo *sin argumentos* es que las definiciones resultantes son más concisas. Un detalle que suele pasar inadvertido es que la η -reducción puede hacer el tipo de una función más general. Por ejemplo, recuérdese que el tipo más general de la declaración de la función *unaVez* es el siguiente:

³Aunque la composición en matemáticas sea asociativa, en HASKELL hay que declarar la asociatividad del operador para resolver la ambigüedad en el cálculo con composiciones.

3.2 - Polimorfismo

unaVez :: (a → b) → a → b
unaVez f x = f x

Dicha función puede ser η -reducida a:

unaVez :: t → t
unaVez f = f

o incluso a:

unaVez :: t → t
unaVez = id

donde *id* es la función identidad definida en PRELUDE. Sin embargo, el tipo obtenido vía η -reducción es más general. No hay problema ya que el tipo original es un caso particular del tipo *t* → *t* en el que se han reemplazado las apariciones de la variable *t* de tipos con el tipo *a* → *b*. En resumen, al parcializar se puede perder información sobre el tipo:

Ecuación	Tipo
<i>fun f x y = f x y</i>	$(a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$
<i>fun f x = f x</i>	$(a \rightarrow b) \rightarrow a \rightarrow b$
<i>fun f = f</i>	$a \rightarrow a$

3.2.2. OTRAS FUNCIONES POLIMÓRFICAS

El PRELUDE define otras funciones polimórficas interesantes. Una de ellas es *flip*:

flip :: (a → b → c) → (b → a → c)
flip f x y = f y x

que permite aplicar una función a sus argumentos en orden inverso.

Ejemplo 3.10 El operador predefinido (++) permite concatenar dos listas del mismo tipo disponiendo los elementos de la segunda tras los de la primera:

```
MAIN> [1, 2, 3] ++ [5, 6]
[1, 2, 3, 5, 6] :: Integer
```

Podemos definir un operador que disponga primero los de la segunda:

(>>) :: [a] → [a] → [a]
xs (>>) ys = ys ++ xs

así que obtendríamos una definición equivalente más breve usando *flip*:

(>>) :: [a] → [a] → [a]
(>>) = flip (++)

ya que:

(><) $xs\ ys = \text{flip} (+) xs\ ys$

MAIN> [1, 2, 3] >< [5, 6]
[5, 6, 1, 2, 3] :: Integer

Ejemplo

Otro uso común de *flip* es para parcializar el primer argumento de una función:

```
fun      :: Integer → Integer → Integer
fun x y = 2 * x + y
f :: Integer → Integer
f = fun 3 -- La función que suma 6
g :: Integer → Integer
g = flip fun 10 -- La función que duplica y suma 10
```

o para escribir secciones:

```
PRELUDE> let mitad = (/2) in mitad 10
5.0 :: Double
PRELUDE> let mitad = flip (/) 2 in mitad 10
5.0 :: Double
```

El operador predefinido (\$) se comporta como una vez:

```
infixr 0 $
($) :: (a → b) → (a → b)
f $ x = f x
```

y puede ser usado para evitar el uso de paréntesis gracias a su baja prioridad:

```
PRELUDE> ((+2) . (+1) . (↑ 2)) 5
52 :: Integer
PRELUDE> (+2) . (+1) . (↑ 2) $ 5
52 :: Integer
```

Obsérvese que el primer argumento de (\$) debe ser una función:

```
MAIN> ($) inc 5
6 :: Integer
MAIN> (inc $) 5
6 :: Integer
MAIN> ($ 5) inc
6 :: Integer
```

de modo que en general se cumplen las siguientes igualdades:

3.2 - Polimorfismo

$$(\$) f = (f \$) = f \\ (\$ x) = \text{flip} (\$) x = \lambda f \rightarrow f x$$

Por último, las funciones *curry* y *uncurry* citadas en la Sección 3.1.1 son también polimórficas.

3.2.3. POLIMORFISMO EN LISTAS

Muchas de las funciones que actúan sobre listas son polimórficas.

Ejemplo 3.11 Sea la siguiente función que calcula la longitud de una lista de caracteres:

```
lengthChar      :: [Char] → Int
lengthChar []   = 0
lengthChar (_ : xs) = 1 + lengthChar xs
```

Es fácil observar que el cuerpo de la función no depende del tipo de los elementos que hay en la lista, por lo que es posible definir una función polimórfica que calcule la longitud de listas con cualquier tipo base:

```
length      :: [a] → Int
length []   = 0
length (_ : xs) = 1 + length xs
```

```
PRELUDE> length [True, False, False]
3 :: Int
```

```
PRELUDE> length [[1, 2, 3], [4, 5]]
2 :: Int
```

```
PRELUDE> length [(+2), (*5)]
2 :: Int
```

Obsérvese que el tipo de la función no es:

length :: a → Int -- error !!!

ya que el primer argumento de la función debe ser una lista y este tipo al ser más general no refleja este hecho. Los constructores para listas tienen tipos polimórficos:

```
(:) :: a → [a] → [a]
[] :: [a]
```

Por ejemplo, en la expresiones:

```
PRELUDE> 1 : [2, 3, 4]
[1, 2, 3, 4] :: [Integer]
```

```
PRELUDE> [True] : [[False, True], [False]]
[[True], [False, True], [False]] :: [[Bool]]
```

el constructor `(:)` actúa con los tipos $\text{Integer} \rightarrow [\text{Integer}] \rightarrow [\text{Integer}]$ y $[\text{Bool}] \rightarrow [[\text{Bool}]] \rightarrow [[\text{Bool}]]$ respectivamente, es decir la variable de tipo a toma los valores Integer y $[\text{Bool}]$. Lo mismo ocurre con el constructor `[]` en las expresiones:

```
PRELUDE> 1 : []      -- [] actúa con tipo [Integer]
[1] :: Integer
```

```
PRELUDE> [True] : []    -- [] actúa con tipo [[Bool]]
[[True]] :: [[Bool]]
```

Sin embargo, la expresión:

```
PRELUDE> 'p' : [1,2,3]
ERROR          : Type error in application
*** expression : 'p' : [1,2,3]
*** term       : 'p'
*** type        : Char
*** does not match : Int
```

produce un error de tipo, ya que la variable a de tipo debería ser reemplazada a dos tipos distintos (Char e Int) en la misma expresión, y esto no es posible.

En realidad, una función con tipo polimórfico tiene muchos tipos. Un tipo polimórfico es una plantilla (o, técnicamente, *un esquema de tipos*) que puede ser utilizada para crear tipos específicos. Para que un uso particular de una función polimórfica sea correcto, las variables de tipo puede ser reemplazadas por tipos concretos (todas las apariciones de la misma variable de tipo deben ser reemplazadas por el mismo tipo). Una expresión en la que intervienen funciones polimórficas es correcta desde el punto de vista de su tipo si se pueden encontrar sustituciones consistentes para las variables de tipo.

La combinación de funciones de orden superior y polimorfismo da lugar a funciones muy útiles para listas.

Ejemplo 3.12 La función `map`, definida en el PRELUDE, aplica una función a todos los elementos de una lista, devolviendo una lista con los resultados:

```
map      :: (a → b) → [a] → [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

```
PRELUDE> map (+ 2) [1,2,3]
[3,4,5] :: [Integer]
```

```
PRELUDE> map toUpper "pepe"
"PEPE" :: String
```

```
PRELUDE> map ord "pepe"
[112,101,112,101] :: [Int]
```

Ejemplo 3.13 Es posible definir un operador que aplique una lista de funciones a un dato, devolviendo una lista de resultados:

3.2. Polimorfismo

$$\begin{aligned} (|>) &:: [a \rightarrow b] \rightarrow a \rightarrow [b] \\ [|] &|>_-=[] \\ (f : fs) &|> x = f x : fs |> x \end{aligned}$$

Este operador puede ser definido sin argumentos (estilo *point-free*) usando `map` y `(\$)`:

$$\begin{aligned} (|>) &:: [a \rightarrow b] \rightarrow a \rightarrow [b] \\ (|>) &= \text{flip} (\text{map} . \text{flip} (\$)) \end{aligned}$$

ya que:

$$\begin{aligned} (|>) fs x &= \text{map} (\$ x) fs \\ &\equiv \{\text{propiedades de } \$\} \\ (|>) fs x &= \text{map} (\text{flip} (\$) x) fs \\ &\equiv \{\text{composición de funciones}\} \\ (|>) fs x &= (\text{map} . \text{flip} (\$)) x fs \\ &\equiv \{\text{por flip}\} \\ (|>) fs x &= \text{flip} (\text{map} . \text{flip} (\$)) fs x \\ &\equiv \{\eta\text{-reducción}\} \\ (|>) fs &= \text{flip} (\text{map} . \text{flip} (\$)) fs \\ &\equiv \{\eta\text{-reducción}\} \\ (|>) &= \text{flip} (\text{map} . \text{flip} (\$)) \end{aligned}$$

Ejemplos de uso son:

```
MAIN> [(+1), (*2), (*3)] |> 5
[5, 10, 15] :: [Integer]
```

```
MAIN> map (*) [1..10] |> 5
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50] :: [Integer]
```

3.2.4. POLIMORFISMO EN TUPLAS

En el PRELUDE también se definen algunas funciones polimórficas sobre tuplas. Las funciones:

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

devuelven, respectivamente, la primera y segunda componente de un par. Obsérvese que se utilizan dos variables de tipo distintas para las componentes del par, ya que éstas pueden tener tipos distintos. Sin embargo, el tipo del resultado ha de coincidir con el tipo de la componente devuelta.

Ejercicio 3.14 Analice cuál es el tipo más general de las siguientes funciones:

<i>const</i> <i>x</i> <i>y</i>	= <i>x</i>
<i>subst</i> <i>f</i> <i>g</i> <i>x</i>	= <i>f</i> <i>x</i> (<i>g</i> <i>x</i>)
<i>flip</i> <i>f</i> <i>x</i> <i>y</i>	= <i>f</i> <i>y</i> <i>x</i>
<i>curry</i> <i>f</i> <i>x</i> <i>y</i>	= <i>f</i> (<i>x</i> , <i>y</i>)
<i>uncurry</i> <i>f</i> (<i>x</i> , <i>y</i>)	= <i>f</i> <i>x</i> <i>y</i>
<i>pair</i> (<i>f</i> , <i>g</i>) <i>x</i>	= (<i>f</i> <i>x</i> , <i>g</i> <i>x</i>)
<i>cross</i> (<i>f</i> , <i>g</i>) (<i>x</i> , <i>y</i>)	= (<i>f</i> <i>x</i> , <i>g</i> <i>y</i>)

Asimismo, compruebe los resultados con el intérprete.

3.2.5. UN ITERADOR POLIMÓRFICO SOBRE LOS NATURALES

El tipo más general del iterador para números naturales definido en la Sección 3.1.4 es polimórfico:

$$\begin{aligned} \text{iter} &:: (\text{Integer} \rightarrow a \rightarrow a) \rightarrow a \rightarrow \text{Integer} \rightarrow a \\ \text{iter op e } 0 &= e \\ \text{iter op e m} @ (n + 1) &= \text{op m} (\text{iter op e n}) \end{aligned}$$

esto significa que no sólo es posible construir funciones que devuelvan un entero utilizando éste, sino también estructuras arbitrarias:

```
listaDecre :: Integer → [Integer]
listaDecre = iter (:) []
palos :: Integer → String
palos = iter (λ n rs → 'r' : rs) []
```

```
MAIN> listaDecre 5
[5, 4, 3, 2, 1] :: [Integer]
MAIN> palos 3
"III" :: String
```

4 DEFINICIÓN DE TIPOS

4.1. SINÓNIMOS DE TIPO

Un sinónimo de tipo introduce un nuevo nombre para un tipo existente, pero no introduce un nuevo tipo. El nuevo nombre y el antiguo son totalmente intercambiables. Para ello se usa la palabra clave *type*:

```
type Entero = Integer
type DeEnteroEnEntero = Entero → Entero
uno :: Entero
uno = 1
sucesor :: DeEnteroEnEntero
sucesor x = x + 1
type ParFloatantes = (Float, Float)
parCeros :: ParFloatantes
parCeros = (0.0, 0.0)
```

El nuevo nombre de tipo debe comenzar por mayúsculas (todos los *identificadores* de tipo comienzan por mayúsculas en HASKELL). Un sinónimo de tipo muy usado, definido en PRELUDE, es *String*:

```
type String = [Char]
```

de modo que las cadenas de caracteres son realmente listas en HASKELL:

```
nombre :: String
nombre = ['p', 'e', 'p', 'e']
```

aunque HASKELL permite una notación más cómoda: escribir toda la cadena entre comillas dobles:

```
nombreYApellidos :: String
nombreYApellidos = "José E. Gallardo"
```

5 EL SISTEMA DE CLASES DE HASKELL

5.1. TIPOS Y CLASES DE TIPOS. JERARQUÍA DE CLASES

El sistema de clases de HASKELL permite restringir el tipo de ciertas funciones polimórficas imponiendo condiciones a los tipos usados en su declaración. Estas condiciones vienen dadas en forma de predicados que los tipos deben verificar. Por ejemplo, si consideramos la función que determina si un elemento pertenece a una lista, *elem*:

```
elem x []      = False  
elem x (y : ys) = if (x == y) then True else elem x ys
```

vemos que su tipo no debe ser $a \rightarrow [a] \rightarrow \text{Bool}$ siendo a un tipo cualquiera, pues esto permitiría utilizar la función con tipos cuyos datos no son comparables con igualdad; por tanto, se debería poder restringir a aquellos tipos que cumplan dicha condición. El tipo de *elem* es $\text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$, y que leeríamos como: siendo a un tipo instancia de la clase *Eq* (es decir, comparable por igualdad), el tipo de la función *elem* es $a \rightarrow [a] \rightarrow \text{Bool}$.

A lo largo de este capítulo se simultáneará el estudio del sistema de clases de HASKELL con el estudio de clases e instancias concretas de PRELUDE. No obstante, el apartado 5.1.1 realiza una descripción a grandes rasgos del sistema de clases para aquellos que no quieran profundizar en su estudio.

5.1.1. EL SISTEMA DE CLASES

Habitualmente, un tipo de datos T es visto como una tupla:

$$T = (T, \{f, g, \dots\})$$

donde T es una colección de datos y $\{f, g, \dots\}$ es un conjunto de funciones aplicables a T ; consideraremos que f es aplicable a T si tiene algún argumento o resultado con dominio T . En nuestro lenguaje funcional f es aplicable a T si aparece T en el tipo de f .

Interesa *sobrecargar* una colección de funciones aplicables a distintos tipos; por ejemplo, los enteros *Int*, los flotantes *Float*, etc., comparten una serie de funciones aplicables a estos tipos, como $(+)$, $(-)$, $(*)\dots$; en ese caso, tenemos una clase de tipos, en el sentido de que cada elemento de la clase (o instancia) dispone de un conjunto de

valores particulares (o tipo de datos) y una forma también particular de cálculo de cada función de la clase (es decir, de cada función sobrecargada).

Veremos a continuación que los mecanismos del lenguaje hasta ahora vistos no resuelven el problema de la clasificación. Por ejemplo, es deseable que un gran número de tipos (al menos los estandarizados) dispongan de un operador de igualdad ($==$), y deberíamos tener algo como:

```
(==) :: Bool → Bool → Bool
(==) :: Int → Int → Int
```

pero HASKELL no permite redeclarar un objeto; si es posible declarar el operador $(==)$ en forma genérica:

```
(==) :: a → a → Bool
```

donde el hecho de que el identificador a aparezca dos veces en $a \rightarrow a \rightarrow Bool$ significa que los dos operandos de $(==)$ deben ser del mismo tipo. Pero el problema que tenemos ahora es que las ecuaciones para $(==)$ deben utilizar tipos concretos; por ejemplo, podríamos intentar las ecuaciones:

```
0 == 0 = True
0.0 == 0.0 = True -- ERROR
```

y ello no es posible ya que, según se ha visto, las ecuaciones para una misma función han de producir tipos compatibles. Podemos intentar una única ecuación, como la primera:

```
(==) :: a → a → Bool
0 == 0 = True
```

pero en ese caso tendremos que el tipo asociado a $(==)$ es demasiado general, ya que la única ecuación le asigna como tipo $Integer \rightarrow Integer \rightarrow Bool$. En definitiva, el problema no parece tener una solución fácil, salvo que introduzcamos un mecanismo nuevo para ello, como el mecanismo de clases. Así, la declaración de HASKELL:

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
  -- Mínimo a implementar: (==) o bien (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

se interpreta en la forma siguiente: Eq es una clase de tipos, cuyas instancias comparten las funciones *sobre*recargadas $(==)$ y $(/=)$. A $Eq\ a$ se le llama un contexto y la letra a en $Eq\ a$ representa un tipo genérico instancia de la clase, y su uso es para poder parametrizar el tipo de la función sobrecregada $a \rightarrow a \rightarrow Bool$. Cada instancia podrá definir una forma particular de cálculo de la función (aunque puede ocurrir que en la clase se proporcione una definición por defecto para algunas funciones); ello se consigue a través de declaraciones de instancia, como por ejemplo:

```
instance Eq Int where
```

```
  0 == 0 = True
```

```
instance Eq Float where
```

```
  0.0 == 0.0 = True
```

```
...
```

El sistema de clases de HASKELL proporciona una forma estructurada para manejar la sobrecarga y, en general, el polimorfismo; una función es polimórfica (en sentido amplio) si no tiene restricciones para los tipos variables, como por ejemplo las funciones:

```
fst :: (a, b) → a
snd :: (a, b) → b
fst (x, _) = x
snd (_, y) = y
```

donde los tipos a y b pueden ser arbitrarios, y el cálculo de la función no viene determinado por los tipos de sus argumentos. Sin embargo, tal polimorfismo puede ser restringido con un contexto, como en (la función que comprueba la pertenencia):

```
elem :: Eq a ⇒ a → [a] → Bool
```

donde el contexto $(Eq\ a)$ indica que a no puede ser arbitrario, sino que debe ser una instancia de la clase Eq . En general podríamos tener un contexto con varias restricciones, como:

```
f :: (A a, B a, C a) ⇒ [a] → Bool
```

que indica que a debe ser instancia de las clases A , B y C , de forma que tal función tendrá sentido solamente para tales tipos.

Es importante reseñar que cada (función) miembro de una clase recibe implicitamente en su tipo el contexto apropiado; por ejemplo, el tipo de la función $(==)$ es:

```
(==) :: Eq a ⇒ a → a → Bool
```

Las relaciones entre contextos dan lugar a una jerarquía de clases. Por ejemplo, es habitual que el tipo asociado a una función g de una clase B necesite el contexto de otra clase A cuando su cálculo utiliza una función f de ésta:

```
class A a where
  f :: a → Bool
class B a where
  g :: A a ⇒ a → a → a
  g x y | f x = y
        | otherwise = x
```

siendo el tipo de la función g :

$$g :: (A\ a, B\ a) \Rightarrow a \rightarrow a \rightarrow a$$

Pero podemos restringir el tipo parámetro a de la clase B con el contexto $A\ a$ directamente:

```
class A\ a \rightarrow B\ a where
  g :: a \rightarrow a \rightarrow a
  g\ x\ y | f\ x = y
          | otherwise = x
```

siendo ahora el tipo de g :

$$g :: B\ a \Rightarrow a \rightarrow a \rightarrow a$$

En este caso se dice que B es subclase de A . Las clases se organizan en una jerarquía, y PRELUDE establece una jerarquía inicial (ver la Figura 5.1) aunque ésta puede ser ampliada por el programador introduciendo nuevas clases. Una flecha indica la relación subclase; así, Ord es subclase de Eq , y $Real$ es subclase de Num y Ord , y a la vez es superclase directa de $RealFrac$ e $Integral$; Eq es superclase directa de Ord e indirecta de $Real$, etc. Cada clase de tipos tiene predefinida unas instancias (definidas en PRELUDE). Además, el programador puede declarar las suyas propias. Las instancias a las que pertenecen los tipos estandarizados pueden verse en la Figura 5.2. En ella aparece $Ratio_$ para indicar los tipos $Ratio\ Int$, $Ratio\ Integer$, $Ratio\ Float$ y $Ratio\ Double$.

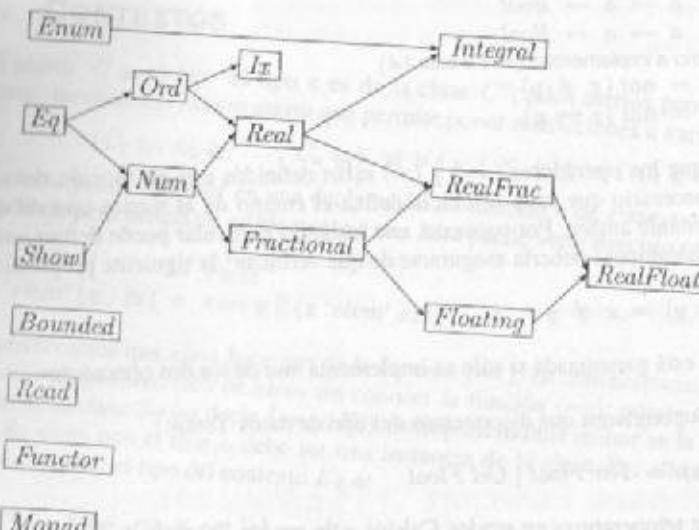


Figura 5.1: Jerarquía de clases de PRELUDE.

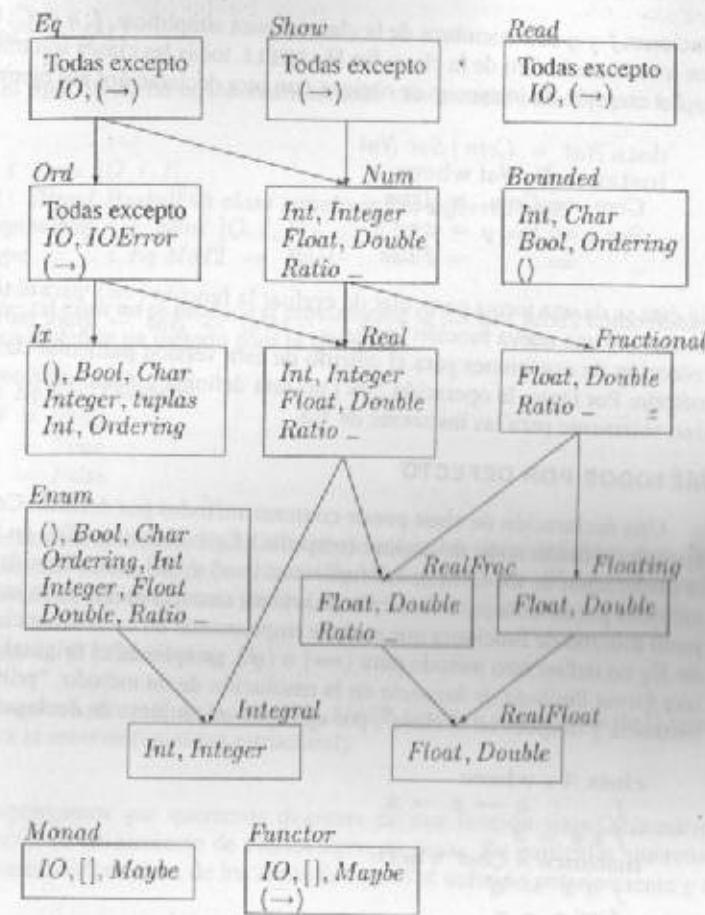


Figura 5.2: Instancias HASKELL estandarizadas para los tipos de PRELUDE.

5.1.2. DECLARACIÓN DE CLASE

Una declaración de clase introduce una colección de funciones sobrecargadas que son compartidas por los tipos de la clase. Por ejemplo, la declaración:

```
class C\ a where
  f :: a \rightarrow Bool
  g :: a \rightarrow Int \rightarrow Bool
```

indica que cada instancia o tipo de esta clase dispone de dos funciones sobrecargadas: las

funciones f y g son miembros de la clase, o para simplificar, $f, g \in C$; también diremos que a es el parámetro de la clase. En HASKELL todas las clases son uniparamétricas.

La creación de instancias se obtiene con otra declaración; por ejemplo:

```
data Nat = Cero | Suc Nat
instance Eq Nat where
  Cero == Cero = True
  Suc x == Suc y = x == y
  _ == _ = False
```

En ésta se da una forma particular de evaluar la función ($==$) para el tipo *Nat*; así pues, tendremos una nueva función $(==)_{Nat} :: Nat \rightarrow Nat \rightarrow Bool$ para tal instancia. La colección de ecuaciones para el cálculo de esta versión particular de ($==$) se llama un método. Por tanto, la operación ($==$) no está definida o disponible para todos los tipos, sino solamente para las instancias de *Eq*.

MÉTODOS POR DEFECTO

Una declaración de clase puede contener métodos por defecto. Consideremos como ejemplo la declaración de la clase (estándar) *Eq* (véase pág. 106) en la cual, además de la declaración de los tipos de las funciones ($==$) y (\neq), miembros de la clase, aparecen métodos por defecto para el cálculo de ambas; como comentario se indica cuál es el conjunto mínimo de funciones que se debe implementar en cada instancia¹. Si una instancia de *Eq* no define otro método para ($==$) o (\neq), se aplicará el original de la clase; esto es una forma limitada de herencia en la resolución de un método: "primero se busca en la instancia y después en la clase"; por ejemplo, el conjunto de declaraciones:

```
class A a where
  f :: a → a → a
  f x y = x
instance A Char where
  f 'h' x = 'q'
  f 'm' x = x
instance A Bool
```

crea una clase *A* y declara dos instancias suyas: *Char* y *Bool*; para *Char* se da el método para *f*, mientras que para *Bool* no, por lo que en una llamada se aplicará el método por defecto:

```
MAIN> f 'h' T
'q' :: Char
MAIN> f 't' y
Program error: instA.v1277.v1281't''y
MAIN> f True False
True :: Bool
```

¹Todas las clases definidas en el PRELUDE incorporan un comentario aclaratorio sobre lo mínimo que se debe implementar para que la instancia esté bien construida.

Obsérvese que para la llamada *f 't' y* no se aplica el método por defecto.

DERIVACIÓN AUTOMÁTICA DE INSTANCIAS

Para un tipo se pueden declarar instancias por defecto de varias clases. Esto se hace a través de la cláusula *deriving* en la definición del tipo. Esta cláusula provoca que el tipo sea instancia de todas las clases que aparezcan detrás (si hay más de una se colocan entre paréntesis y separadas por comas). Estas instancias se comportan atendiendo a la estructura de los constructores de datos que forman parte del tipo y por tanto se generan instancias de las funciones miembros en forma estructural. Así, por ejemplo, la declaración:

```
data Color = Rojo | Amarilla | Azul | Verde deriving Eq
```

hace que el tipo *Color* sea instancia de la clase *Eq* de manera que el *Rojo* es igual al *Rojo*, el *Amarillo* al *Amarillo*, etc. Es posible crear instancias automáticas de ciertas clases definidas en PRELUDE (*Eq*, *Ord*, *Show*, *Read*, *Ix*, *Enum*, y *Bounded*).

5.1.3. LA CLASE *Eq* DE PRELUDE

Ya se ha hecho mención de la clase *Eq* cuyas instancias pueden compararse a través del operador de igualdad. Repetimos la definición completa de la clase:

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
  -- Mínimo a implementar: (==) o bien (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

donde vemos que los operadores ($==$) y (\neq) están definidos uno en función del otro. Por tanto, es necesario que cada instancia defina el cuerpo de al menos uno de ellos para tener disponible ambos. Por supuesto, una instancia particular puede definir ambos operadores pero entonces debería asegurarse de que verifican² la siguiente propiedad:

$$\text{not } (x == y) = x \neq y$$

Esta propiedad está garantizada si sólo se implementa uno de los dos operadores.

Ejemplo 5.1 Supongamos que disponemos del tipo de datos *Temp*:

```
data Temp = Far Float | Cel Float
```

para representar temperaturas en grados Celsius o en grados Fahrenheit: *Far 10* representa una temperatura de 10° Fahrenheit y *Cel 32* representa 32° Celsius. Por otro lado, disponemos de una función de conversión que traduce grados Celsius a Fahrenheit y viceversa:

² Esta restricción no es verificada por el intérprete, por lo que es el programador el que debe asegurar dicha consistencia.

```

convTemp :: Temp → Temp
convTemp (Far x) = Cel (((x - 32.0)/9.0) + 5.0)
convTemp (Cel x) = Far ((x * 9.0)/5.0 + 32.0)

```

Para poder utilizar el operador de igualdad (`==`) con las temperaturas necesitamos que sea instancia de la clase `Eq`. Ahora bien, sabemos que `Far 32` es lo mismo que `Cel 0`. Una instancia automática de `Eq` para este tipo podría detectar que `Far 32` es igual que `Far 32` (tienen la misma estructura) pero nunca que es igual a `Cel 0`. En este caso no interesa crear la instancia de forma estructural ya que no capturaría las igualdades y de igualdades entre grados expresados en distintas escalas. Por tanto, creamos la instancia manualmente:

```

instance Eq Temp where
  Far x == Far y = x == y
  Cel x == Cel y = x == y
  t1 == t2 = convTemp t1 == convTemp t2

```

que nos permite comparar temperaturas independientemente de la forma en que vienen representados los datos. Con tal definición, las temperaturas podrán ser usadas en cualquier función polimórfica en la que se requiera el contexto `Eq`, como por ejemplo la función `elem` vista anteriormente:

```

MAIN> elem (Cel 0) [Cel 10, Far 32, Cel 20]
True :: Bool

```

5.2. CONTEXTOS

El aserto $C \in a$ se lee: "el tipo a es de la clase C "; tales asertos permiten escribir contextos; un contexto es otro aserto que permite poner restricciones a varios tipos; así:

$$(C_1 t_1, C_2 t_2, \dots, C_n t_n) \equiv \forall i . 1 \leq i \leq n . C_i t_i$$

Un contexto puede aparecer en una declaración de función o de clase; por ejemplo, el predicado de pertenencia de un elemento a una lista puede venir descrito en la forma:

```

x `elem` [] = False
x `elem` (y : xs) = x == y || (x `elem` xs)

```

donde observamos que `elem` hace uso de la función (`==`); en consecuencia, no es posible generar ninguna versión de `elem` sin conocer la función (`==`); puesto que (`==`) es miembro de la clase `Eq` (es decir, $(==) \in Eq$), es imprescindible incluir en la declaración de tipo de `elem` que el tipo a debe ser una instancia de la clase `Eq`; ello se consigue haciendo preceder el tipo del contexto `Eq a`:

```

elem :: Eq a => a → [a] → Bool

```

y la declaración se leerá ahora: "para cualquier tipo a de la clase `Eq`, `elem` tiene el tipo $a \rightarrow [a] \rightarrow Bool$ "; en consecuencia, existirá una versión de `elem` para cada a de la clase `Eq`. Veamos otro ejemplo; sea:

5.2 - Contextos

```
data Mod2 = O | I
```

de forma que el tipo `Mod2` no es automáticamente una instancia de `Eq`, por lo que la llamada:

```

MAIN> I `elem` [O, I, I]
ERROR : Illegal Haskell 98 class constraint in inferred type
*** Expression : I `elem` [O, I, I]
*** Type      : Eq Mod2 → Bool

```

produce un error; tal error no se produce si previamente se declara `Mod2` como instancia de `Eq`, y de paso se define un método para la igualdad:

```

instance Eq Mod2 where
  O == O = True
  I == I = True
  _ == _ = False

```

Como ya hemos mencionado, HASKELL permite que los tipos definidos con `data` sean automáticamente instancias de las clases `Eq`, `Ord`, `Ix`, `Enum`, `Bounded`, `Read` o `Show`; ello se consigue con la cláusula `deriving`:

```
data Mod2 = O | I deriving Eq
```

y en este caso se genera, además de la instancia `Mod2`, también una versión de la igualdad equivalente a la anterior (igualdad estructural).

Ejemplo 5.2 Supongamos que queremos disponer de una función `numCiclomático` que calcule el nivel de anidamiento de ciertos tipos de datos. En particular, queremos calcular los números ciclomáticos de los datos de tipo `Nat` definido anteriormente y de los del tipo:

```
data Árbol a = Vacío | Hoja a | Nodo (Árbol a) a (Árbol a)
```

Con objeto de no tener que dar a la función un nombre particular para cada tipo de datos, se va a crear una clase `Cic` que disponga de la función `numCiclomático` y se construirán para ambos tipos instancias de esta clase. Cada instancia determinará cómo calcular el número ciclomático con una versión diferente de la función:

```

class Cic a where
  numCiclomático :: a → Int

```

```

instance Cic Nat where
  numCiclomático Cero = 0
  numCiclomático (Suc x) = 1 + numCiclomático x

```

```
instance Cic (Árbol a) where
    numCiclomático Vacío = 0
    numCiclomático (Hoja _) = 1
    numCiclomático (Nodo i d) = 1 + max (numCiclomático i)
                                (numCiclomático d)
```

A partir de esta definición, tenemos el diálogo siguiente:

```
MAIN> numCiclomático (Suc (Suc (Suc Cero)))
3 :: Int
MAIN> numCiclomático (Nodo (Hoja 5) 7 (Nodo (Hoja 4) 8 Vacío))
3 :: Int
```

El tipo completo es `numCiclomático :: Cic a → a → Int.`

Tipos

Ejemplo 5.3 Sea la clase `Unidades` que contiene dos funciones constantes `cero` y `uno` de la que se van a definir instancias para varios tipos numéricos:

```
class Unidades a where
    una :: a
    cero :: a

instance Unidades Int where
    uno = 1
    cero = 0

instance Unidades Double where
    uno = 1.0
    cero = 0.0
```

Ahora se puede definir una función `moveHaciaCero` que mueve su argumento una unidad hacia `cero` (puede que se pase y le cambie de signo para el caso de un `Double`) y lo deja igual si ya era `cero`:

```
moveHaciaCero :: (Unidades a, Ord a, Num a) ⇒ a → a
moveHaciaCero x | x > cero = x - uno
                 | x < cero = x + uno
                 | otherwise = x
```

Vemos que la función exige al tipo de su argumento varias cosas; en primer lugar que tenga definido `cero` y `uno`, es decir, que sea instancia de `Unidades`; en segundo lugar exige que sus datos sean comparables con (`<`) y (`>`), es decir, que sea instancia de la clase `Ord`; por último, también deben operarse con (`+`) y (`-`), lo que exige que sea instancia de la clase `Num`. Por eso, el tipo de la función `moveHaciaCero` tiene el contexto antes descrito. Con esta función es posible el siguiente diálogo:

```
MAIN> moveHaciaCero (3.4 :: Double)
2.4 :: Double
```

5.3 - Subclases. La clase `Ord` de PRELUDE

```
MAIN> moveHaciaCero (-4.1 :: Double)
- 3.1 :: Double
MAIN> moveHaciaCero (2 :: Int)
1 :: Int
```

Ejemplo

5.2.1. INSTANCIAS PARAMÉTRICAS

El uso de contextos permite también generar instancias en forma paramétrica; por ejemplo, si un tipo `a` dispone de la igualdad, es deseable que el tipo `[a]` disponga también de una igualdad; para ello se declaran las listas como instancias genéricas restringidas:

```
instance Eq a ⇒ Eq [a] where
    [] == [] = True
    (x : xs) == (y : ys) = x == y && xs == ys
    _ == _ = False
```

donde observamos que el contexto `Eq a` permite el uso de la igualdad de los elementos del tipo base, y por extensión, la igualdad de listas. Así, no todas las listas son instancias de `Eq`, sino aquéllas cuyo tipo base sea instancia de `Eq`.

Obsérvese la potencia de la creación de instancias en forma genérica. Veamos otro ejemplo de ello; sea el tipo `Árbol a` definido anteriormente, si queremos generar `Árbol a` como instancia de `Eq`, donde la igualdad sea la estructural, podemos escribir:

```
instance Eq a ⇒ Eq (Árbol a) where
    Vacío == Vacío = True
    Hoja x == Hoja y = x == y
    (Nodo a x b) == (Nodo a' x' b') = (x == x') && (a == a') && (b == b')
    _ == _ = False
```

aunque en este caso se podría haber derivado directamente la instancia:

```
data Árbol a = Vacío |
               Hoja a |
               Nodo (Árbol a) a (Árbol a) deriving Eq
```

y la igualdad se toma también en sentido estructural.

5.3. SUBCLASES. LA CLASE `Ord` DE PRELUDE

Una declaración de clase precedida de un contexto permite utilizar los miembros de las clases del contexto en la declaración de los miembros de la clase. Por ejemplo, tenemos la clase estándar `Ord` definida en PRELUDE (ver la Figura 5.3) cuyos miembros son (`<`), (`≤`), (`>`), (`≥`), `max`, `min` y `compare`; como se observa, todas las desigualdades tienen métodos por defecto basados en `compare`. Por otro lado, `compare` utiliza el tipo `Ordering` cuya definición es:

```

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (≤), (≥), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
  -- Mínimo a implementar: (≤) o compare
  compare x y | x == y = EQ
    | x ≤ y = LT
    | otherwise = GT
  x ≤ y = compare x y ≠ GT
  x < y = compare x y == LT
  x ≥ y = compare x y ≠ LT
  x > y = compare x y == GT
  max x y | x ≥ y = x
    | otherwise = y
  min x y | x ≤ y = x
    | otherwise = y

```

Figura 5.3: La clase *Ord*.

```

data Ordering = LT | EQ | GT
deriving (Eq, Ord,Ix,Enum,Read>Show,Bounded)

```

Una instancia debe definir el operador (\leq) o, si lo prefiere, simplemente debe definir el método *compare*.

El contexto *Eq a* en la declaración *class Eq a => Ord a where ...*, permite utilizar los miembros (visibles) de la clase *Eq* (esto es, ($=$) y (\neq)) para definir métodos por defecto en la clase, como se observa en la definición de *compare* en la que se utiliza ($=$) y, en la terminología de HASKELL, diremos que *Ord* es una subclase de *Eq*; así, una subclase tiene visibilidad sobre los métodos de la clase. Esto no significa herencia ya que las instancias de una clase *B* que es subclase de *A* no son automáticamente instancias de *A*; intentar crear una instancia de una clase sin crear la instancia de las superclases produce un error de compilación. Ahora bien, perfectamente puede hacerse:

```

data Mod2 = O | I deriving Ord
instance Eq Mod2

```

que permite realizar comparaciones con los elementos de este tipo.

Al definir la instancia de *Ord* por defecto se utilizará el siguiente criterio de comparación: un dato es menor que otro si su constructor de datos aparece más a la izquierda en la definición del tipo, y entre datos con el mismo constructor se comparan de esta forma: los datos también de izquierda a derecha. A este orden lo llamaremos *orden lexicográfico*.

Con el criterio de ordenación estructural, *O* es menor que *I*. Por otro lado, ninguno de los operadores (igual y distinto) de la clase *Eq* se utilizarán para la comparación de igualdad los miembros por defecto, lo que provocaría un des-

bordamiento de pila (*stack overflow*) ya que se producen llamadas alternadas a (\neq) y ($=$).

Siguiendo con el ejemplo del apartado anterior, se podrían definir los árboles como instancias de *Ord* directamente:

```

data Árbol a = Vacío | Hoja a | Nodo (Árbol a) a (Árbol a)
deriving (Eq, Ord)

```

y, como el orden es estructural, los constructores se ordenan según aparecen en la definición y, en caso de necesidad, los argumentos se comparan de izquierda a derecha. En este caso, el operador (\leq) que se genera es equivalente al de la derivación:

```

instance Ord a => Ord (Árbol a) where
  Vacío ≤ _ = True
  Hoja x ≤ Hoja x' = x ≤ x'
  Hoja _ ≤ Nodo _ _ = True
  Nodo a x b ≤ Nodo a' x' b' = (a ≤ a') ||
    (a == a') && (x ≤ x') ||
    (a == a') && (x == x') && (b ≤ b')
  _ ≤ _ = False

```

Es importante señalar que si la definición del dato hubiera sido:

```

data Árbol a = Hoja a | Vacío | Nodo (Árbol a) a (Árbol a)
deriving (Eq, Ord)

```

entonces, cualquier hoja del árbol sería menor que el árbol vacío.

Podemos decir que el mecanismo de subclasificación no implica una herencia para las instancias, sino solamente introduce una noción de herencia en visibilidad, que, por otro lado, se consigue con un contexto (por lo que el uso de la palabra subclase quizás no sea apropiado).

Un tipo puede ser instancia de varias clases sean disjuntas o no; para ello basta describir, si es necesario, un contexto con varias clases:

```

class (C1 a, C2 a, ..., Cn a) => C a where ...

```

y en ese caso tendremos que en los miembros de *C* pueden aparecer llamadas a miembros de las clases *C_i*.

Para el tipo *Temp* ya descrito (véase el Ejemplo 5.1) no podemos derivar una instancia por defecto de *Ord* y, al igual que con *Eq*, debemos crear la instancia manualmente:

```

instance Ord Temp where
  Far x ≤ Far y = x ≤ y
  Cel x ≤ Cel y = x ≤ y
  t1 ≤ t2 = convTemp t1 ≤ t2

```

Ahora es posible comparar dos temperaturas, ordenar los elementos de una lista de temperaturas, construir un árbol binario de búsqueda cuyos elementos son temperaturas, etc.

5.3.1. UN EJEMPLO: LOS ENTEROS MÓDULO n

Los enteros módulo n quedan caracterizados por ser un grupo cíclico; dos funciones *sucesor* y *predecesor* (periódicas) permiten definir, entre otras, las operaciones algebraicas (+) y (-) del grupo; podemos considerar tales enteros como instancias de una clase *Ciclo* algo más general, que sea subclase de *Eq* y de *Unidades*:

```
class (Eq a, Unidades a) => Ciclo a where
  suc      :: a -> a -- sucesor
  pre      :: a -> a -- predecesor
  miembros :: [a] -- miembros del tipo a
  noCeros  :: [a] -- miembros no ceros
  noCeros = miembros \\ [cero]
```

Nota 5.4 El operador (`\\"`) (definido en el módulo *List*) devuelve la lista que resulta de eliminar de la lista primer argumento cada elemento de la lista segundo argumento.

Por ejemplo, el tipo *Mod6* (enteros módulo 6):

```
data Mod6 = O | I | II | III | IV | V deriving Eq
```

es, por construcción, una instancia de *Eq*, y si lo declaramos también como instancia de *Unidades*:

```
instance Unidades Mod6 where
  cero = O
  uno = I
```

podemos ya generar una instancia de *Ciclo*:

```
instance Ciclo Mod6 where {
  miembros = [O, I, II, III, IV, V];
  suc O = I; suc I = II; suc II = III;
  suc III = IV; suc IV = V; suc V = O;
  -- pre O = V; pre I = O; pre II = I;
  -- pre III = II; pre IV = III; pre V = IV }
```

y por tanto se produce el siguiente diálogo:

```
MAIN> elem I noCeros
True :: Bool
```

```
MAIN> elem O noCeros
False :: Bool
```

5.3.2. INTERSECCIÓN DE CLASES

El mecanismo de clases permite crear una clase intersección de otras. En la definición se puede enriquecer tal intersección con funciones miembro nuevas, o bien considerar una intersección pura, como en:

5.4 - Visualizando y leyendo datos. *Read y Show*

```
class (Eq a, Unidades a, Ciclo a) => CicloAlgebraico a
```

Recordemos una vez más que las instancias no se generan de forma automática y, por consiguiente, el hecho de ser *Mod6* instancia de las clases *Eq*, *Unidades* y *Ciclo*, no implica que lo sea de *CicloAlgebraico* directamente, salvo que se declare como tal:

```
instance CicloAlgebraico Mod6
```

Como se observa, una ventaja de la intersección es la sustitución de un contexto por otro más simple, y tenemos las dos declaraciones alternativas:

```
f :: (Eq a, Unidades a, Ciclo a) => [a] -> Bool
f :: (CicloAlgebraico a) => [a] -> Bool
```

5.4. VISUALIZANDO Y LEYENDO DATOS. *Read Y Show*

Cuando se evalúa una expresión es natural querer visualizar el resultado de la misma. Este dato resultado debe ser presentado en la pantalla en forma de cadena de caracteres para que su lectura sea fácil y entendible. Para ello, en PRELUDE se define la clase *Show*. Los datos correspondientes a las instancias de esta clase son convertibles a cadenas (de caracteres) a través del método *show*:

```
type ShowS = String -> String
class Show a where
  show    :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList :: [a] -> ShowS
  -- Mínimo: show <: showsPrec
  show x     = showsPrec 0 x ***
  showsPrec _ x s = show x ++ s
```

donde para simplificar hemos introducido el sinónimo de tipo *ShowS*. Todos los tipos básicos son instancias de esta clase (ver la Figura 5.2)³.

Si podemos mostrar un dato de un tipo base, también podemos definir un procedimiento para mostrar un dato estructurado sobre ese tipo base; un ejemplo sencillo lo proporcionan los árboles. Si utilizamos una representación de árboles como la siguiente:

```
data Árbol a = Vacío | Hoja a | Nodo (Árbol a) a (Árbol a)
```

podemos escribir nuestro propio *show* construyendo una instancia de la clase *Show* para el tipo *Árbol a*. Previamente, necesitamos que el tipo base ya sea "mostrable", es decir,

³Es conveniente derivar cualquier tipo como instancia de esta clase con objeto de poder disponer de una representación de sus datos. HUOS utiliza esa representación para visualizar los resultados por defecto.

que sea también instancia de la clase `Show`. A la vista de las declaraciones por defecto será suficiente definir `showPrec` o `show`. Si lo hacemos a través de `show`, podemos escribir:

```
instance Show a => Show (Árbol a) where
    show Vacío = "o"
    show (Hoja x) = show x
    show (Nodo i x d) = "<" ++ show i ++
                        "|" ++ show x ++
                        "|" ++ show d ++
                        ">"
```

de forma que:

```
MAIN> show (Nodo (Nodo Vacío 1 Vacío) 2 (Hoja 3))
"<<0|1|o>|2|3>" :: [Char]
```

REDUCIENDO LA COMPLEJIDAD. FUNCIONES "MOSTRADORAS"

Puesto que `(++)` tiene complejidad lineal, la función `show` definida para el tipo `Árbol` tiene complejidad cuadrática; podemos reducir la complejidad a lineal con una función auxiliar. Así, en PRELUDE aparece:

```
shows :: Show a => a -> Shows
shows = showsPrec 0
```

que es una función que toma dos argumentos; el primero es el valor a mostrar y el segundo una cadena. Como vemos, esta función se define por medio de `showsPrec`, que es una función mostradora cuyo primer argumento es un índice de precedencia⁴, el segundo es el valor a mostrar y el tercero una cadena. La función `showsPrec` debe introducir (previa conversión) el valor (segundo argumento) al principio de la cadena (tercer argumento) devolviendo la cadena resultante. Este tercer argumento actúa como un acumulador y es el que reduce la complejidad. La función mostradora para una estructura se obtiene por composición (funcional) de las funciones mostradoras de sus subestructuras. Obsérvese la definición de `showsPrec` para `Nodo` en la siguiente instancia. Con esto, en lugar de definir el método particular `show` para árboles, podemos definir `showsPrec`:

```
instance Show a => Show (Árbol a) where
    showsPrec _ Vacío = showChar 'o'
    showsPrec _ (Hoja x) = shows x
    showsPrec _ (Nodo i x d) = (showChar '<').(shows i).
                                (showChar '|').(shows x).
                                (showChar '|').(shows d).(showChar '>')
```

donde la función `showChar` está definida en PRELUDE como:

⁴Aunque en la práctica no se usa y se le asigna siempre 0, cuando se usa, se utiliza para controlar el nivel de anidamiento de una estructura y así tabular, colocar paréntesis, ...

```
showChar :: Char -> Shows
showChar = (:)
```

Obsérvese que ahora la complejidad es lineal. En cualquier caso, esta derivación de instancia normalmente se suele declarar por defecto vía la cláusula `deriving Show`, obteniendo así una representación estructural de los datos del tipo. Por ejemplo:

```
data Árbol a = Vacío | Hoja a | Nodo (Árbol a) a (Árbol a) deriving Show
```

La diferencia es que al derivar de forma automática, el sistema "muestra" la estructura en forma literal; así, con esta última declaración tendremos:

```
MAIN> show (Nodo Vacío 3 Vacío)
"Nodo Vacío 3 Vacío" :: [Char]
```

mientras que antes obteníamos:

```
MAIN> show (Nodo Vacío 3 Vacío)
"<<0|3|o>" :: [Char]
```

Del mismo modo, para el tipo de datos `Temp` podemos derivar una instancia de la clase `Show` automáticamente:

```
data Temp = Far Float | Cel Float deriving Show
```

y ahora, se tiene:

```
MAIN> show (Far 32)
"Far 32" :: [Char]
MAIN> show (Cel 15)
"Cel 15" :: [Char]
```

LA CLASE `Read`

El problema inverso, es decir, construir datos a partir de cadenas de caracteres, se resuelve con la clase `Read` y el sinónimo de tipo `ReadS`:

```
type ReadS a = String -> [(a, String)]
class Read a where
    readsPrec :: Int -> ReadS a
    readList :: ReadS [a]
    -- Minimally: readsPrec
    ...
```

donde la función `readsPrec` realiza una lectura de datos en forma acumulada que permite descomponer el dato de entrada para generar un *análizador*. Para leer un dato se puede utilizar la función `read` de PRELUDE.

```
read :: Read a => String a -> a
```

Si ahora se intenta mantener el siguiente diálogo:

```
MAIN> read "23"
```

se produce el error:

```
ERROR : Unresolved overloading
*** Type : Read a => a
*** Expression : read "23"
```

debido a que no es posible inducir el tipo del resultado. Si le ayudamos a decidirlo no se produce ningún problema:

```
MAIN> (read "23") :: Int
23 :: Int
```

Al igual que para la clase *Show*, en PRELUDE se define una función:

```
reads :: Read a => ReadS a
```

cuyo tipo equivale a *reads* :: *Read a* \Rightarrow *String* \rightarrow $[(a, String)]$. Ésta extrae de la cadena de entrada los datos necesarios para construir un dato del tipo solicitado, dejando en la cadena el resto no utilizado. El tipo solicitado debe quedar patente dentro de la expresión donde se utilice *reads*:

```
MAIN> 3 + u where [(u, v)] = reads "23kk"
26 :: Integer
```

y si no es así, deberá indicarse explícitamente:

```
MAIN> (reads "23kk") :: [(Int, String)]
[(23, "kk")] :: [(Int, String)]
```

El Capítulo 14 realiza un estudio detallado de esta clase y construye un analizador.

5.5. LAS CLASES *Num*, *Integral* Y *Fractional* DE PRELUDE

Estas clases están definidas para sobrecargar ciertas operaciones aritméticas de modo que todos los tipos numéricos puedan utilizarlas. Así, la clase *Num* define los métodos correspondientes a las operaciones aritméticas elementales: suma, resta, multiplicación pero no la división, etc. La definición es:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  -- Minima definición: todos, excepto negate o (-)
  x - y = x + negate y
  negate x = 0 - x
```

Las instancias definidas para esta clase en PRELUDE se muestran en la Figura 5.2. Como puede verse, esta clase no define la división que se deja para las clases *Integral* y *Fractional*. *Integral* define la división entera y *Fractional* define la división fraccionaria⁵.

5.5.1. LOS TIPOS NUMÉRICOS DE HASKELL

Entre los tipos numéricos que PRELUDE incluye cabe destacar los siguientes: *Int* que representan enteros cortos (32 bits), *Integer* que representan enteros de precisión ilimitada (enteros largos), flotantes (*Float*) y flotantes de doble precisión (*Double*). Además, existe un módulo estandarizado llamado *Ratio* que incluye la definición de números racionales sobre cada uno de los tipos de enteros mencionados (*Ratio Int* y *Ratio Integer*). En lo que sigue consideraremos que este módulo se está utilizando e incluiremos sus tipos en nuestras deliberaciones⁶.

Todas las constantes numéricas en HASKELL están sobrecargadas por lo que su tipo dependerá del contexto en que se encuentren. Así, la constante 2 puede interpretarse como *Int*, *Integer*, *Ratio Int*, *Ratio Integer*, *Float* o *Double*, mientras que la constante 2.1 puede interpretarse como *Ratio Int*, *Ratio Integer*, *Float* o *Double*. Por ejemplo, la función:

```
inc :: Integer -> Integer
inc x = x + 1
```

determina que la constante 1 que aparece en el cuerpo de la definición sea de tipo *Integer*, y la llamada:

```
MAIN> inc 7
8 :: Integer
```

determina con claridad que el número 7 debe tener tipo *Integer*. Ahora bien, si definimos:

```
inc' :: Double -> Double
inc' x = x + 1
```

es claro que ahora el 1 del cuerpo de la definición es interpretado como un dato de tipo *Double*. Además, la llamada:

⁵Para una descripción del resto de clases numéricas puede consultarse el informe de HASKELL.

⁶Existen otros módulos estandarizados que proporcionan más tipos numéricos como complejos, etc.

```
MAIN> inc' 7
8.0 :: Double
MAIN> inc' 2.1
3.1 :: Double
```

determina que tanto 7 como 2.1 son también de tipo *Double*. Para resolver el problema de la interpretación de los tipos de los datos numéricos, HASKELL utiliza el siguiente criterio:

- Todo número no decimal es equivalente a la aplicación de la función *fromInteger* a ese número visto como *Integer*.
- Todo número decimal es equivalente a la aplicación de la función *fromRational* a ese número visto como *Ratio Integer* (un sinónimo de *Ratio Integer* es *Rational*).

Por ello, la ecuación por defecto $\text{negate } x = 0 - x$ se interpreta como:

$$\text{negate } x = \text{fromInteger } 0 - x$$

Además:

```
7
=> { Por la traducción de constantes }
    fromInteger 7
=> { ya que fromInteger :: Num a => Integer -> a }
    fromInteger 7 :: Num a => a
```

También,

```
2.1
=> { Por la traducción de constantes }
    fromRational 2.1
=> { ya que fromRational :: Fractional a => Rational -> a }
    fromRational 2.1 :: Fractional a => a
```

De manera que la reducción que se produce para evaluar *inc'* 2.1 es:

```
inc' 2.1
=> { Tratamiento de las constantes }
    inc' (fromRational 2.1 :: Fractional a => a)
=> { El argumento de inc' debe ser de tipo Double }
    (2.1 :: Double) + (fromInteger 1 :: Num a => a)
=> { (+) :: (Num a => a -> a -> a). Se toma fromInteger de la clase Double }
    (2.1 :: Double) + (1 :: Double)
=> { suma para el tipo Double }
    3.1 :: Double
```

donde el paso clave está en encontrar cuál es la instancia de la clase *Num* de la que debe obtener la función *fromInteger* para aplicar a la constante 1; dado que esta expresión es el segundo argumento de una suma y dicha suma tiene como primer argumento un dato de tipo *Double*, se hace necesario que este segundo argumento también sea *Double*, de ahí el hecho de que en el siguiente paso, el 1 aparezca con tipo *Double*.

5.5.2. AMBIGÜEDAD EN LAS CONSTANTES NUMÉRICAS

En los ejemplos anteriores hemos visto que por el contexto era relativamente fácil resolver los tipos a los que pertenecen las constantes y, por tanto, los tipos de las expresiones. Sin embargo hay otras ocasiones en la que el contexto de una expresión no determina con claridad el tipo que deben tener las constantes que aparecen en ella. Ese es el caso de la expresión $2 + 3$. En este caso se produce una ambigüedad, pues dado que la traslación por la equivalencia de las constantes lleva a la expresión:

$$(\text{fromInteger } 2 :: \text{Num } a \Rightarrow a) + (\text{fromInteger } 3 :: \text{Num } a \Rightarrow a)$$

no sabemos a cuál de las instancias de *Num* se refieren dichas constantes.

La forma más sencilla de resolver este problema es indicar explícitamente el tipo de las constantes en una expresión. Por ejemplo en el siguiente diálogo no se produce ninguna ambigüedad:

```
MAIN> (2 :: Int) + (3 :: Int)
5 :: Int
MAIN> (2 + 3) :: Int
5 :: Int
MAIN> (2 :: Int) + 3
5 :: Int
```

aunque no parece la solución más satisfactoria. Para evitar tener que estar indicando el tipo de las constantes en cada expresión que pueda producir ambigüedad, HASKELL permite definir tipos numéricos por defecto que serán aplicables en estos casos. Esto se hace a través de la cláusula *default*. Esta cláusula debe aparecer dentro de un módulo y debe ir seguida por una serie de tipos numéricos (sin variables) separados por comas y entre paréntesis. Cuando una constante quede ambigua dentro de una expresión, se interpretará como perteneciente al primer tipo de esa serie que verifique el contexto en el que se encuentra la expresión. Si no existe ninguno que lo verifique se obtendrá un error de sobrecarga no resuelta. Así, por ejemplo si se activa la cláusula *default* (*Integer*, *Double*), entonces:

```
MAIN> 2 + 3
5 :: Integer
MAIN> 2.1 + 5.2
7.3 :: Double
```

Con la cláusula *default* anterior, la reducción de la expresión $2 + 3.1$ será:

$$2 + 3.1$$

```

⇒ { Tratamiento de las constantes }
  (fromInteger 2 :: Num a ⇒ a) +
  (fromRational 3.1 :: Fractional a ⇒ a)
⇒ ((+) :: Num a ⇒ a → a → a) +
  (fromInteger 2 :: Num a ⇒ a) +
  (fromRational 3.1 :: Fractional a ⇒ a)) :: Num a ⇒ a
⇒ { Ambigüedad. Se utiliza fromInteger y fromRational de Double }
  (2 :: Double) + (3.1 :: Double) :: Double
⇒ { suma para el tipo Double }
  5.1 :: Double

```

Si la cláusula `default` hubiera sido:

```
default (Integer, Ratio Int, Double)
```

el resultado para la expresión anterior hubiera sido `5.1 :: Ratio Int` (más exactamente `10695475 % 2097152 :: Ratio Int`); y si la cláusula hubiera sido:

```
default (Int, Integer)
```

se produciría un error de sobrecarga no resuelta pues ninguno de los tipos por defecto satisface el contexto.

Si un módulo no utiliza ninguna cláusula `default` se considera que se está utilizando `default (Integer, Double)`, es decir, ésta es la cláusula `default` por defecto. En realidad, cuando estamos trabajando sin módulos, ésta es la cláusula que se utiliza.

5.5.3. PROMOCIÓN NUMÉRICA

A veces lo que interesa es convertir un dato que se sabe que es de un tipo dado a otro tipo. Por ejemplo, puede que nos interese dividir dos expresiones de tipo `Int` pero obteniendo un resultado de tipo `Double` (división no entera). Para estos casos, se utilizan las funciones:

```
fromIntegral :: (Integral a, Num b) ⇒ a → b
fromIntegral = fromInteger . toInteger
```

```
realToRational :: (Real a, Fractional b) ⇒ a → b
realToRational = fromRational . toRational
```

La primera de ellas convierte la expresión a `Integer` y luego le aplica `fromInteger`. La segunda convierte la expresión a `Rational` y luego le aplica `fromRational`. Dependiendo del contexto, se usarán las funciones sobrecargadas `fromInteger` y `fromRational` de una clase u otra.

Como es fácil observar, estas funciones permiten tratar a las expresiones de cualquier tipo como si fueran constantes, por lo que es aplicable todo lo dicho para constantes en el apartado anterior. Así, por ejemplo, dadas las definiciones (y supuesto el `default` por defecto):

5.5 - Las clases `Num`, `Integral` y `Fractional` de PRELUDE

```

m :: Int
m = 5

frec :: Fractional a ⇒ Int → a
frec n
| n < m   = fromIntegral n / fromIntegral (m + 1)
| otherwise = fromIntegral n / fromIntegral (m + 2)

```

las reducciones para `frec 2` serán:

```

frec 2
⇒ { Definición de frec }
fromIntegral 2 / fromIntegral (5 + 1)
⇒ ...
⇒ ...
0.333333 :: Double

```

Desgraciadamente, esto no funciona si en el contexto de la expresión se exige al tipo sobrecargado la pertenencia a alguna clase no numérica (o definida por el usuario). Por ejemplo, dadas las definiciones:

```

class Mía a where
  f :: a → Integer
  f x = 1
instance Mía Integer
instance Mía Int

```

se produce el siguiente error en el diálogo:

```

MAIN> f 3
ERROR: Unresolved overloading
*** Type : (Num a, Mía a) ⇒ Integer
*** Expression : f 3

```

porque el contexto exige a 3 la pertenencia a la clase `Num` y a la clase `Mía`. Y es que para seleccionar una instancia de `Num` para dar el tipo a 3 debe recurrir al mecanismo por defecto de control de ambigüedad, pero éste está inhabilitado a causa de que aparece `Mía` en ese contexto. Este problema queda perfectamente resuelto si proporcionamos de forma explícita el tipo de la constante, de manera que no se utilice el mecanismo por defecto:

```

MAIN> f (3 :: Int)
3 :: Integer

```

Existen otras clases numéricas como `Real`, `RealFrac`, `RealFloat` o `Floating` de las que sólo mencionaremos algunos métodos interesantes. De la clase `Real` ya hemos mencionado el método `toRational`; de la clase `Floating` destacar que incluye métodos para

operaciones trigonométricas, logarítmicas y exponenciales; de la clase *RealFrac* descubrimos los métodos que convierten un dato *RealFrac* a un dato *Integral* como *truncate*, *round*, *ceiling* y *floor*, que truncan o redondean los decimales de su argumento. Por ejemplo, la siguiente función devuelve su argumento truncado de manera que a lo sumo tiene presentes dos decimales:

```
dosDec :: RealFrac a => a -> a
dosDec = (/100).fromInteger.truncate.(*100)
```

5.5.4. EJEMPLO: LOS RACIONALES COMO INSTANCIAS GENÉRICAS

Aunque el módulo *Ratio* proporciona tipos racionales sobre varios tipos numéricos, podemos querer introducir otros números racionales definidos por nosotros. Éstos pueden definirse como una subclase de *Num* o bien como instancias genéricas; lo haremos de esta última forma. Realmente nos interesa solamente una operación nueva: la normalización (eliminación de factores comunes al numerador y denominador); tal normalización dependerá esencialmente del concepto de multiplicación, por lo que no se puede hacer lo suficientemente general. Pongamos en ese caso una clase:

```
class TiposConNormalización b where
    norm :: b -> b --normalización
```

Sea ahora el tipo:

```
infix 5 :/
data Rac a = a :/ a
```

que fácilmente se declara como instancia de *Show*:

```
instance Show a => Show (Rac a) where
    showsPrec _ (x :/ y) = shows x . showChar ' ' . shows y
```

de forma que:

```
MAIN> show (3 :/ (-2))
"3/-2" :: [Char]
```

Obsérvese que en la declaración del dato no hemos derivado directamente con:

```
data Rac a = a :/ a deriving (Eq, Ord)
```

ya que el concepto de igualdad y orden no es estructural; por el contrario tendríamos la igualdad correcta:

```
instance Num a => Eq (Rac a) where
    (x :/ y) == (x' :/ y') = x * y' == x' * y
```

donde se observa la necesidad del contexto *Num a* que nos permite tener visibilidad y acceso al producto e igualdad del tipo base.

5.6. Ejercicios

La operación de normalización puede ser genérica:

```
instance Integral a => TiposConNormalización (Rac a) where
    norm (x :/ 0) = error "Racional con denominador 0"
    norm (x :/ y) = ((signum (x * y)) * ((abs x) `div` m)) :/ ((abs y) `div` m)
        where m = gcd x y
```

donde el contexto aparece debido a que el tipo de la función *gcd* (*greater common divisor*) que se utiliza en la definición es:

```
gcd :: Integral a => a -> a -> a
```

La función *signum* calcula el signo de un número. La definición de *Rac a* como instancia de *Num* con normalización se define como:

```
instance Integral a => Num (Rac a) where
    (x :/ y) + (x' :/ y') = norm ((x * y' + y * x') :/ (y * y'))
    (x :/ y) - (x' :/ y') = x :/ y + (negate x' :/ y')
    (x :/ y) * (x' :/ y') = norm ((x * x') :/ (y * y'))
    abs (x :/ y) = abs x :/ abs y
    fromInteger i = (fromInteger i) :/ 1
```

Finalmente, para definir una instancia de la clase *Ord*, sólo es necesario definir el operador (\leq) (ya que los restantes — $\{ <, >, \neq, \geq \}$, *max*, *min* y *compare* — están definidos por defecto):

```
instance (Integral a, Ord a) => Ord (Rac a) where
    u ≤ v = x * y' ≤ y * x'
    where (x :/ y) = norm u
          (x' :/ y') = norm v
```

y aquí el contexto que aparece es imprescindible para poder multiplicar, normalizar y comparar los elementos del tipo base.

5.6. EJERCICIOS

5.5 Defina una instancia de *Ord* para el tipo *data Nat = Cero | Suc Nat*.

5.6 Defina una instancia de *Eq* y una instancia de *Ord* para el tipo *ColorSimple*:

```
data ColorSimple = Violeta | Añil | Azul | Verde | Amarillo | Naranja | Rojo
```

teniendo en cuenta que dos colores simples son iguales si son idénticos y que el orden de los colores es el que viene determinado por la enumeración dada.

5.7 Defina una instancia de *Eq* para el tipo *Color*:

```
data Color = Violeta | Añil | Azul | Verde | Amarillo | Naranja | Rojo |
            Mezcla Color Color
```

teniendo en cuenta que dos colores son iguales si, o bien siendo simples son idénticos, o bien siendo compuestos tienen el mismo porcentaje de cada color. Por ejemplo:

Mezcla Rojo (Mezcla Rojo Azul) == Mezcla Azul (Mezcla Rojo Roja)
Mezcla Rojo (Mezcla Azul Roja) ≠ Mezcla Rojo (Mezcla Azul Azul)

5.8 Una mochila es como un conjunto excepto en que un elemento dado puede estar más de una vez; suponiendo que el tipo base de la mochila dispone de la igualdad y de una relación de orden:

- (a) Escriba una definición polimórfica para mochilas.
- (b) Defina una función añadir que añada un elemento a una mochila; considere la posibilidad de conservar cierta ordenación dependiendo del tipo elegido en el apartado (a).
- (c) Defina una función extraer que devuelva un elemento de cierta mochila junto con la mochila con el elemento recién extraído.
- (d) Defina un predicado esVacia que indique si cierta mochila está vacía.
- (e) Defina una función unión sobre mochilas en términos de la función añadir.
- (f) Sobrecargue el operador (==) para contemplar la igualdad entre mochilas.

5.9 ¿Se podría resolver el Ejercicio 5.7 utilizando mochilas?

6 PROGRAMACIÓN CON LISTAS

6.1. EL TIPO LISTA

La lista es uno de los tipos estructurados más útiles en programación. Una lista es una estructura muy apropiada para representar colecciones de objetos homogéneos (todos los objetos han de tener el mismo tipo). A diferencia de los conjuntos, una lista puede contener elementos repetidos y el orden de los elementos es importante. El tipo lista está predefinido en HASKELL. Se trata de un tipo recursivo. Así, una lista que almacena elementos de tipo *a* (este tipo se denota [a]) está predefinida como:

```
infixr 5 :  
data [a] = [] | a : [a] -- pseudocódigo
```

Como vemos, existen dos constructores cuyos tipos son:

```
[] :: [a]  
(:) :: a → [a] → [a]
```

El primero es utilizado para denotar *listas vacías* (listas que almacenan cero elementos) de cualquier tipo. El segundo permite añadir un nuevo elemento al principio de una lista. Obsérvese que el tipo del nuevo elemento ha de coincidir con el de los demás. En otro caso se produce un error de tipo:

```
PRELUDE> True : []  
[True] :: [Bool]  
  
PRELUDE> False : (True : [])  
[False, True] :: [Bool]  
  
PRELUDE> 'a' : (True : [])  
ERROR : Type error in application  
*** Expression : 'a' : True : []  
*** Term : 'a'  
*** Type : Char  
*** Does not match : Bool
```

El constructor (:) es asociativo a la derecha, por lo que algunos paréntesis pueden ser omitidos. Aun así, la sintaxis sigue siendo engorrosa por lo que se usa una sintaxis

para listas más cómoda: escribir los elementos entre corchetes y separados por comas.

```
PRELUDE> 1 : (2 : (3 : []))
[1, 2, 3] :: [Integer]
```

```
PRELUDE> 1 : 2 : 3 : []
[1, 2, 3] :: [Integer]
```

```
PRELUDE> [1, 2, 3]
[1, 2, 3] :: [Integer]
```

```
class Enum a where
    succ, pred      :: a → a
    toEnum          :: Int → a
    fromEnum        :: a → Int
    enumFrom        :: a → [a]
    enumFromThen   :: a → a → [a]
    enumFromTo     :: a → a → [a]
    enumFromThenTo :: a → a → a → [a]
    succ
    pred
    enumFrom x
    enumFromTo x y
    enumFromThen x y
    enumFromThenTo x y z = map toEnum [fromEnum x ..]
```

= toEnum . (1+) . fromEnum
= toEnum . subtract 1 . fromEnum
= map toEnum [fromEnum x ..]
= map toEnum [fromEnum x .. fromEnum y ..]
= map toEnum [fromEnum x .. fromEnum y .. fromEnum z ..]

Figura 6.1: La clase *Enum*.

6.1.1. SECUENCIAS ARITMÉTICAS. LA CLASE *Enum*

Una clase de PRELUDE es *Enum* (ver la Figura 6.1): la clase de los tipos *enumerables* (aquellos para los cuales tiene sentido el concepto de sucesor y antecesor). La clase contiene métodos para obtener el predecesor (*pred*) y el sucesor (*succ*) de un elemento, para obtener el orden de un valor como un entero (*fromEnum*) y para obtener un valor a partir de su orden (*toEnum*). Las instancias de esta clase pueden ser derivadas automáticamente para tipos enumerados. Por ejemplo, dada la siguiente declaración:

```
data Día = Lunes | Martes | Miércoles | Jueves
          | Viernes | Sábado | Domingo deriving (Show, Enum)
```

podemos mantener el siguiente diálogo:

```
MAIN> succ Lunes
Martes :: Día
```

6.1.2 El tipo lista

```
MAIN> pred Miércoles
Martes :: Día
```

```
MAIN> fromEnum Lunes
0 :: Int
```

```
MAIN> toEnum 4 :: Día
Viernes :: Día
```

El orden del primer constructor del tipo es 0. Hay que especificar el tipo destino al usar *toEnum* si éste no puede ser inferido del contexto, como ocurre en el ejemplo.

Para tipos no enumerados no es posible derivar automáticamente la instancia y es necesario realizar ésta explícitamente. Todos los métodos están declarados por defecto excepto *toEnum* y *fromEnum*, por lo que basta definir éstos. Por ejemplo, podemos hacer que el tipo *Nat* sea instancia de esta clase del siguiente modo:

```
data Nat = Cero | Suc Nat deriving Show
```

```
instance Enum Nat where
    toEnum 0      = Cero
    toEnum (n + 1) = Suc (toEnum n)
    fromEnum Cero = 0
    fromEnum (Suc n) = 1 + fromEnum n
```

El siguiente diálogo muestra el comportamiento de la instancia definida:

```
MAIN> succ Cero
Suc Cero :: Nat
```

```
MAIN> fromEnum (Suc Cero)
1 :: Int
```

```
MAIN> toEnum 5 :: Nat
Suc (Suc (Suc (Suc (Suc Cero)))) :: Nat
```

Observe que las definiciones anteriores pueden ser expresadas de un modo más breve utilizando las funciones de orden superior definidas en capítulos previos:

```
instance Enum Nat where
    toEnum = iter (λ _ q → Suc q) Cero
    fromEnum = foldNat (+1) 0
```

Es posible definir listas que formen una secuencia aritmética mediante una sintaxis muy compacta siempre que los elementos sean instancia de la clase *Enum*:

```
PRELUDE> [1 .. 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] :: [Integer]
```

Dos puntos consecutivos separan el elemento inicial y final de la secuencia. La distancia entre dos elementos consecutivos de la secuencia no tiene que ser uno, pero en ese caso hay que especificar dos elementos iniciales:

```
PRELUDE> [1, 3..11]
[1, 3, 5, 7, 9, 11] :: [Integer]
```

```
PRELUDE> [10, 9..1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1] :: [Integer]
```

Si no se especifica el elemento final de la secuencia, se pueden obtener listas infinitas:

```
PRELUDE> [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15{Interrupted!}]
```

```
MAIN> [Cero ..]
[Cero, Suc Cero, Suc (Suc Cero), Suc (Suc (Suc Cero)), Suc (Suc (Suc (Suc Cero)))]{Interrupted!}
```

anque la lista es finita si el tipo base lo es:

```
MAIN> [Jueves ..]
[Jueves, Viernes, Sábado, Domingo] :: [Día]
```

Las secuencias aritméticas se implementan mediante los distintos métodos según la siguiente tabla:

Semántica de las secuencias aritméticas

$[e_1 ..]$	$\Rightarrow enumFrom e_1$
$[e_1 .. e_2]$	$\Rightarrow enumFromTo e_1 e_2$
$[e_1, e_2 ..]$	$\Rightarrow enumFromThen e_1 e_2$
$[e_1, e_2 .. e_3]$	$\Rightarrow enumFromThenTo e_1 e_2 e_3$

6.2. CONCATENACIÓN DE LISTAS

El operador de *concatenación* de listas (`++`) permite obtener una nueva lista disponiendo los elementos de la segunda tras los de la primera:

```
MAIN> [1, 2, 3] ++ [2, 7]
[1, 2, 3, 2, 7] :: [Integer]
```

Podemos definir fácilmente este operador examinando la forma del primer argumento:

```
infixr 5 ++
(+)      :: [a] → [a] → [a]
[] + ys = ys
(x : xs) + ys = x : (xs ++ ys)
```

6.2 - Concatenación de listas

Los dos argumentos del operador han de ser listas del mismo tipo.
Una reducción con la definición anterior es:

```
[1, 2, 3] ++ [2, 7]
⇒ { sintaxis de listas }
(1 : (2 : (3 : []))) ++ [2, 7]
⇒ { segunda ecuación de (+) }
1 : ((2 : (3 : [])) ++ [2, 7])
⇒ { segunda ecuación de (+) }
1 : (2 : ((3 : []) ++ [2, 7]))
⇒ { segunda ecuación de (+) }
1 : (2 : (3 : ([] ++ [2, 7])))
⇒ { primera ecuación de (+) }
1 : (2 : (3 : [2, 7]))
⇒ { sintaxis de listas }
[1, 2, 3, 2, 7]
```

Ejercicio 6.1 Defina el operador (`++`) a partir de patrones en el segundo argumento:

$$\begin{array}{lcl} xs ++ [] & = & xs \\ xs ++ (y : ys) & = & \dots \end{array}$$

Ejercicio 6.2 Para concatenar un elemento al inicio de una lista puede usarse el constructor (`:`). Defina un operador (`>+>`) que permita concatenar un elemento al final de una lista. Por ejemplo:

```
MAIN> [1, 2, 3] >+> 7
[1, 2, 3, 7] :: [Integer]
```

Algunas propiedades del operador (`++`) son:

Elemento neutro a la derecha:

$$\forall xs :: [a] . xs ++ [] = xs$$

Elemento neutro a la izquierda:

$$\forall ys :: [a] . [] ++ ys = ys$$

Asociatividad:

$$\forall xs, ys, zs :: [a] . (xs ++ ys) ++ zs = xs ++ (ys ++ zs)$$

Al ser el operador asociativo es indiferente que se defina como asociativo a la izquierda o a la derecha, pero al realizarse la recursión sobre el primer argumento resulta más eficiente declarar el operador asociativo a la derecha.

Ejercicio 6.3 Compruebe que la evaluación de $xs \uparrow\uparrow ys$ realiza $n + 1$ reducciones, donde n es el número de elementos de la lista xs .

Ejercicio 6.4 Sean n_{xs} , n_{ys} y n_{zs} las longitudes de las listas xs , ys y zs respectivamente. ¿Cuántas reducciones son necesarias para evaluar la expresión $(xs \uparrow\uparrow ys) \uparrow\uparrow zs$? ¿Cuál de las dos expresiones se calcula más eficientemente?

También existe la función `concat` que toma una lista de listas y devuelve como resultado la lista concatenación de todas ellas:

```
PRELUDE> concat [[1,2], [3,4,5], [6]]
[1,2,3,4,5,6] :: [Integer]
```

Es fácil definir esta función de modo recursivo sobre la lista argumento:

```
concat      :: [[a]] → [a]
concat []    = []
concat (xs : xss) = xs ++ concat xss
```

Ejercicio 6.5 Evalúe perezosamente la expresión `concat [[1,2], [3,4,5], [6]]`.

Ejercicio 6.6 Defina la función que invierte el orden de los elementos de una lista:

```
PRELUDE> reverse [1,2,3]
[3,2,1] :: [Integer]
```

6.3. INDUCCIÓN SOBRE LISTAS

Las listas son estructuras inductivas, por lo que hay un principio de inducción para ellas:

$\text{Principio de inducción para listas finitas}$ $\forall xs :: [a] . P(xs) \Leftrightarrow \left\{ \begin{array}{l} P([]) \\ \wedge \\ \forall xs :: [a], \forall r :: a . (P(xs) \Rightarrow P(r : xs)) \end{array} \right.$

La lectura del paso inductivo es que deberemos probar que $P(x : xs)$ es cierto sea cual sea x y en esta demostración podremos asumir que $P(xs)$ es cierto. Este principio sólo es válido para listas finitas. Veremos posteriormente cómo ampliarlo para listas infinitas.

Ejemplo 6.7 Usando el principio anterior, demostraremos la siguiente propiedad:

$(\uparrow\uparrow)$ distribuye con `length` mediante $(+)$:

6.4 - Selectores

$$\forall xs, ys :: [a] . length (xs \uparrow\uparrow ys) = length xs + length ys$$

donde la función `length` calcula la longitud de una lista:

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

Realizaremos la inducción sobre la variable xs , por lo que la propiedad a probar es:

$$P(xs) = \forall ys :: [a] . length (xs \uparrow\uparrow ys) = length xs + length ys$$

- Caso Base: $P([])$; o sea,

$$\forall ys :: [a] . length ([] \uparrow\uparrow ys) = length [] + length ys$$

- Paso Inductivo: $\forall xs :: [a], \forall x :: a . P(xs) \Rightarrow P(x : xs)$; o sea,

$$\begin{aligned} \forall xs :: [a], \forall x :: a, \\ \forall ys :: [a] . length ((x : xs) \uparrow\uparrow ys) &= length (x : xs) + length ys \\ \Rightarrow \forall ys :: [a] . length ((x : xs) \uparrow\uparrow ys) &= length (x : xs) + length ys \end{aligned}$$

Caso Base:

$$\begin{aligned} \text{length} ([] \uparrow\uparrow ys) &\stackrel{\text{definición de } (\uparrow\uparrow)}{=} \\ &\stackrel{\text{length } ys}{=} \\ &\stackrel{\text{aritmética}}{=} \\ &0 + \text{length } ys \\ &\stackrel{\text{definición de } \text{length}}{=} \\ &\text{length} [] + \text{length } ys \end{aligned}$$

Paso Inductivo:

$$\begin{aligned} \text{length} ((x : xs) \uparrow\uparrow ys) &\stackrel{\text{definición de } (\uparrow\uparrow)}{=} \\ &\stackrel{\text{length } (x : (xs \uparrow\uparrow ys))}{=} \\ &\stackrel{\text{definición de } \text{length}}{=} \\ &1 + \text{length} (xs \uparrow\uparrow ys) \\ &\stackrel{\text{hipótesis de inducción}}{=} \\ &1 + (\text{length } xs + \text{length } ys) \\ &\stackrel{\text{asociatividad de } + \text{ para enteros}}{=} \\ &(1 + \text{length } xs) + \text{length } ys \\ &\stackrel{\text{definición de } \text{length}}{=} \\ &\text{length} (x : xs) + \text{length } ys \end{aligned}$$

con lo que queda demostrada cualquier propiedad para cualquier lista finita.

Ejercicio 6.8 Demuestre las tres propiedades de la concatenación de listas descritas en la Sección 6.2.

6.4. SELECTORES

Las funciones `selectoras` permiten extraer parte de una lista. Las dos más simples son `head` y `tail` que permiten extraer la cabeza y el resto de una lista no vacía:

```
head      :: [a] → a
head (x : xs) = x
tail      :: [a] → [a]
tail (_ : xs) = xs
```

La función *last* extrae el último elemento de una lista mientras que *init* devuelve la lista que se obtiene al suprimir el último elemento (véase la Figura 6.2). Las cuatro funciones citadas están parcialmente definidas (sólo están definidas para listas no vacías). El siguiente diálogo muestra su comportamiento:

```
PRELUDE> head [1..5]
1 :: Integer
PRELUDE> tail [1..5]
[2, 3, 4, 5] :: [Integer]
PRELUDE> last [1..5]
5 :: Integer
PRELUDE> init [1..5]
[1, 2, 3, 4] :: [Integer]
```

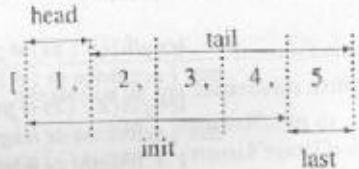


Figura 6.2: *head*, *tail*, *init* y *last*.

Ejercicio 6.9 Defina las funciones *last* e *init* recursivamente.

Ejercicio 6.10 Demuestre la siguiente propiedad para listas no vacías:

Relación entre *head* y *tail*:

$$\forall ls :: [a] . ls \neq [] . head ls : tail ls = ls$$

También es posible seleccionar una cantidad de elementos iniciales de una lista (véase la Figura 6.3) mediante la función *take*:

```
PRELUDE> take 3 [1..5]
[1, 2, 3] :: [Integer]
```

6.4 - Selectores

```
PRELUDE> take 10 [1..5]
[1, 2, 3, 4, 5] :: [Integer]
```

Obsérvese que si el número de elementos requerido es mayor que el tamaño de la lista no se produce un error, sino que se devuelven todos los posibles. Esto se debe a la segunda línea de la definición de esta función:

```
take      :: Int → [a] → [a]
take 0    = []
take _ [] = []
take n (x : xs) | n > 0 = x : take (n - 1) xs
take _ _ = error "Prelude.take: negative argument"
```

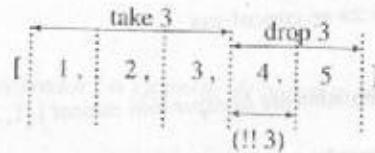


Figura 6.3: *take*, *drop* y *(!!)*.

La función *drop* permite eliminar un número de elementos del principio de una lista:

```
PRELUDE> drop 3 [1..5]
[4, 5] :: [Integer]
```

```
PRELUDE> drop 10 [1..5]
[] :: [Integer]
```

Ejercicio 6.11 Defina la función *drop*.

La función *splitAt* combina los resultados de *take* y *drop* en una tupla:

```
PRELUDE> splitAt 3 [1..5]
([1, 2, 3], [4, 5]) :: ([Integer], [Integer])
```

```
PRELUDE> splitAt 10 [1..5]
([1, 2, 3, 4, 5], []) :: ([Integer], [Integer])
```

mientras que el operador *!!* permite seleccionar un elemento de una lista a partir de su posición. Se considera que el elemento de la cabeza ocupa la posición cero:

```
PRELUDE> [1..5] !! 3
4 :: Integer
```

PRELUDE> [1..5] !! 10

Program error : Prelude.!! : index too large

Como muestra el ejemplo, se produce un error si el elemento que ocupa la posición solicitada no existe. La definición de este operador es:

```
infixl 9 !!
(!) :: [a] → Int → a
(x : _) !! 0 = x
(_ : xs) !! n | n > 0 = xs !! (n - 1)
(_ : _) !! _ = error "Prelude.!!: negative index"
[] !! _ = error "Prelude.!!: index too large"
```

6.5. EMPAREJANDO LISTAS

La función predefinida *zipWith*:

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f (a : as) (b : bs) = f a b : zipWith f as bs
zipWith _ _ = []
```

aplica cierta función a los elementos de dos listas tomándolos de dos en dos:

```
PRELUDE> zipWith (+) [1..3] [10..]
[11, 13, 15] :: [Integer]
```

Obsérvese que la longitud de la lista resultado coincide con la de menor longitud.

La función *zip*:

```
zip :: [a] → [b] → [(a, b)]
zip = zipWith (λ a b → (a, b))
```

es un caso particular de *zipWith* y construye una lista de pares a partir de dos listas:

```
PRELUDE> zip [1..3] ['a'.. 'c']
[(1, 'a'), (2, 'b'), (3, 'c')] :: [(Integer, Char)]
```

La función *unzip* realiza el proceso inverso:

```
unzip :: [(a, b)] → ([a], [b])
unzip [] = ([], [])
unzip ((a, b) : xs) = (a : as, b : bs)
where
  (as, bs) = unzip xs
```

6.6. APPLICANDO UNA FUNCIÓN A LOS ELEMENTOS DE UNA LISTA

Una de las operaciones más frecuentes con listas es transformar todos sus elementos aplicándoles una misma función. Por ejemplo, la función que eleva al cuadrado todos los elementos de una lista es:

```
listaAlCuadrado :: [Integer] → [Integer]
listaAlCuadrado [] = []
listaAlCuadrado (x : xs) = alCuadrado x : listaAlCuadrado xs
where
  alCuadrado = (↑ 2)
```

```
MAIN> listaAlCuadrado [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :: [Integer]
```

Otro ejemplo es la función que pasa a mayúsculas una cadena de caracteres:

```
listaAMayúsculas :: [Char] → [Char]
listaAMayúsculas [] = []
listaAMayúsculas (x : xs) = toUpper x : listaAMayúsculas xs
```

```
MAIN> listaAMayúsculas "hola"
"HOLA" :: [Char]
```

Podemos observar que la definición de ambas funciones es muy similar (sólo cambia la función a aplicar a cada elemento de la lista), por lo que podemos definir una función de orden superior que tome esta función como argumento. Este combinador está definido en PRELUDE como *map*:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Así, los ejemplos anteriores pueden ser expresados como:

```
PRELUDE> map (↑ 2) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :: [Integer]
```

```
PRELUDE> map toUpper "hola"
"HOLA" :: [Char]
```

Obsérvese que el tipo de la lista resultado no tiene que coincidir con el de la lista argumento, aunque el dominio de la función debe coincidir con el tipo base de la lista argumento, y el tipo base de la lista resultado coincidirá con el rango de ésta:

```
PRELUDE> map ord "hola"
[104, 111, 108, 97] :: [Int]
```

Ejercicio 6.12 Demuestre las siguientes propiedades:

map distribuye con (.):

$$\forall xs :: [a], g :: a \rightarrow b, f :: b \rightarrow c . \text{map}(f.g) xs = (\text{map } f . \text{map } g) xs$$

Relación entre map e id:

$$\forall xs :: [a] . \text{map id} xs = xs$$

map no modifica la longitud de una lista:

$$\forall xs :: [a], f :: a \rightarrow b . \text{length}(\text{map } f xs) = \text{length } xs$$

6.7. FILTROS

Un filtro permite seleccionar los elementos de una lista que cumplen cierta propiedad. La función filter cumple este cometido. Se trata de una función de orden superior que toma como argumentos un *predicado* (de tipo $a \rightarrow \text{Bool}$) y una lista, y devuelve la lista formada por aquellos elementos que verifican el predicado:

PRELUDE> filter even [1..10]
[2, 4, 6, 8, 10] :: [Integer]

PRELUDE> filter (> 'g') "me gustan las listas"
"mustn't lists" :: [Char]

Como vemos, la función es polimórfica ya que puede actuar sobre listas con cualquier tipo base. La definición de la función es:

filter	$\vdash (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$
filter $p []$	= []
filter $p (x : xs)$	$ p x = x : \text{filter } p xs$ $ \text{otherwise} = \text{filter } p xs$

El tipo del argumento del predicado ha de coincidir con el de los elementos de la lista.

Ejercicio 6.13 Demuestre la siguiente propiedad:

filter no aumenta la longitud de una lista:

$$\forall xs :: [a], p :: a \rightarrow \text{Bool} . \text{length}(\text{filter } p xs) \leq \text{length } xs$$

La función filter recorre por completo (y en forma perezosa) la lista a filtrar. Otra filtro similar es takeWhile que, con los mismos argumentos que filter, devuelve el mayor segmento inicial de elementos de la lista que verifican el predicado. En cuanto uno

de los elementos no verifique el predicado, no se comprueba la condición para los que le siguen. El siguiente diálogo muestra la diferencia entre filter y takeWhile:

PRELUDE> filter even [2, 4, 8, 9, 10, 11, 12]
[2, 4, 8, 10, 12] :: [Integer]

PRELUDE> takeWhile even [2, 4, 8, 9, 10, 11, 12]
[2, 4, 8] :: [Integer]

filter devuelve todos los elementos pares de la lista original, pero takeWhile para en cuanto encuentra uno impar.

La función takeWhile es muy útil al trabajar con listas infinitas. El siguiente ejemplo muestra cómo se pueden calcular los cuadrados menores que 100 de los números naturales:

PRELUDE> takeWhile (< 100) (map (↑ 2) [0..])
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81] :: [Integer]

La expresión $\text{map} (\uparrow 2) [0..]$ representa la lista infinita de los cuadrados de los números naturales. Por ser esta lista monótona creciente, la función takeWhile (< 100) extrae tan sólo los elementos que nos interesan. Obsérvese que se obtiene como resultado final una lista finita aunque una de las expresiones intermedias sea infinita. Esto es posible gracias al mecanismo de evaluación perezosa, que genera tan sólo la parte inicial de la lista infinita necesaria para calcular el resultado.

Ejercicio 6.14 La definición de la función takeWhile es

takeWhile	$\vdash (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$
takeWhile $p []$	= []
takeWhile $p (x : xs)$	$ p x = x : \text{takeWhile } p xs$ $ \text{otherwise} = []$

Evalue perezosamente la expresión takeWhile (< 10) ($\text{map} (\uparrow 2) (\text{desde } 0)$), donde

desde :: Integer \rightarrow [Integer]
desde $n = n : \text{desde } (n + 1)$

Ejercicio 6.15 La función filter examina por completo la lista a filtrar, por lo que el cálculo no termina si la lista argumento es infinita aunque la evaluación sea perezosa. Compruébelo evaluando filter (< 10) ($\text{map} (\uparrow 2) (\text{desde } 0)$).

El último filtro predefinido es dropWhile, que elimina de una lista el mayor segmento inicial de elementos que verifican un predicado. El primer elemento que no verifica el predicado y todos los que le siguen pasan al resultado:

PRELUDE> dropWhile even [2, 4, 8, 9, 10, 11, 12]
[9, 10, 11, 12] :: [Integer]

Ejercicio 6.16 Defina la función $\text{dropWhile} :: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$.

6.8. LISTAS POR COMPRENSIÓN

HASKELL permite definir listas mediante una notación similar a la utilizada para describir *conjuntos por comprensión* en matemáticas. Por ejemplo,

$$\{x \mid x \in N, x \text{ es par}\}$$

denota el conjunto de los x tales que x es un número natural par. En HASKELL podemos usar la notación de listas por comprensión para obtener el mismo resultado:

```
PRELUDE> [x | x ← [0..], even x]
[2, 4, 6, 8, 10, 12, 14, 16]{Interrupted!}
```

La sintaxis de una lista por comprensión en HASKELL es:

$$[\text{expresión} \mid \text{cualificador}_1, \text{cualificador}_2, \dots, \text{cualificador}_n]$$

donde cada cualificador puede ser:

- Un *generador*: expresión que genera una lista.
- Una *guarda*: expresión booleana.
- Una *definición local*: se usan para definir elementos locales a la expresión.

En el ejemplo:

```
PRELUDE> [2 * x | x ← [1..5]]
[2, 4, 6, 8, 10] :: [Integer]
```

el único cualificador es un generador. La variable x se instancia a cada uno de los elementos de la lista $[1..5]$ y, para cada instancia, un elemento de la forma $2 * x$ se añade a la lista resultado. El ejemplo muestra que la expresión a la izquierda del símbolo $|$ puede depender de las variables introducidas en los generadores. La expresión del ejemplo es equivalente a $\text{map} (2*) [1..5]$. De hecho, es posible definir la función map de forma compacta utilizando la sintaxis de listas por comprensión:

$$\begin{aligned} \text{map} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \text{map } f \text{ xs} &= [f x \mid x \leftarrow \text{xs}] \end{aligned}$$

Una guarda puede ser utilizada para seleccionar los elementos de un generador que cumplen cierta condición:

```
PRELUDE> [2 * x | x ← [1..5], even x]
[4, 8] :: [Integer]
```

Sólo los elementos pares de la lista $[1..5]$ generan un elemento en la lista resultado. El ejemplo muestra que un cualificador puede utilizar una variable introducida en uno previo (más a la izquierda). El ejemplo equivale a la expresión $\text{map} (2*) (\text{filter even} [1..5])$.

6.8 - Listas por comprensión

También es posible definir la función filter utilizando la sintaxis de listas por comprensión:

$$\begin{aligned} \text{filter} &:: (\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{filter } p \text{ xs} &= [x \mid x \leftarrow \text{xs}, p x] \end{aligned}$$

Si se introduce más de un generador, los que aparecen a la derecha cambian más rápido:

```
PRELUDE> [(x, y) | x ← [1..3], y ← ['a', 'b']]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')] :: [(Integer, Char)]
```

```
PRELUDE> concat (map (\x → map (\y → (x, y)) ['a', 'b']) [1..3])
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')] :: [(Integer, Char)]
```

La segunda expresión es un modo equivalente de escribir la expresión ejemplo. Como vemos, es posible escribir cualquier expresión sin utilizar la sintaxis de listas por comprensión¹. La principal ventaja de esta sintaxis es que muchas expresiones resultan más claras. Por ejemplo, la siguiente función calcula los divisores de un número natural:

$$\begin{aligned} \text{divideA} &:: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Bool} \\ d \cdot \text{divideA} \cdot n &= n \cdot \text{mod} \cdot d == 0 \end{aligned}$$

$$\begin{aligned} \text{divisoresDe} &:: \text{Integer} \rightarrow [\text{Integer}] \\ \text{divisoresDe } n &= [x \mid x \leftarrow [1..n], x \cdot \text{divideA} \cdot n] \end{aligned}$$

Ahora es fácil calcular el máximo común divisor de dos números:

$$\begin{aligned} \text{mcd} &:: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\ \text{mcd } a \text{ } b &= \text{maximum} [x \mid x \leftarrow \text{divisoresDe } a, x \cdot \text{divideA} \cdot b] \end{aligned}$$

donde la función predefinida maximum calcula el mayor elemento de una lista.

Un número natural es primo si sólo tiene dos divisores distintos: 1 y él mismo,

$$\begin{aligned} \text{esPrimo} &:: \text{Integer} \rightarrow \text{Bool} \\ \text{esPrimo } n &= \text{divisoresDe } n == [1, n] \end{aligned}$$

La siguiente función proporciona la lista de todos los números primos:

$$\begin{aligned} \text{losPrimos} &:: [\text{Integer}] \\ \text{losPrimos} &= [n \mid n \leftarrow [2..], \text{esPrimo } n] \end{aligned}$$

y podemos tomar los diez primeros:

```
MAIN> take 10 losPrimos
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29] :: [Integer]
```

o calcular cuántos primos hay menores que 100:

¹Es posible traducir una lista por comprensión en otra lista equivalente vía las funciones map y concat .

MAIN> `length (takeWhile (< 100) losPrimos)`
`25 :: Int`

o incluso calcular el primer primo mayor que mil:

MAIN> `head (dropWhile (≤ 1000) losPrimos)`
`1009 :: Integer`

Ejercicio 6.17 Escriba una expresión que muestre los primos menores que mil que acaban en tres.

Ejercicio 6.18 Defina la función `primeroQue :: (a → Bool) → [a] → a` que devuelve el primer elemento de la lista (posiblemente infinita) segundo argumento que verifique el predicado primer argumento:

MAIN> `primeroQue (> 1000) losPrimos`
`1009 :: Integer`

Ejercicio 6.19 Un número es perfecto si coincide con la suma de sus divisores propios. Por ejemplo, 6 es perfecto: $1 + 2 + 3 = 6$. Escriba una función que devuelva la lista de los números perfectos.

Otro cualificador es una definición local que se introduce con la palabra `let`. El ámbito de la definición introducida abarca todos los cualificadores a la derecha de `let` y la expresión a la izquierda de `|`. Como ejemplo, la siguiente expresión calcula los cuadrados pares de los naturales menores que 11:

PRELUDE> `[cuadrado | n ← [0..10], let cuadrado = n * n, even cuadrado]`
`[0, 4, 16, 36, 64, 100] :: [Integer]`

Ejemplo 6.20 Las ternas pitagóricas son aquellas ternas de números enteros que pueden ser lados de un triángulo rectángulo:

`ternasPitHasta :: Integer → [(Integer, Integer, Integer)]`
`ternasPitHasta n = [(x, y, z) | let ns = [1..n],`
`x ← ns, y ← ns, z ← ns,`
`x ↑ 2 + y ↑ 2 == z ↑ 2]`

Ejemplo 6.21 La siguiente función comprueba si una operación binaria f es commutativa sobre un conjunto representado por la lista ds :

`commuta :: (a → a → b) → [a] → Bool`
`commuta f ds = and [a `f` b == b `f` a | a ← ds, b ← ds]`

donde la función predefinida `and` comprueba si todos los elementos de una lista de booleanos son ciertos. Por ejemplo:

MAIN> `commuta (+) [1..10]`
`True :: Bool`

MAIN> `commuta (&&) [True, False]`
`True :: Bool`

Ejemplo

6.8.1. SEMÁNTICA DE LISTAS POR COMPRENSIÓN

Como hemos comentado previamente, es posible traducir cualquier expresión que use la sintaxis de listas por comprensión en otra equivalente que use las funciones `map` y `concat`. Las siguientes reglas indican cómo hacerlo:

Semántica de listas por comprensión	
(1)	$[e]$ $\Rightarrow [e]$
(2)	$[e b, Q] \Rightarrow \text{if } b \text{ then } [e Q] \text{ else } []$
(3)	$[e p ← l, Q] \Rightarrow \begin{array}{l} \text{let } \\ \quad ok\ p = [e Q] \\ \quad ok_ = [] \\ \text{in } \end{array}$
(4)	$[e \text{let } decls, Q] \Rightarrow \begin{array}{l} \text{let } \\ \quad decls \\ \text{in } \end{array} [e Q]$

El símbolo \Rightarrow indica que una expresión cuya forma coincide con la que aparece a su izquierda es traducida a la expresión de la derecha. En este esquema de traducción, e representa una expresión, p un patrón, l una lista, b una expresión booleana, $decls$ una lista de definiciones locales, Q cero o más cualificadores separados por coma, y ok es un nombre de función *fresca* (una función cuyo nombre no coincide con otra que aparezca en el resto de la expresión). Esta función es utilizada para eliminar los elementos de la lista que no encajan con el patrón p .

Por ejemplo, la traducción de la expresión $[2 * x | x ← [1..3], even x]$ es:

$[2 * x | x ← [1..3], even x] \Rightarrow \{\text{regla (3)}\}$
 $\Rightarrow \{\text{regla (2)}\}$
 $\Rightarrow \{\text{regla (1)}\}$

2. Escriba una red de procesos (y sus ecuaciones) para computar la lista:

$$\text{Inombra } xs \equiv [xs, \text{nombra } xs, \text{nombra}(\text{nombra } xs), \dots]$$

3. Modifique la red anterior y escriba ecuaciones para obtener la secuencia de elementos u (de la lista Inombra) que verifican:

$$\text{length } u = \text{length}(\text{nombra } u)$$

4. Escriba un programa para comprobar las siguientes conjeturas:

(a) Las longitudes de los elementos de la lista:

$$\text{Inombra}' n \equiv [[n], \text{nombra } [n], \text{nombra}(\text{nombra } [n]), \dots]$$

son independientes del valor entero n .

(b) No existe ninguna lista xs perteneciente a Inombra' u tal que verifique:

$$\text{length } xs > \text{length}(\text{nombra } xs)$$

(c) Si el valor inicial es positivo, todas las listas terminan en el mismo elemento.

(d) Si el valor inicial es negativo, todas las listas terminan en el mismo elemento.

(e) Si $u \neq v$, entonces ninguna lista de Inombra' u contiene a v .

8.22 Siendo $\text{fix } f\ x = f\ x\ (\text{fix } f)$, calcule en forma perezosa algunos términos de la lista:

$$\text{fix } f\ a \text{ where } f\ x\ q = \text{map } (+x)\ (x : q\ (x + 1))$$

8.23 Defina, a partir de iterate, funciones para calcular:

1. Una lista con los múltiplos de 5.

2. Una lista con las potencias de 2.

3. Una lista con valores lógicos alternativos empezando por True.

4. La lista $[***, ****, ***^3, \dots]$

8.24 Basándonos en que el producto de las columnas siguientes produce los factoriales de los naturales, escriba un programa que los calcule.

1	2	3	4	5	6	...
1	2	3	4	5	6	...
1	2	3	4	5	6	...

.

8.25 Escriba una expresión que calcule la lista:

$$[[1, 4, 9, \dots], [1, 8, 27, \dots], [1, 16, 81, \dots], \dots]$$

8.26 Escriba una expresión que calcule la lista:

$$[[[(1, 1)], [(2, 1), (1, 2)], [(3, 1), (2, 2), (1, 3)], \dots]]$$

9

PROGRAMACIÓN CON ÁRBOLES Y GRAFOS

9.1. ÁRBOLES

Un árbol es una estructura no lineal acíclica utilizada para organizar información de forma eficiente. La definición de un árbol es recursiva. Un árbol es una colección de valores $\{v_1, v_2, \dots, v_n\}$ tales que:

- Si $n = 0$ el árbol se dice vacío.
- En otro caso, existe un valor destacado que se denomina raíz (p.e., v_1), y los demás elementos forman parte de colecciones disjuntas que a su vez son árboles.

Gráficamente, los árboles suelen dibujarse con la raíz en la parte superior, tal como muestra la Figura 9.1.

La raíz del árbol representado es el valor 10. Los valores 22, 15 y 12 forman el primer subárbol, 35 el segundo, mientras que 52 y 33 forman el último. En HASKELL, el siguiente tipo puede ser utilizado para representar árboles que almacenan datos de tipo a :

$$\text{data Árbol } a = \text{Vacío} \mid \text{Nodo } a \text{ [Árbol } a\text{]} \text{ deriving Show}$$

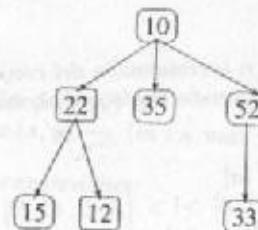


Figura 9.1: Un árbol con 7 elementos.

donde el primer argumento del constructor *Nodo* es el dato en la raíz del árbol y el segundo la lista de los subárboles del nodo raíz. El constructor *Vacio* puede ser utilizado para representar un árbol vacío. Así, el árbol de la Figura 9.1 es:

```
a1 :: Árbol Integer
a1 = Nodo 10 [a11, a12, a13]
where
  a11 = Nodo 22 [Nodo 15 [], Nodo 12 []]
  a12 = Nodo 35 []
  a13 = Nodo 52 [Nodo 33 []]
```

Una función básica sobre árboles que devuelve la raíz es:

```
raíz      :: Árbol a → a
raíz Vacío = error "raíz de árbol vacío"
raíz (Nodo x _) = x
```

El tamaño de un árbol es el número de elementos que almacena y puede ser computado del siguiente modo:

```
tamaño      :: Árbol a → Integer
tamaño Vacío = 0
tamaño (Nodo _ xs) = 1 + sum (map tamaño xs)
```

Los distintos elementos de un árbol están agrupados por niveles de modo que se considera que la raíz del árbol se encuentra a nivel cero, las raíces de los subárboles del nodo raíz están en nivel 1 y así sucesivamente. Se define la profundidad de un árbol como el máximo nivel del árbol más uno:

```
profundidad    :: Árbol a → Integer
profundidad Vacío = 0
profundidad (Nodo _ []) = 1
profundidad (Nodo _ xs) = 1 + maximum (map profundidad xs)
```

Los nodos sin subárboles, como el nodo 15 en el ejemplo, se llaman nodos hoja y se caracterizan porque su lista de subárboles es vacía:

```
esHoja      :: Árbol a → Bool
esHoja (Nodo _ []) = True
esHoja _     = False
```

Ejercicio 9.1 Defina una función *sumÁrbol* que calcule la suma de los valores almacenados en un árbol de números.

Ejercicio 9.2 Defina una función *maxÁrbol* que calcule el máximo valor almacenado en un árbol.

9.1.1. FUNCIONES DE ORDEN SUPERIOR SOBRE ÁRBOLES

La función *map* sólo está predefinida para listas, pero existe una versión sobrecargada, denominada *fmap*, predefinida en la siguiente clase:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

de modo que *f* puede ser instanciada con constructores de tipos polimórficos de aridad uno. Por ejemplo, las listas son una instancia predefinida de esta clase:

```
instance Functor [] where
  fmap = map
```

por lo que es posible usar tanto *map* como *fmap* con listas. La función *fmap* también tiene sentido para árboles y puede ser definida mediante la siguiente instancia:

```
instance Functor Árbol where
  fmap f Vacío      = Vacío
  fmap f (Nodo x xs) = Nodo (f x) (map (fmap f) xs)
```

Como puede observarse, se aplica la función al dato en la raíz y mediante *map* a todos los subárboles. La instancia anterior se puede expresar también con una lista por comprensión:

```
instance Functor Árbol where
  fmap f Vacío      = Vacío
  fmap f (Nodo x xs) = Nodo (f x) [fmap f z | x ← xs]
```

Utilizando esta función podemos, por ejemplo, definir una función que duplique los datos almacenados en un árbol:

```
duplicar :: Num a → Árbol a → Árbol a
duplicar = fmap (*2)
```

La función de plegado para árboles puede ser definida como:

```
foldÁrbol   :: (a → [b] → b) → b → (Árbol a → b)
foldÁrbol f e = fun
  where
    fun Vacío      = e
    fun (Nodo x xs) = f x (map fun xs)
```

O también como:

```

foldÁrbol      :: (a → [b] → b) → b → Árbol a → b
foldÁrbol f e Vacío = e
foldÁrbol f e (Nodo x xs) = f x (map (foldÁrbol f e) xs)

```

de modo que e sustituye al constructor *Vacio* y f sustituye a *Nodo*. Por ejemplo, la definición de *tamaño* usando el plegado es:

```

tamaño' :: Árbol a → Integer
tamaño' = foldÁrbol (λ r ts → 1 + sum ts) 0

```

donde 0 es el resultado del problema para árboles vacíos y la λ -expresión es la función que a partir del dato en la raíz y una lista con los tamaños de los subárboles calcula el tamaño total del árbol. Otro ejemplo es la siguiente función que suma los elementos de un árbol:

```

sumÁrbol' :: Num a ⇒ Árbol a → a
sumÁrbol' = foldÁrbol (λ r ss → r + sum ss) 0

```

Obsérvese que los argumentos recibidos en este caso por la λ -expresión son la raíz y una lista con las sumas correspondientes a cada uno de los subárboles. Otro ejemplo de función de orden superior es la siguiente, que comprueba si todos los elementos de un árbol cumplen cierta condición:

```

todosÁrbol      :: (a → Bool) → Árbol a → Bool
todosÁrbol p Vacío = True
todosÁrbol p (Nodo x xs) = p x && and (map (todosÁrbol p) xs)

```

Por ejemplo:

```

MAIN> todosÁrbol (> 0) a1
True :: Bool

```

Esta función también puede ser definida como un plegado:

```

todosÁrbol' :: (a → Bool) → Árbol a → Bool
todosÁrbol' p = foldÁrbol (λ r bs → p r && and bs) True

```

donde *True* es el resultado para el árbol vacío y la λ -expresión toma la raíz y una lista con los resultados de comprobar la condición en los subárboles.

Ejercicio 9.3 Defina una función:

```

algunaÁrbol :: (a → Bool) → Árbol a → Bool

```

que compruebe si algún elemento de un árbol cumple una condición. Dé dos versiones, una recursiva y otra usando la función de plegado.

Ejercicio 9.4 Defina las funciones profundidad y *fmap* a partir de la función de plegado.

Ejercicio 9.5 Defina una función:

```

ocurrencias :: Eq a ⇒ a → Árbol a → Integer

```

que calcule el número de veces que aparece un dato en un árbol. Dé dos versiones, una recursiva y otra usando la función de plegado.

9.2. ÁRBOLES BINARIOS

Un *árbol binario* es un árbol tal que cada nodo tiene como máximo dos subárboles. En HASKELL podemos usar el siguiente tipo para representar estos árboles:

```

data ÁrbolB a = VacíoB | NodoB (ÁrbolB a) a (ÁrbolB a) deriving Show

```

Consideraremos que las tres componentes del constructor *NodoB* son el subárbol izquierdo, el dato raíz y el subárbol derecho respectivamente. Los árboles binarios suelen ser utilizados como *contenedores* de datos, por lo que podemos definir una función que compruebe si un dato pertenece a un árbol binario:

```

perteneceB      :: Eq a ⇒ a → ÁrbolB a → Bool
perteneceB x VacíoB = False
perteneceB x (NodoB i r d)
| x == r        = True
| otherwise     = perteneceB x i || perteneceB x d

```

El problema fundamental para definir *perteneceB* es que tenemos que hacer una búsqueda exhaustiva, ya que si el elemento a buscar no coincide con la raíz tenemos que contemplar la posibilidad de que esté en cualquiera de los dos subárboles.

9.2.1. ÁRBOLES BINARIOS DE BÚSQUEDA

El algoritmo de búsqueda exhaustiva es el único posible si no tenemos más información sobre la disposición de los elementos. Para mejorar la eficiencia de la búsqueda podemos utilizar *árboles binarios ordenados* (o de búsqueda) siempre que los elementos a almacenar en el árbol dispongan de una relación de orden. Un árbol binario de búsqueda es un árbol binario tal que:

- o bien es vacío;
- o no es vacío y para cualquier nodo se cumple que:
 - los elementos del correspondiente subárbol izquierdo son menores o iguales al almacenado en el nodo;
 - y los elementos del correspondiente subárbol derecho son estrictamente mayores al almacenado en el nodo.

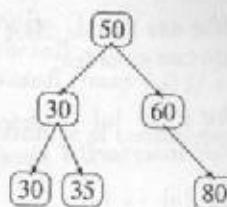


Figura 9.2: Un árbol ordenado.

El árbol de la Figura 9.2 está ordenado y puede ser representado con la siguiente expresión:

```

a2 :: ÁrbolB Integer
a2 = NodoB (NodoB (hojaB 30)
            30
            (hojaB 35))
      50
      (NodoB VacíoB
       60
       (hojaB 80))
  
```

where

$hojaB\ x = \text{NodoB VacíoB}\ x\ VacíoB$

La siguiente función puede ser utilizada para comprobar si un árbol binario es de búsqueda:

```

esÁrbolBB          :: Ord a => ÁrbolB a -> Bool
esÁrbolBB VacíoB   = True
esÁrbolBB (NodoB i r d) = todosÁrbolB ( $\leq r$ ) i &&
                           todosÁrbolB ( $> r$ ) d &&
                           esÁrbolBB i &&
                           esÁrbolBB d

todosÁrbolB        :: (a -> Bool) -> ÁrbolB a -> Bool
todosÁrbolB p VacíoB = True
todosÁrbolB p (NodoB i r d) = p r &&
                               todosÁrbolB p i && todosÁrbolB p d
  
```

Para los árboles ordenados la función de búsqueda es más eficiente ya que si el dato no coincide con la raíz sólo hay que buscar en uno de los subárboles:

$perteneceBB$	$:: Ord a \Rightarrow a -> \text{ÁrbolB a} \rightarrow \text{Bool}$
$perteneceBB\ x\ VacíoB$	= False
$perteneceBB\ x\ (\text{NodoB i r d})$	= True
$ x == r$	= perteneceBB x i
$ x < r$	= perteneceBB x d
$ otherwise$	

de modo que como máximo se realizan tantas comparaciones como profundidad tenga el árbol. Dado que es posible almacenar $2^m - 1$ datos en un árbol de profundidad m , se puede localizar un dato en un árbol con n elementos realizando del orden de $\log_2 n$ comparaciones como máximo (siempre que el árbol tenga todos sus niveles completos). La ganancia es significativa frente a utilizar una lista, ya que con esta estructura puede ser necesario realizar n comparaciones.

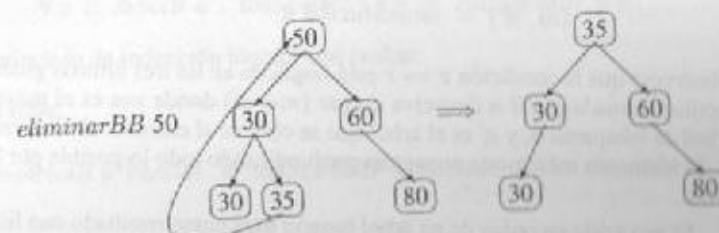


Figura 9.3: Eliminación de un dato.

Definimos ahora una función para insertar un nuevo dato dentro de un árbol de búsqueda, de modo que se obtenga otro árbol de búsqueda:

$insertarBB$	$:: Ord a \Rightarrow a -> \text{ÁrbolB a} \rightarrow \text{ÁrbolB a}$
$insertarBB\ x\ VacíoB$	= NodoB VacíoB x VacíoB
$insertarBB\ x\ (\text{NodoB i r d})$	= NodoB (insertarBB x i) r d
$ x \leq r$	= NodoB i r (insertarBB x d)
$ otherwise$	

Basta con ir comparando el dato a insertar con los distintos nodos del árbol y descender por el subárbol adecuado hasta alcanzar un subárbol vacío. La eliminación de un dato es un poco más complicada, ya que si el nodo a eliminar tiene dos subárboles no se puede dejar un hueco en su lugar. Una solución consiste en tomar el mayor elemento del subárbol izquierdo del nodo a eliminar y colocar éste en el hueco. De este modo el nuevo árbol seguirá siendo ordenado (ver Figura 9.3):

$esVacioB$	$:: \text{ÁrbolB a} \rightarrow \text{Bool}$
$esVacioB\ VacíoB$	= True
$esVacioB\ _-$	= False

```

eliminarBB :: Ord a => a -> ÁrbolB a -> ÁrbolB a
eliminarBB x VacíoB = VacíoB
eliminarBB x (NodoB i r d)
| x < r = NodoB (eliminarBB x i) r d
| x > r = NodoB i r (eliminarBB x d)
| esVacíoB i = d
| esVacíoB d = i
| otherwise = NodoB i' m d
where
  (m, i') = tomaMaxBB i
  tomaMaxBB (NodoB i r VacíoB) = (r, i)
  tomaMaxBB (NodoB i r d) = (m, NodoB i r d')
  where
    (m, d') = tomaMaxBB d

```

Obsérvese que la condición $x == r$ está implícita en las tres últimas guardas. La función auxiliar $tomaMaxBB a$ devuelve un par (ma, a') donde ma es el mayor elemento del árbol de búsqueda a , y a' es el árbol que se obtiene al eliminar el elemento ma del árbol a . El elemento máximo se encuentra profundizando todo lo posible por la derecha en el árbol.

El recorrido en orden de un árbol binario dará como resultado una lista con los datos de un árbol de modo que primero se recorre el subárbol izquierdo, luego la raíz y por último el derecho:

```

enOrden :: ÁrbolB a -> [a]
enOrden VacíoB = []
enOrden (NodoB i r d) = enOrden i ++ (r : enOrden d)

```

Aunque la complejidad de esta función es cuadrática, se obtiene una versión mejor con un parámetro acumulador:

```

enOrden' :: ÁrbolB a -> [a]
enOrden' x = aux x []
where
  aux :: ÁrbolB a -> [a] -> [a]
  aux VacíoB xs = xs
  aux (NodoB i r d) xs = aux i (r : aux d xs)

```

donde la especificación de la función auxiliar aux es $aux a xs = enOrden a ++ xs$. Una propiedad interesante es que si se realiza una visita en orden de un árbol de búsqueda se obtiene una lista ordenada:

```

MAIN> enOrden' a2
[30, 30, 35, 50, 60, 80] :: [Integer]

```

9.2 - Árboles binarios

por lo que es posible ordenar una lista de datos construyendo un árbol de búsqueda con sus elementos y recorriendo éste en orden:

```

listaAÁrbolBB :: Ord a => [a] -> ÁrbolB a
listaAÁrbolBB = foldr insertarBB VacíoB
treeSort :: Ord a => [a] -> [a]
treeSort = enOrden . listaAÁrbolBB

```

Por ejemplo:

```

MAIN> treeSort [4, 7, 1, 2, 9]
[1, 2, 4, 7, 9] :: [Integer]

```

Al construir un árbol de búsqueda con una lista de elementos ordenados se obtiene un árbol degenerado:

```

MAIN> listaAÁrbolBB [1, 2, 3]
NodoB (NodoB (NodoB VacíoB 1 VacíoB) 2 VacíoB) 3 VacíoB
:: ÁrbolB Integer

```

con lo que se pierde la eficiencia al acceder a los elementos del árbol. Cuando la lista esté ordenada podemos obtener un árbol de búsqueda no degenerado con la siguiente función:

```

lOrdAÁrbolBB :: Ord a => [a] -> ÁrbolB a
lOrdAÁrbolBB [] = VacíoB
lOrdAÁrbolBB xs = NodoB (lOrdAÁrbolBB ys) z (lOrdAÁrbolBB zs)
where
  (ys, z : zs) = partir xs
  partir xs = splitAt (length xs `div` 2) xs

```

La función predefinida $splitAt$ es utilizada para partir la lista en dos mitades:

```

MAIN> let xs = [1..6] in splitAt (length xs `div` 2) xs
([1, 2, 3], [4, 5, 6]) :: ([Integer], [Integer])

```

```

MAIN> lOrdAÁrbolBB [1..8]
NodoB (NodoB (NodoB (NodoB VacíoB 1 VacíoB) 2 VacíoB) 3
(NodoB VacíoB 4 VacíoB)) 5 (NodoB (NodoB VacíoB 6 VacíoB) 7
(NodoB VacíoB 8 VacíoB)) :: ÁrbolB Integer

```

Así, se distribuye la mitad de los elementos en cada subárbol.

9.2.2. FUNCIONES DE ORDEN SUPERIOR PARA ÁRBOLES BINARIOS

La definición de $fmap$ para árboles binarios es la siguiente:

```
instance Functor ÁrbolB where
  fmap f VacíoB = VacíoB
  fmap f (NodoB i r d) = NodoB (fmap f i) (f r) (fmap f d)
```

La siguiente función de plegado sustituye el constructor *VacioB* por el valor *e* y el constructor *NodoB* por la función *f*:

```
foldÁrbolB :: (b → a → b → b) → b → ÁrbolB a → b
foldÁrbolB f e = fun
  where
    fun VacíoB = e
    fun (NodoB i r d) = f (fun i) r (fun d)
```

Esta función también puede definirse como:

```
foldÁrbolB :: (b → a → b → b) → b → ÁrbolB a → b
foldÁrbolB f e VacíoB = e
foldÁrbolB f e (NodoB i r d) = f (foldÁrbolB f e i) r (foldÁrbolB f e d)
```

Muchas de las funciones definidas previamente de modo recursivo se definen de un modo más breve con *foldÁrbolB*:

```
perteneceB' :: Eq a ⇒ a → ÁrbolB a → Bool
perteneceB' x = foldÁrbolB (λ pi r pd → x == r || pi || pd) False
todosÁrbolB' :: (a → Bool) → ÁrbolB a → Bool
todosÁrbolB' p = foldÁrbolB (λ ti r td → p r && ti && td) True
```

9.2.3. INDUCCIÓN PARA ÁRBOLES BINARIOS

Sólo hay dos constructores para los árboles binarios definidos, por lo que el principio de inducción para árboles definidos¹ de este tipo es:

Principio de inducción para valores definidos del tipo ÁrbolB a

$$\forall x :: \text{ÁrbolB } a, P(x) \Leftrightarrow \left\{ \begin{array}{l} P(\text{VacíoB}) \\ \wedge \\ \forall i, d :: \text{ÁrbolB } a, \forall r :: a, \\ P(i) \wedge P(d) \Rightarrow P(\text{NodoB } i \ r \ d) \end{array} \right.$$

Es decir, a la hora de probar la propiedad *P* para cualquier árbol definido hay que probarla para el caso base (constructor *VacioB*), y para el paso inductivo (constructor *NodoB*) podemos suponer la propiedad cierta para los subárboles izquierdos y derechos, y hay

¹Los árboles infinitos necesitan una demostración para el caso ⊥.

que probar que la propiedad *P* sigue siendo cierta para cualquier árbol que se construya a partir de estos subárboles.

Ejemplo 9.6 Vamos a probar que las funciones:

```
todosÁrbolB p VacíoB = True
todosÁrbolB p (NodoB i r d) = p r &&
                                todosÁrbolB p i && todosÁrbolB p d
todosÁrbolB' p = foldÁrbolB (λ ti r td → p r && ti && td) True
```

calculan el mismo resultado:

Equivalencia de *todosÁrbolB* y *todosÁrbolB'*:

$$\forall x :: \text{ÁrbolB } a, \text{ todosÁrbolB } p x = \text{ todosÁrbolB}' p x$$

Por el principio de inducción bastará con probar:

- Caso Base:

$$\text{ todosÁrbolB } p \text{ VacíoB } = \text{ todosÁrbolB}' p \text{ VacíoB}$$

- Paso Inductivo:

$$\begin{aligned} \forall i, d :: \text{ÁrbolB } a, \forall r :: a, \\ \text{ todosÁrbolB } p i = \text{ todosÁrbolB}' p i \\ \wedge \\ \text{ todosÁrbolB } p d = \text{ todosÁrbolB}' p d \\ \Rightarrow \\ \text{ todosÁrbolB } p (\text{NodoB } i \ r \ d) = \text{ todosÁrbolB}' p (\text{NodoB } i \ r \ d) \end{aligned}$$

El caso base es trivial:

$$\begin{aligned} \text{ todosÁrbolB } p \text{ VacíoB} \\ \equiv \{ \text{ por definición de todosÁrbolB} \} \\ \text{ True} \end{aligned}$$

$$\begin{aligned} \text{ todosÁrbolB}' \text{ VacíoB} \\ \equiv \{ \text{ por definición de todosÁrbolB}' \} \\ \text{ foldÁrbolB} (\lambda ti r td \rightarrow p r \&\& ti \&\& td) \text{ True VacíoB} \\ \equiv \{ \text{ por definición de foldÁrbolB} \} \\ \text{ True} \end{aligned}$$

con lo que queda demostrado el caso base. Para el paso inductivo desarrollamos primero la parte izquierda:

$$\begin{aligned} & \text{todosÁrbolB } p (\text{NodoB } i \ r \ d) \\ \equiv & \{ \text{por definición de todosÁrbolB} \} \\ & p \ r \ \&& \text{todosÁrbolB } p \ i \ \&& \text{todosÁrbolB } p \ d \\ \equiv & \{ \text{por hipótesis de inducción} \} \\ & p \ r \ \&& \text{todosÁrbolB}' \ p \ i \ \&& \text{todosÁrbolB } p \ d \\ \equiv & \{ \text{por hipótesis de inducción} \} \\ & p \ r \ \&& \text{todosÁrbolB}' \ p \ i \ \&& \text{todosÁrbolB}' \ p \ d \end{aligned}$$

y ahora la parte derecha:

$$\begin{aligned} & \text{todosÁrbolB}' \ p (\text{NodoB } i \ r \ d) \\ \equiv & \{ \text{por definición de todosÁrbolB}' \} \\ & \text{foldÁrbolB } (\lambda \ ti \ r \ td \rightarrow p \ r \ \&& \ ti \ \&& \ td) \ \text{True} \ (\text{NodoB } i \ r \ d) \\ \equiv & \{ \text{llamando } h \text{ a } (\lambda \ ti \ r \ td \rightarrow p \ r \ \&& \ ti \ \&& \ td) \} \\ & \text{foldÁrbolB } h \ \text{True} \ (\text{NodoB } i \ r \ d) \\ \equiv & \{ \text{por definición de foldÁrbolB} \} \\ & h(\text{foldÁrbolB } h \ \text{True} \ i) \ r (\text{foldÁrbolB } h \ \text{True} \ d) \\ \equiv & \{ \text{por definición de todosÁrbolB}' \} \\ & h(\text{todosÁrbolB}' \ p \ i) \ r (\text{foldÁrbolB } h \ \text{True} \ d) \\ \equiv & \{ \text{por definición de todosÁrbolB}' \} \\ & h(\text{todosÁrbolB}' \ p \ i) \ r (\text{todosÁrbolB}' \ p \ d) \\ \equiv & \{ \text{por definición de } h \} \\ & p \ r \ \&& \text{todosÁrbolB}' \ p \ i \ \&& \text{todosÁrbolB}' \ p \ d \end{aligned}$$

con lo que quedan igualadas las partes izquierda y derecha, lo que establece el paso inductivo y, junto con el caso base, la equivalencia de ambas funciones.

9.3. ARRAYS

Una estructura muy utilizada en programación es el *array* o *vector*. Se trata de un *contenedor* que almacena una colección de datos del mismo tipo, de modo que cada dato almacenado tiene asociada una posición. El acceso a los datos se hace mediante la posición correspondiente. Suelen existir operaciones para crear un array, y para acceder y modificar el elemento que ocupa una posición.

9.3.1. UNA IMPLEMENTACIÓN INEFICIENTE

Una implementación trivial de arrays se realiza mediante listas:

`type Array a = [a]`

de modo que la función que crea un array a partir de una lista de elementos es la identidad:

9.3 - Arrays

`creaArray :: [a] → Array a`
`creaArray xs = xs`

`ar :: Array Char`
`ar = creaArray ['a'..'d']`

Si consideramos que las posiciones correspondientes a un array de n elementos están en el rango $[0..n - 1]$, podemos seleccionar el elemento que ocupa cierta posición del siguiente modo:

`infixl 9 !`
`(!) :: Array a → Integer → a`
`(x : xs) ! 0 = x`
`(_ : xs) ! (n + 1) = xs ! n`
`_ ! _ = error "(!): Posición no válida"`

Por ejemplo, se puede acceder a la primera y última posición del array ejemplo mediante el operador anterior:

`MAIN> (ar!0, ar!3)`
`('a', 'd') :: (Char, Char)`

El operador que permite obtener un nuevo array modificando el elemento almacenado en cierta posición es el siguiente:

`infixl 8 :=`
`(=:) :: Array a → (Integer, a) → Array a`
`(x : xs) =: (0, y) = (y : xs)`
`(x : xs) =: (n + 1, y) = x : (xs =: (n, y))`
`_ =: _ = error "(=:): Posición no válida"`

El segundo argumento del operador anterior es un par con la posición a modificar junto con el nuevo valor:

`MAIN> let br = ar =: (3, 'z') in br ! 3`
`'z' :: Char`

Gracias a la asociatividad izquierda del operador es posible realizar varias modificaciones de un modo compacto:

`MAIN> let br = ar =: (0, 'A') =: (3, 'Z') in (br!0, br!1, br!2, br!3)`
`('A', 'B', 'C', 'Z') :: (Char, Char, Char, Char)`

Es posible representar una matriz mediante un array de arrays:

`type Matriz = Array (Array Float)`

```

matriz :: Int → Matriz
matriz n = creaArray (replicate n creaFila)
where
  creaFila = creaArray (replicate n 0.0)

```

donde la función predefinida `replicate` crea una lista con un número de copias de un objeto:

```

MAIN> replicate 3 True
[True, True, True] :: [Bool]

```

Podemos acceder a un elemento concreto usando dos veces el operador (`!`), gracias a la asociatividad izquierda de éste. Por ejemplo, para acceder al último elemento de la primera fila de una matriz de dimensión tres podemos usar:

```

MAIN> let m = matriz 3 in m ! ! 2
0.0 :: Float

```

El principal problema de la implementación de arrays mediante listas es la falta de eficiencia. Obsérvese que para acceder o modificar el último elemento del array es necesario pasar por todos los que le preceden, por lo que en el peor caso se realizarán tantas llamadas recursivas como tamaño tenga el array.

9.3.2. UNA IMPLEMENTACIÓN EFICIENTE

Una implementación más eficiente de arrays se realiza mediante árboles binarios. Con esta implementación, un array de n elementos se representará mediante un árbol de altura $\log_2 n$, de modo que en el peor de los casos las operaciones (`!`) y (`=`) serán también de este orden. La ganancia es significativa frente a utilizar una lista, ya que con esta estructura el número de llamadas recursivas era del orden de n .

En la nueva implementación utilizaremos el siguiente tipo que permite representar árboles binarios no vacíos que almacenen datos tan sólo en sus hojas:

```

data ÁrbolH a = HojaH a | NodoH (ÁrbolH a) (ÁrbolH a) deriving Show
type Array a = ÁrbolH a

```

Para construir un árbol a partir de una lista situamos los elementos que ocupen posiciones pares en el subárbol izquierdo (el elemento en la cabeza de la lista ocupa una posición par: la posición cero) y los pares en el derecho. Repetimos este proceso en cada subárbol hasta obtener listas de un solo elemento representadas mediante nodos hoja:

```

creaArray :: [a] → Array a
creaArray [x] = HojaH x
creaArray xs@( _ : _ : _ ) = NodoH (creaArray s) (creaArray d)
where
  (s, d) = partir xs

```

```

partir []      :: [a] → ([a], [a])
partir []     = ([], [])
partir [x]    = ([x], [])
partir (x : y : zs) = (x : xs, y : ys)
where
  (xs, ys) = partir zs
xr :: Array Char
xr = creaArray 'a' .. 'd'

```

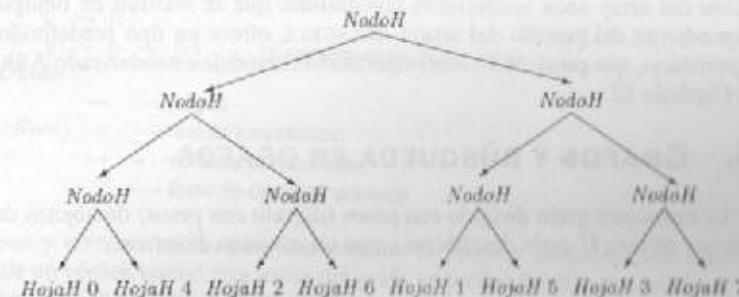


Figura 9.4: Árbol para `creaArray [0..7]`.

Por ejemplo, para la expresión `creaArray [0..7]` el árbol obtenido es el representado en la Figura 9.4. Para acceder a una posición n par del array buscamos en el subárbol izquierdo la posición $n \text{ `div' } 2$, mientras que para acceder a una impar lo hacemos en el derecho. El caso base consiste en acceder a la posición cero de un nodo hoja:

```

infixl 9 !
(!) :: Array a → Integer → a
HojaH x ! 0 = x
NodoH i d ! n = if even n then
  i ! (n `div` 2)
  else
    d ! (n `div` 2)
  _ = error "posición no válida"

```

El operador para modificar una posición sigue el mismo patrón recursivo, pero se copia en cada llamada recursiva el subárbol (izquierdo o derecho) que no es modificado. En el caso base se modifica la hoja correspondiente:

```

infixl 8 :=
(=) :: Array a -> (Integer, a) -> Array a
HojaH x =: (0, y) = HojaH y
NodoH i d =: (n, y) = if even n then
    NodoH (i =: (n `div` 2), y)) d
  else
    NodoH i (d =: (n `div` 2), y))
=:= error "(=): Posición no válida"

```

Obsérvese que para las dos operaciones se realizan tantas llamadas recursivas como altura tenga el árbol.

En los lenguajes imperativos es habitual que el acceso y la modificación de una posición del array sean operaciones predefinidas que se realizan en tiempo constante (independiente del tamaño del array). HASKELL ofrece un tipo predefinido con estas características, que puede utilizarse importando el módulo estandarizado ARRAY tratado en el Capítulo 12.

9.4. GRAFOS Y BÚSQUEDA EN GRAFOS

La estructura grafo dirigido con pesos (digrafo con pesos) de objetos de tipo base T y pesos de tipo P suele describirse como un conjunto de temas:

$$G = \{(x, y, p) \mid x, y \in T \wedge p \in P \wedge p(x, y, p)\}$$

siendo p cierto predicado o relación. Pueden describirse las operaciones sobre grafos con tales estructuras pero hacemos notar dos inconvenientes:

- los elementos de T que no son origen de ningún arco hay que describirlos de forma artificial (p.e., con un peso especial o ficticio),
- certas operaciones, como la búsqueda de caminos, calculan eficientemente cuáles son los vértices conectados con cada vértice x (sucesores de x).

$$S(x) = \{(y, p) \mid (x, y, p) \in G\}$$

Si representamos vértices y sucesores con listas, podemos describir un grafo en la forma:

$$\text{data Grafo } v \text{ } p = G [v] (r \rightarrow [(v, p)])$$

De esta forma tenemos resueltos los dos problemas planteados al principio: están perfectamente definidos los vértices, y eficientemente localizables los sucesores de cada vértice. Si se tratara de un grafo sin pesos se consideraría la representación simplificada:

$$\text{data Grafo } v = G [v] (v \rightarrow [v])$$

donde el constructor G permite identificar la estructura.

9.4 - Grafos y búsqueda en grafos

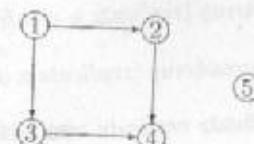


Figura 9.5: Un grafo.

Ejemplo 9.7 Según la representación anterior, el código del grafo de la Figura 9.5 sería:

```

g = G [1, 2, 3, 4, 5] suc
where
  suc 1 = [2, 3]
  suc 2 = []
  suc 3 = [4]
  suc _ = []

```

La representación anterior es interesante y sencilla, pero veremos una representación más rica utilizando el sistema de clases de HASKELL.

9.4.1. BÚSQUEDA EN ANCHURA Y EN PROFUNDIDAD

La mayoría de algoritmos sobre (di)grafos examinan o procesan los vértices o arcos en cierto orden. La búsqueda en profundidad (*depth-first search*) y la búsqueda en anchura (*breadth-first search*) son las dos formas de recorridos más utilizados.

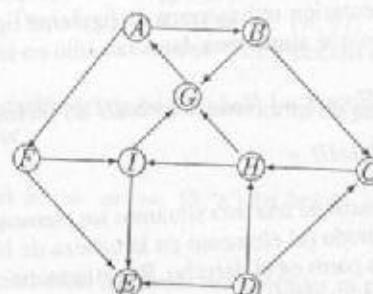


Figura 9.6: Otro grafo.

El recorrido en profundidad (BEP) es una generalización del recorrido en profundidad en un árbol. La idea es que, partiendo de un vértice inicial (que es el primero en procesarse), se toma un sucesor y se busca en profundidad a partir de tal sucesor, para

después proseguir con el resto de los sucesores del inicial. Por ejemplo, para el grafo de la Figura 9.6, tenemos el árbol de visitas en profundidad de la Figura 9.7 (numeramos los arcos según el orden de visita comenzando desde el nodo A). Obsérvese que no se visitan todos los vértices ya que el grafo no es conexo (en estos casos interesa un algoritmo que realice una BEP a lo largo de un *bosque* de tales árboles).

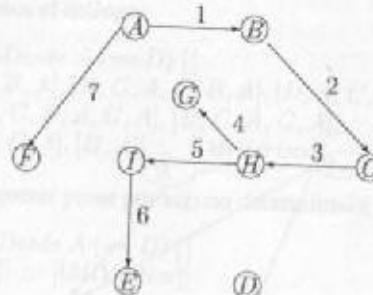


Figura 9.7: Árbol de visitas en profundidad.

En la búsqueda en anchura (BEA) los vértices se visitan en orden creciente de distancia (arcos recorridos) desde el punto de comienzo. Por ejemplo, para el digrafo de la Figura 9.6, tenemos el árbol BEA de la Figura 9.8 y, al igual que con la BEP, no se alcanzan todos los vértices. Antes de ver las funciones que realizan el recorrido sobre grafos veamos una interesante representación de éstos.

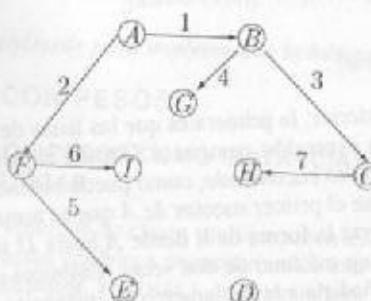


Figura 9.8: Árbol de visitas en anchura.

9.4.2. LOS GRAFOS COMO INSTANCIAS DE UNA CLASE UNIPARAMÉTRICA

Una alternativa a la representación:

data Grafo v = G [v] { v → [v] }

es proporcionada por el sistema de clases de HASKELL, donde las funciones esenciales (*vértices* y *sucesor*) se capturan como miembros de una clase, en la cual incorporamos otras funciones para resolver problemas típicos de búsqueda de caminos:

```

class Eq v ⇒ Grafo v where
  vértices :: [v]
  suc     :: v → [v]
  camino :: v → v → [v]   -- un camino entre dos vértices
  caminoDesde :: 
    v           → -- origen
    (v → Bool) → -- test de localización
    [v]         → -- vértices ya recorridos
    [[v]]       → -- listas de caminos solución
  
```

Las funciones *suc* y *vértices* bastan para implementar la función *caminoDesde* (que realiza, a partir de un vértice inicial, una visita del grafo en profundidad hasta satisfacer el test de localización) si podemos decidir cuándo un vértice ya ha sido recorrido y de esta forma evitar los posibles *ciclos*. Normalmente tal decisión viene dada por una función (\neq), que leeremos *no pertenece*², cuyo tipo es:

$(\neq) :: Eq v \Rightarrow v \rightarrow [v] \rightarrow Bool$

y que comprueba si un objeto no es miembro de una lista; una versión trivial es:

$x \neq ys = and [x \neq y | y \leftarrow ys]$

que no es más que la función *notElem* de PRELUDE. Sin embargo, en aplicaciones particulares puede interesar otra forma de *poda* de la búsqueda. Por ello podríamos incorporar tal función a la clase dando un valor por defecto, así como también tendrá una definición por defecto el algoritmo de búsqueda en profundidad:

```

class Eq v ⇒ Grafo v where
  vértices      :: [v]
  suc          :: v → [v]
  ( $\neq$ )        :: v → [v] → Bool
  camino       :: v → v → [v]
  caminoDesde :: v → (v → Bool) → [v] → [[v]]
  
```

²El código *real* para el símbolo \neq puede ser $</=$, al igual que el código para \rightarrow lo escribimos en la forma $<->$.

```
-- miembros por defecto
x ≠ ys      = and [x ≠ y | y ← ys]
camino u v   = head (caminoDesde u (== v) [])
caminoDesde o te vis = ...
```

donde *caminoDesde o te vis* devuelve la lista de todos los caminos partiendo desde el vértice *o* hasta satisfacer el test de localización *te* (o método de poda), siendo el tercer argumento *vis* la lista de vértices ya visitados (que inicialmente será una lista vacía). Por ejemplo, para el test de localización ($\equiv v$) tendremos caminos que terminan en el vértice *v*. De entre todos los caminos, seleccionando el primero (por ejemplo) tenemos un único camino, y ésta puede ser una definición por defecto:

```
camino u v = head (caminoDesde u (== v) [])
```

Además, si sabemos que nuestro grafo es acíclico entonces nos interesarán definir la función (\neq) en la forma $- \neq - = \text{True}$.

Veamos ahora la definición de *caminoDesde*. En primer lugar habrá que comprobar si el vértice origen verifica el test:

```
caminoDesde o te vis
| te o      = [o : vis]
| otherwise = ...
```

y en ese caso tenemos una solución *o : vis* que incluimos en forma de lista: $[o : vis]$. Por esta razón, en todos los caminos la lista de vértices visitados estará invertida. En el caso de que el vértice no verifique el test de localización, tendremos que realizar una búsqueda (en profundidad) desde *o* y para ello habrá que buscar (también en profundidad) desde los sucesores de *o* que no hayan sido ya visitados; esto puede realizarse con la lista:

```
(camA) [caminoDesde o' te (o : vis) | o' ← suc o, o' ≠ vis]
```

en la cual sólo se seleccionan los sucesores de *o* no visitados, con objeto de evitar los *ciclos*; de esta forma se obtiene una lista de listas de listas. Finalmente tendremos que concatenarlas todas y obtendremos la definición final:

```
caminoDesde o te vis
| te o      = [o : vis]
| otherwise = concat [caminoDesde o' te (o : vis) | o' ← suc o, o' ≠ vis]
```

La evaluación perezosa de una llamada a *caminoDesde* realiza una visita en profundidad, ya que la llamada a la función *concat* evalúa el primer elemento de la lista (*camA*), que a su vez realiza una búsqueda en profundidad a partir de un sucesor no visitado de *o*.

Ejemplo 9.8 (Una instancia de la clase Grafo) Consideraremos el grafo de la Figura 9.9 representado por la siguiente instancia de nuestra clase *Grafo*:

```
data MiVértice = A | B | C | D | E deriving (Show, Eq, Eum)
instance Grafo MiVértice where
    vértices = [A .. E]
    suc A   = [B, C, D]
    suc B   = [C]
    suc C   = [A, D]
    suc D   = [C]
    suc E   = []
```

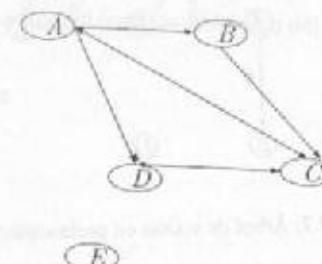


Figura 9.9: Un grafo sencillo.

Con tal definición, el lector podrá comprobar que tenemos el siguiente diálogo:

```
MAIN> caminoDesde A (== D) []
[[D, C, B, A], [D, C, A], [D, A]] :: [[MiVértice]]
MAIN> camino A D
[D, C, B, A] :: [MiVértice]
```

Dos observaciones al diálogo anterior: la primera es que las listas de caminos (aparecen invertidas!); la segunda es que la expresión *camino A D* toma el primer elemento de la lista, y por tanto es el primer camino encontrado, como puede obtenerse de la expresión de la lista dada en (*camA*), ya que el primer sucesor de *A* que se toma es el *B*. Supongamos ahora que queremos encontrar la forma de ir desde *A* hasta *D* si permitimos pasar por cada vértice (distinto de *D*) un máximo de dos veces. Entonces debemos modificar la versión de la función (\neq) y añadirla a la declaración de instancia:

```
instance Grafo MiVértice where
    vértices = [A .. E]
    ...
    v ≠̄ us = v `notElem` (us \\ [v])
```

donde la función ($\backslash\backslash$) calcula la diferencia de listas. Tal función aparece en el módulo *List*, aunque damos aquí la siguiente definición alternativa:

```
(\ \) :: Eq a => [a] -> [a] -> [a]
(\ \) = foldl elimDe
```

```
elimDe          :: Eq a => [a] -> a -> [a]
elimDe []      x = []
elimDe (y : ys) x = if x == y then ys else y : elimDe ys x
```

De esta forma tendríamos el diálogo:

```
MAIN> caminoDesde A (== D) []
[[D, C, B, A, C, B, A], [D, C, A, C, B, A], [D, A, C, B, A],
 [D, C, B, A], [D, C, B, A, C, A], [D, C, A, C, A],
 [D, A, C, A], [D, C, A], [D, A]] :: [[MiVertice]]
```

Incluso podríamos no querer pasar por alguno determinado, como por ejemplo el *B*:

```
MAIN> caminoDesde A (== D) []
[[D, C, A], [D, A]] :: [[MiVertice]]
```

En este caso, únicamente tenemos que modificar la versión por defecto del algoritmo de poda en la forma:

v $\not\leftarrow$ *vs* = *v* `notElem` *vs* && *v* \neq *B*

Tips&tricks

Ejercicio 9.9

1. Instancie la clase *Grafo* para obtener el grafo de la Figura 9.6.
2. Considerando la instancia de la clase *Grafo* del Ejemplo 9.8 analice la llamada *caminoDesde A (== A) []* si eliminamos la definición de la función $\not\leftarrow$ en la instancia.
3. Modifique *caminoDesde* para implementar la búsqueda en anchura.

9.5. GRAFOS CON PESOS

9.5.1. GRAFOS CON PESOS COMO INSTANCIAS DE CLASES BIPARAMÉTRICAS

Los ejemplos más sencillos de clases multiparamétricas son aquellos que permiten implementar tipos abstractos genéricos parametrizados con varias variables de tipo con restricciones. Desgraciadamente, HASKELL no permite las clases multiparamétricas, pero sí HUGS (si activamos el directive de interpretación convenientemente). Así, varias clases pueden enriquecerse en forma conjunta para formar una nueva clase que tenga como objeto declarar (y posiblemente definir por defecto) nuevas funciones:

```
class (C1 a, C2 b, ...) => C a b where ...
-- funciones a añadir
```

9.5 - Grafos con pesos

Si un tipo aparece en una instanciación de algún parámetro en una clase multiparamétrica entonces el tipo hereda las funciones miembro de la clase; con esto se permite mezclar tipos para obligar a que cierto tipo o función tenga un comportamiento dependiente de los restantes tipos o funciones. Ejemplos típicos de clases biparamétricas los proporcionan los grafos con pesos, donde el tipo asociado a los pesos es un tipo que hereda una serie de operaciones como:

```
(+), (-) -- suma y diferencia de pesos
(>), (≥) -- comparación de pesos
```

que son proporcionadas por las clases *Num* y *Ord*, que deben aparecer en el contexto; por otro lado, sería interesante resolver problemas como búsqueda en profundidad o atajos (caminos con peso menor que cierto valor), por lo que tendremos algo como:

```
data Arco v p = Arc v p
type Arcos v p = [Arco v p]

class (Eq v, Num p, Ord p) => Grafo v p where
    vértices :: [v]
    suc     :: v -> Arcos v p
    (+)    :: (v, p) -> [a] -> Bool
    atajos :: v -> v -> p -> [[v]]
    -- camino con peso total no mayor que el tercer argumento
```

Si una clase tiene varios parámetros cualquier miembro debe "tocar" todos los parámetros ya que de lo contrario habría problemas de ambigüedad al resolver la sobrecarga; por ejemplo, en el tipo de la función *vértices* no aparece el parámetro *b*, por lo que dicho tipo sería ambiguo en el contexto de las declaraciones:

```
instance Grafo V Int where ...
instance Grafo V Float where ...
```

Este problema se resuelve, bien con el tipado explícito:

```
if x `elem` vértices :: [Int] then ...
```

o bien utilizando un parámetro fícticio (*dummy*):

```
class (Eq a, Num b) => Grafo a b where
    vértices :: b -> [a] -- primer argumento mudo
    ...
```

de forma que en el tipo de *vértices* el primer argumento es mudo porque no se usa más que para resolver el problema de ambigüedad; así, en las instancias no se especifica:

```
instance Grafo Int Int where
    vértices _ = [1..6]
```

y el tipo final se deduce del contexto. Para la búsqueda en profundidad incluiremos una nueva función *pCaminoDesde* que ahora necesita algunos argumentos adicionales:

<i>pCaminoDesde</i> ::	
<i>v</i>	→ -- origen
<i>(v → p → Bool)</i>	→ -- test de encontrado
<i>(p → Bool)</i>	→ -- test de retorno
<i>[v]</i>	→ -- vértices ya recorridos
<i>p</i>	→ -- peso ya consumido
<i>[[v]]</i>	

Los argumentos extra son esenciales para una poda del recorrido del grafo que dependa también de una función de test de retorno que permite realizar la vuelta atrás para ciertos valores del peso acumulado; una implementación por defecto (que debe aparecer por tanto como ecuación dentro de la clase) suficientemente general puede ser:

```

pCaminoDesde o te tr vis p
  | te o p = [o : vis]
  | otherwise = concat [pCaminoDesde o' te tr (o : vis) np |
    Arc o' p' — suc o,
    (o', p')  $\not\in$  vis, let np = p + p', not (tr np)]
  
```

El resto de operaciones por defecto se puede definir a partir de las operaciones ya descritas en esta sección.

9.5.2. UNA CLASE HASKELL PARA GRAFOS CON PESOS

En la versión de HASKELL actual no es posible utilizar clases multiparamétricas. Sin embargo podemos dar una solución alternativa introduciendo contextos específicos en los tipos de las funciones de la clase. Por ejemplo, consideremos la siguiente clase:

```

class Eq v => GrafoPesos v where
  pVértices :: [v]
  pSuc :: Num p => v → Arcos v p
  ( $\neq$ ) :: Num p => (v, p) → [v] → Bool
  atajos :: (Num p, Ord p) => v → v → p → [[v]]
  pCaminoDesde :: Num p =>
    v → (v → p → Bool) → (p → Bool) → [v] → p → [[v]]
  
```

donde hemos cambiado los identificadores de las funciones para distinguirlas de las elegidas para la clase *Grafo*. Además, vemos que en el tipo de la función *atajos* hemos añadido el contexto *(Num p, Ord p)* ya que muy posiblemente necesitaremos sumar pesos, o comparar pesos, con el objetivo de obtener únicamente caminos con un peso máximo consumido menor que cierto valor. Como se observa, de nuevo en el test para capturar los vértices visitados hemos incluido un nuevo argumento, ya que éste puede ser utilizado para comprobar *visitas especiales*, aunque lo normal es que el valor por defecto de esta comprobación no lo utilice. Veamos la colección de definiciones por defecto:

9.5 - Grafos con pesos

-- definiciones por defecto en la clase *GrafoPesos*

$(x, _) \neq ys = \text{and } [x \neq y \mid y \leftarrow ys]$
 $\text{atajos } v \ u' pt = pCaminoDesde v (\lambda x _ \rightarrow x == u') (> pt) [] 0$
 $pCaminoDesde o te tr vis p = \dots$ — igual que en la Sección 9.5.1

Como puede observarse el valor por defecto de \neq no usa el peso. En la función *atajos* observamos que en el test de búsqueda $(\lambda x _ \rightarrow x == u')$ tampoco utilizamos el peso. Sin embargo, en el contexto de tal función aparece *Ord p* ya que utilizamos el test de retorno ($> pt$). Una posible instancia para el grafo con pesos correspondiente a la Figura 9.10 puede ser:

```

instance GrafoPesos Int where
  pSuc 1 = [Arc 2 30, Arc 3 20, Arc 4 40]
  pSuc 3 = [Arc 4 50]
  pSuc 2 = [Arc 4 20]
  
```

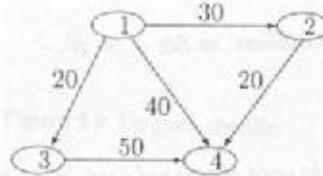


Figura 9.10: Un grafo con pesos.

En el grafo de la figura vemos que es posible ir del vértice 1 al 4 de varias formas, pero los atajos con peso máximo 60 solamente son dos. Podemos comprobarlo con el siguiente diálogo:

```

MAIN> atajos 1 4 40 :: [[Int]]
[[4, 1]] :: [[Int]]
MAIN> atajos 1 4 50 :: [[Int]]
[[4, 2, 1], [4, 1]] :: [[Int]]
MAIN> atajos 1 4 70 :: [[Int]]
[[4, 2, 1], [4, 3, 1], [4, 1]] :: [[Int]]
MAIN> atajos (1 :: Int) 4 70
[[4, 2, 1], [4, 3, 1], [4, 1]] :: [[Int]]
  
```

Como se observa, ya que el objeto 1 es compartido por varios tipos, es necesario dar el tipo del resultado (*atajos 1 4 40 :: [[Int]]*) para evitar ambigüedades, aunque también se puede dar el tipo de un argumento, como por ejemplo en la última línea del diálogo.

9.6. EJERCICIOS

9.10 Considérese la siguiente función para calcular la profundidad de un árbol binario:

$$\begin{aligned} \text{profundidadB} &:: \text{ÁrbolB } a \rightarrow \text{Integer} \\ \text{profundidadB VacíoB} &= 0 \\ \text{profundidadB} (\text{NodoB } i \tau d) &= 1 + \max(\text{profundidadB } i) (\text{profundidadB } d) \end{aligned}$$

Demuestre la siguiente propiedad:

Profundidad:

$$\forall x :: \text{ÁrbolB } a . \text{ profundidadB } x \geq 0$$

9.11 Sea el siguiente tipo para representar árboles binarios que almacenan datos tan sólo en sus hojas:

$$\text{data ÁrbolH } a = \text{HojaH } a \mid \text{NodoH} (\text{ÁrbolH } a) (\text{ÁrbolH } a) \text{ deriving Show}$$

Defina las funciones:

1. *profundidadH* que calcule la profundidad de un árbol.
2. *tamañoH* que calcule el tamaño de un árbol.
3. *perteneceH* que compruebe si un dato pertenece a un árbol.
4. *todosÁrbolH* que compruebe si todos los datos almacenados en un árbol cumplen una condición.
5. *fmap* que aplique una función a todos los datos de un árbol.

9.12 Redefina las funciones propuestas en el Ejercicio 9.11 con la función de plegado:

$$\begin{aligned} \text{foldÁrbolH} :: (b \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow \text{ÁrbolH } a \rightarrow b \\ \text{foldÁrbolH } f g (\text{HojaH } x) = g x \\ \text{foldÁrbolH } f g (\text{NodoH } i d) = f (\text{foldÁrbolH } f g i) (\text{foldÁrbolH } f g d) \end{aligned}$$

9.13 Enuncie el principio de inducción para el tipo *ÁrbolH*. Demuestre que la función:

$$\begin{aligned} \text{espejoH} &:: \text{ÁrbolH } a \rightarrow \text{ÁrbolH } a \\ \text{espejoH} (\text{HojaH } x) &= (\text{HojaH } x) \\ \text{espejoH} (\text{NodoH } i d) &= \text{NodoH} (\text{espejoH } d) (\text{espejoH } i) \end{aligned}$$

verifica la propiedad:

Doble espejo:

$$\forall x :: \text{ÁrbolH } a . \text{ espejoH} (\text{espejoH } x) = x$$

9.6 - Ejercicios

9.14 Sea la siguiente clase:

$$\begin{aligned} \text{class TieneMáximo } f \text{ where} \\ \text{máximo} :: \text{Ord } a \Rightarrow f a \rightarrow a \end{aligned}$$

para estructuras de datos para las que tiene sentido calcular el máximo elemento almacenado. Realice instancias de la clase con listas y todos los tipos de árboles estudiados en el capítulo:

$$\begin{aligned} \text{instance TieneMáximo [] where ...} \\ \text{instance TieneMáximo ÁrbolB where ...} \\ \dots \end{aligned}$$

9.15 Defina una clase *TieneTamaño* con un método *tamaño* para calcular cuantos datos almacena una estructura. Realice instancias de la clase anterior con listas y todos los tipos de árboles estudiados en el capítulo.

9.16 Sea la siguiente estructura para representar árboles con claves en los nodos y múltiples hijos:

$$\begin{aligned} \text{data Arb } c = V &\quad -- \text{el árbol vacío} \\ | N c [\text{Arb } c] &\quad -- \text{nodo con clave de tipo } c \text{ y lista con subárboles} \\ \text{deriving Show} \end{aligned}$$

1. Defina la función:

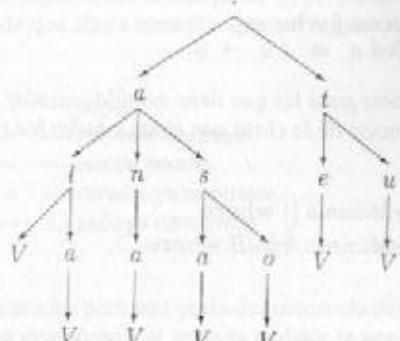
$$\begin{aligned} \text{reduce} &:: ([b] \rightarrow c \rightarrow b) \rightarrow \text{Arb } c \rightarrow b \rightarrow b \\ \text{reduce } g \ a z &= \dots \\ &\quad -- \text{reduce todos los elementos del árbol } a \text{ a un único elemento por aplicación} \\ &\quad -- \text{recursiva de la función de reducción } g \text{ a partir del valor inicial } z \end{aligned}$$

2. Escriba, en función de *reduce*, las siguientes funciones:

$$\begin{aligned} \text{aplica} &:: (c \rightarrow b) \rightarrow \text{Arb } c \rightarrow \text{Arb } b \\ \text{aplica } f t &= \dots \\ &\quad -- \text{devuelve un árbol con la misma estructura y} \\ &\quad -- \text{con las imágenes mediante } f \text{ de los nodos} \end{aligned}$$

$$\begin{aligned} \text{prof} &:: \text{Arb } c \rightarrow \text{Int} \\ \text{prof } t &= \dots \quad -- \text{la profundidad del árbol } t \end{aligned}$$

$$\begin{aligned} \text{visita} &:: \text{Arb } c \rightarrow [c] \\ \text{visita } t &= \dots \quad -- \text{las claves de } t \text{ para un recorrido en preorden} \end{aligned}$$



Palabras: al, ala, ana, asa, uso, te, tu

$N'! [N'a' [N'! [V, N'a'[V]]]$
 $, N'n' [N'a'[V]]$
 $, N's' [N'a'[V], N'o'[V]]]$
 $, N't' [N'e'[V], N'u'[V]]]$

Figura 9.11: Un diccionario.

3. Tomemos los árboles anteriores para representar diccionarios de palabras.

`type Dic = Arb Char`

organizadas en orden alfabetico por iniciales de sus prefijos (ver Figura 9.11) (la raíz se representa con el carácter '!'). Escriba las siguientes funciones:

`pals :: Dic → [String]`
`pals d = ... -- la lista de las palabras de d en orden alfabetico`

`aMayus :: Dic → Dic`
`aMayus d = ... -- transforma a mayúsculas todas las palabras del diccionario d`

`estáEn :: Dic → String → Bool`
`p'estáEn' d = ... -- la palabra p está en el diccionario d`

`(:<) :: Dic → String → Dic`
`d <: p = ... -- inserta en forma ordenada la palabra p en el diccionario d`

9.17 (Códigos de Huffman, 1952) Dado un alfabeto A , podemos definir un código variable donde ninguno de los elementos codificados sea prefijo de otro. Un ejemplo de código variable binario para el alfabeto $A=\{a, b, c, d, e\}$ puede ser:

$(a, 11) (b, 10) (c, 01) (d, 001) (e, 000)$

La codificación del mensaje "abace" será "11101101000". La decodificación puede realizarse sin ambigüedad ya que ningún código es prefijo de otro; por ejemplo, la decodificación del mensaje "010001011000" produce "cebae". La ventaja de utilizar códigos variables es patente cuando tenemos un alfabeto del que conocemos la frecuencia de aparición de los elementos. Si a los elementos más frecuentes le damos una codificación más corta, podremos conseguir enviar mensajes con menos código. David Albert Huffman creó un algoritmo para construir códigos de este tipo (llamados códigos Huffman); vamos a describirlo paralelamente al ejemplo anterior. El código anterior resulta de considerar las frecuencias:

$(a, 35\%) (b, 25\%) (c, 21\%) (d, 15\%) (e, 4\%)$

Se pretende construir un árbol binario:

`data Huf a = Hoja (a, Int) | Nodo (Huf a) Int (Huf a) deriving Show`
en el cual la clave numérica de un Nodo sea la suma de las frecuencias de sus dos ramas. Procedemos del siguiente modo:

- Con cada elemento del alfabeto se construye una Hoja y se coloca en una lista as ordenada por frecuencias, de menores a mayores,
- en la lista as quedé más de un elemento, extraer los dos primeros elementos de la lista, formar un nuevo nodo (un árbol) con esos dos elementos como ramas (y con la suma de frecuencias como clave numérica) para después insertarlo en la lista en su lugar adecuado (dependiendo de la frecuencia).
- Cuando quede un elemento tomaremos éste como árbol de Huffman.

En nuestro caso,

`as = [Hoja(e, 4), Hoja(d, 15), Hoja(c, 21), Hoja(b, 25), Hoja(a, 35)]`
`as = [Nodo(Hoja(e, 4)) 19 (Hoja(d, 15),`
`Hoja(c, 21), Hoja(b, 25), Hoja(a, 35))]`
`as = [Hoja(b, 25), Hoja(a, 35),`
`Nodo(Nodo(Hoja(e, 4)) 19 (Hoja(d, 15))) 40 (Hoja(c, 21))]`

$$as = [Nodo(Nodo(Hoja(e, 4)) 19 (Hoja(d, 15))) 40 (Hoja(c, 21)),\\ \quad Nodo(Hoja(b, 25)) 60 (Hoja(a, 35))]$$

$$as = [Nodo(Nodo(Hoja(e, 4)) 19 (Hoja(d, 15))) 40 (Hoja(c, 21))),\\ \quad 100 \\ \quad (Nodo(Hoja(b, 25)) 60 (Hoja(a, 35)))]$$

y este único elemento es el árbol de Huffman buscado. Ahora codificamos cada elemento según la regla:

- Avanzar a la izquierda es un 0.
- Avanzar a la derecha es un 1.

Así,

$$(e, 000)(d, 001)(c, 01)(b, 10)(a, 11)$$

que era el código inicial. Construya las funciones necesarias para:

1. Dada una lista de elementos y frecuencias, construir el árbol de Huffman.
2. Dado un elemento del alfabeto y un árbol de Huffman, devolver el código de Huffman asociado.
3. Dada una palabra del alfabeto (una lista de elementos) y árbol de Huffman, devolver la palabra codificada.
4. Dada una palabra codificada y un árbol de Huffman, devolver la palabra correspondiente del alfabeto.

9.18 (Teorema de Good, 1947)

1. Si a cada vértice V de un digrafo le hacemos corresponder dos índices:

$$\begin{aligned} \text{entranEn } V &\equiv \text{Número de arcos con destino } V \\ \text{salenDe } V &\equiv \text{Número de arcos con origen } V \end{aligned}$$

se dice que un digrafo es balanceado si, para cada vértice V ,

$$\text{entranEn } V \equiv \text{salenDe } V$$

Escriba un programa para comprobar si un digrafo es balanceado.

2. Se dice que un digrafo no tiene vértices aislados si cada uno de los vértices tiene al menos un arco que entra o sale de él. Escriba un programa para comprobar si un digrafo no tiene vértices aislados.

9.6 - Ejercicios

3. Si a es un arco en un digrafo, denotemos con $\text{org } a$, $\text{dest } a$ a los vértices origen y destino del arco a ; diremos que $[a_1, \dots, a_n]$ es un camino orientado de longitud n si:

$$\forall i, 1 \leq i \leq n-1, \text{dest } a_i = \text{org } a_{i+1}$$

Escriba un programa para comprobar si cierta lista de arcos es un camino orientado.

4. Un ciclo euleriano (circuito euleriano) es un camino orientado que contiene (sin repetir) todos los arcos del grafo y además $\text{dest } a_n = \text{org } a_1$. Escriba un programa para comprobar si cierta lista de arcos es un ciclo euleriano.
5. Se dice que un digrafo es conexo cuando para cualquier par (V, W) de vértices distintos existe un camino orientado de V a W . Escriba un programa para comprobar si un digrafo es conexo.
6. Un digrafo se dice que es euleriano si en él es posible encontrar un ciclo euleriano. Escriba un programa para comprobar si un digrafo es euleriano.
7. Escriba un programa para comprobar el teorema de Good:

Un digrafo sin vértices aislados es euleriano si y sólo si es balanceado y conexo.

9.19 (Teorema de Euler-Hierholzer, 1873) Cuando en un grafo no importa la dirección de sus arcos, dicho grafo se llama no dirigido; al igual que en el Ejercicio 9.18, consideraremos un grafo no dirigido como una lista de vértices y una función sucesor:

$$\text{data Grafo } a = G[a] (a \rightarrow [a])$$

de forma que (V, W) es un arco entre los vértices V y W de un grafo no dirigido si y sólo si $W \in \text{suc } V$, y $V \in \text{suc } W$.

1. Si a cada vértice V de un grafo no dirigido le hacemos corresponder un índice:

$$\text{gradoDe } V \equiv \text{Número de arcos en los que aparece } V$$

se dice que un grafo no dirigido es regular si todos sus vértices tienen el mismo grado. Escriba un programa para comprobar si un grafo no dirigido es regular.

2. Se dice que un grafo no dirigido no tiene vértices aislados si todos los vértices tienen grado mayor que 0. Escriba un programa para comprobar si un grafo no dirigido no tiene vértices aislados.
3. Si a es un arco en un grafo no dirigido, denotemos con $\text{org } a$, $\text{dest } a$ a los vértices que conecta el arco a ; diremos que $[a_1, \dots, a_n]$ es un camino de longitud n si:

$$\forall i, 1 \leq i \leq n-1, \text{dest } a_i = \text{org } a_{i+1}$$

sobrentendiendo que el orden del arco (*org a, dest a*) no es relevante salvo en los extremos del camino. Escriba un programa para comprobar si cierta lista de arcos es un camino.

4. Un ciclo euleriano (circuito euleriano) es un camino que contiene (sin repetir) todos los arcos del grafo y además $\text{dest } a_n = \text{org } a_1$; un camino euleriano es un camino que contiene (sin repetir) todos los arcos del grafo y además $\text{dest } a_n \neq \text{org } a_1$. Escriba programas para comprobar si cierta lista de arcos es un ciclo euleriano y si es un camino euleriano.
5. Se dice que un grafo no dirigido es conexo cuando para cualquier par (V, W) de vértices distintos existe un camino de V a W . Escriba un programa para comprobar si un grafo no dirigido es conexo.
6. Un grafo no dirigido se dice que es euleriano si en él es posible encontrar un ciclo euleriano, y se dice que es semieuleriano si en él es posible encontrar un camino euleriano; si no se puede encontrar ni un ciclo euleriano ni un camino euleriano, el grafo no dirigido se dice que es no euleriano. Escriba programas para comprobar si un grafo no dirigido es euleriano, semieuleriano y no euleriano.
7. Escriba programas para comprobar los siguientes teoremas:
 - Un grafo no dirigido conexo es euleriano si y sólo si el número de vértices con grado impar es cero.
 - Un grafo no dirigido conexo es semieuleriano si y sólo si el número de vértices con grado impar es dos.
 - Un grafo no dirigido conexo es no euleriano si y sólo si el número de vértices con grado impar es mayor que dos.

10 PROGRAMACIÓN MODULAR Y TIPOS ABSTRACTOS DE DATOS

10.1. MÓDULOS

Cualquier proyecto de programación de cierta envergadura no puede ser realizado de forma monolítica. Generalmente, el sistema es descompuesto en varias partes que son desarrolladas independientemente (normalmente por distintos programadores). El programa final se obtendrá al ensamblar las distintas partes. Cada una de estas partes es lo que se denomina un *módulo*.

El lenguaje HASKELL permite descomponer un programa en módulos. Cada módulo puede ser compilado y probado por separado, lo que permite que distintos programadores trabajen simultáneamente en las distintas partes de un programa.

Cada módulo debe definir un conjunto de declaraciones relacionadas entre sí. Estas declaraciones pueden ser utilizadas desde otros módulos, de modo que es posible ampliar el lenguaje mediante las denominadas *bibliotecas*. Podemos ver un módulo como una entidad que proporciona una serie de servicios que pueden ser utilizados desde otros módulos. Esta característica proporciona *abstracción* al lenguaje de programación, ya que un programador puede utilizar otras definiciones sabiendo tan sólo lo que hacen sin preocuparse de cómo están implementadas.

10.2. BIBLIOTECAS ESTANDARIZADAS

El lenguaje HASKELL define una serie de módulos que deben ser proporcionados por cualquier implementación de éste. La librería estandarizada de HASKELL 98 [Peyton Jones y Hughes, 2002] proporciona la descripción de estos módulos. La utilidad de los distintos módulos es muy diversa, desde definir datos numéricos racionales y complejos, pasando por funciones para la generación de números aleatorios, hasta una biblioteca de funciones para el manejo de listas.

Un módulo especial es PRELUDE. Este módulo incluye las declaraciones predefinidas del lenguaje que pueden ser utilizadas directamente desde cualquier otro módulo o programa.

10.3. DECLARACIONES DE MÓDULOS

Un módulo es un conjunto de declaraciones. Por ejemplo:

```

module Cadenas (rellena, aMayúsculas) where
  -- Añade espacios a la izquierda
  rellena    :: Int → String → String
  rellena n s = espacios (n - length s) ++ s

  -- Pasa a mayúsculas
  aMayúsculas :: String → String
  aMayúsculas = map toUpper

  espacios :: Int → String
  espacios n = replicate n ' '

```

En HASKELL un módulo comienza por la palabra `module` seguida del nombre de éste. Los nombres de módulo deben empezar por una letra mayúscula. Tras el nombre del módulo aparece una *lista de exportación* entre paréntesis. Las entidades que aparezcan en esta lista pueden ser utilizadas desde otros módulos; las que no aparezcan se denominan entidades *privadas* del módulo y sólo pueden ser usadas dentro del módulo donde se definen. Por ejemplo, la función `espacios` es privada al módulo `Cadenas`. Si la lista de exportación no aparece tras el nombre del módulo, se considera que todas las definiciones del módulo son exportadas. A esto se denomina *exportación implícita* y es útil cuando no queremos definir ninguna entidad privada para el módulo.

Las entidades exportadas deben estar definidas en el módulo o deben haber sido importadas de otro módulo. Es posible exportar funciones, operadores, definiciones de tipos y clases. Todas las instancias realizadas en un módulo son exportadas implícitamente. Al exportar una clase, hay que especificar qué métodos de la clase se exportan. Esto se consigue escribiendo la lista correspondiente de métodos entre paréntesis tras el nombre de la clase en la lista de exportación. Por ejemplo, el módulo:

```

module Áreas (Área, TieneÁrea(área)) where
  type Área = Float

  class TieneÁrea a where
    área :: a → Área

```

exporta el método `área` de la clase `TieneÁrea` y el tipo `Área`. Los constructores de un tipo de datos también pueden ser exportados selectivamente. Para ello, se escribe la lista de los constructores a exportar tras el nombre del tipo entre paréntesis. Por ejemplo, el siguiente módulo:

```

module Triángulos (Lado,
                    Triángulo(Equilátero, Isósceles, Escaleno)) where
  type Lado = Float

```

```

data Triángulo = Equilátero Lado
                | Isósceles Lado Lado
                | Escaleno Lado Lado Lado
                deriving Show

```

exporta los tres constructores del tipo `Triángulo`. Para exportar todos los métodos de una clase o todos los constructores de un tipo se puede sustituir la lista de elementos por dos puntos entre paréntesis. Así, la cabecera del módulo anterior puede abreviarse como:

```
module Triángulos (Lado, Triángulo(..)) where
```

Por último, un módulo puede exportar todas sus definiciones incluyendo el nombre del módulo precedido de la palabra `module` en la lista de exportación:

```
module Triángulos (module Triángulos) where
```

De este modo se pueden exportar todas las definiciones importadas desde otro módulo, creándose uniones de módulos.

Es necesario escribir el código correspondiente a cada módulo en un fichero de texto cuyo nombre coincida con el del módulo. Por ejemplo, el código correspondiente al módulo anterior debe ser almacenado en el fichero de texto "Triángulos.hs".

10.4. IMPORTACIÓN

Para poder utilizar los servicios proporcionados por un módulo es necesario *importarlos* (la única excepción a esta regla es el uso del módulo PRELUDE que es importado automáticamente en cualquier programa). Un módulo puede importar los servicios proporcionados por otro módulo utilizando la cláusula `import`. Por ejemplo:

```

module Ejemplo where
  import Cadenas

  -- Añade espacios a la derecha
  rellenaDer :: Int → String → String
  rellenaDer n = reverse . rellena n . reverse

```

La cláusula `import` debe aparecer antes de cualquier declaración (al inicio del módulo tras la cabecera). Una declaración de importación como la anterior importa todos los elementos exportados por el módulo `Cadenas`. A veces interesa importar sólo algunos. Esto puede ser interesante si en varios de los módulos aparecen identificadores repetidos y solo necesitamos uno de ellos. Para ello, hay que añadir una *lista de importación* tras el nombre del módulo. Por ejemplo,

```
import Cadenas (rellena)
```

importa del módulo `Cadenas` tan sólo la función `rellena`. La sintaxis de la lista de importación es similar a la de la lista de exportación (aunque no se puede usar la forma `module NombreMódulo` en la lista de importación). Como es de esperar, si se intenta

importar un elemento no exportado se produce un error. Del mismo modo, si se intenta utilizar una función no importada o definida se produce el correspondiente error.

Cuando se desea importar casi todas las definiciones de un módulo es más cómodo especificar cuáles no se quieren importar que listar todas las que se van a importar. Para esto se puede usar la palabra *hiding* antes de la lista de importación:

```
import Cadenas hiding (rellena)
```

Esta declaración importa todas las entidades exportadas por el módulo *Cadenas* excepto *rellena*.

Cuando no se indica ningún nombre de módulo, es igual que si se hubiera escrito:

```
module Main where
  import Prelude
```

Nota 10.1 En HUGS, el símbolo de invitación a la entrada de datos refleja el nombre del módulo con el que se está trabajando. Cuando no hay ningún módulo cargado, este símbolo es PRELUDE; cuando el usuario carga funciones sin definir ningún nombre de módulo el símbolo es MAIN, y cuando el módulo tiene un nombre, refleja el nombre de ese módulo.

10.4.1. IMPORTACIÓN CUALIFICADA

A veces es necesario importar varias entidades con el mismo nombre definidas en módulos distintos. En ese caso, existiría un problema de ambigüedad, ya que al usar la entidad dentro del módulo que importe, el compilador no podría determinar cuál de las dos versiones se desea utilizar. Para resolver este tipo de problemas es posible realizar *importaciones cualificadas*. Por ejemplo, el módulo desarrollado al comienzo de esta sección puede ser también escrito como¹:

```
module Ejemplo where
  import qualified Cadenas
  -- Añade espacios a la derecha
  rellenaDer :: Int → String → String
  rellenaDer n = reverse . Cadenas.rellena n . reverse
```

La declaración importa el módulo *Cadenas* de un modo cualificado. Para poder usar una entidad importada de este modo es necesario preceder el nombre de la entidad con el nombre del módulo del que procede y un punto. No puede existir ningún espacio antes o después del punto, ya que podrían producirse confusiones con el operador de composición de funciones. En la función *rellenaDer* el primer y tercer punto denotan composiciones de funciones, mientras que el segundo se usa para cualificar la función *rellena*.

¹ Obsérvese el espacio que se deja delante y detrás del operador de composición y la ausencia de espacio en el símbolo de importación cualificada.

Al realizar una importación cualificada es posible renombrar localmente el módulo importado en el módulo que importa. Esto suele usarse para abbreviar el nombre del módulo importado a la hora de usar nombres cualificados. Por ejemplo,

```
module Ejemplo where
  import qualified Cadenas as C
  -- Añade espacios a la derecha
  rellenaDer :: Int → String → String
  rellenaDer n = reverse . C.rellena n . reverse
```

asigna el nombre *C* al módulo *Cadenas* dentro del módulo *Ejemplo*.

10.5. TIPOS ABSTRACTOS DE DATOS

El mecanismo de ocultación que proporcionan los módulos de HASKELL es suficiente para la construcción de tipos abstractos de datos. Un tipo abstracto de datos (TAD) oculta al usuario los detalles de la implementación mostrando sólo la interfaz de acceso a los datos del tipo, también llamada *signatura*. De esta forma, la implementación puede ser alterada sin que un módulo cliente se vea afectado.

La representación de tipos abstractos de datos en HASKELL puede tener distintas aproximaciones. Aquí mostraremos dos de ellas; la primera donde las funciones de la interfaz no pueden sobrecargarse, y la segunda en la que sí, aunque veremos que ello presenta un problema en un sistema de clases uniparamétricas como es actualmente el de HASKELL.

10.6. REPRESENTACIÓN

Para la representación de un tipo abstracto de datos necesitamos definir una estructura para los datos y definir las funciones de acceso a esos datos. La definición de datos puede hacerse bien con un sinónimo de tipo (*type*), bien con un constructor de tipo (*data*), aunque con la primera no se define realmente un tipo nuevo. Las diferencias entre ellas quedan patentes cuando por cualquier requerimiento se necesita construir instancias de alguna clase para ese tipo (como *Eq*, *Ord* o *Show*) ya que HASKELL no permite construir instancias para sinónimos de tipo.

Para ocultar los detalles de implementación, la definición del tipo abstracto de datos se realizará dentro de un módulo, exportando exclusivamente las funciones de acceso al tipo junto al nombre del tipo (pero no su definición). Esto es una facilidad que proporcionan los módulos de HASKELL.

10.6.1. REPRESENTACIÓN CON UNA INTERFAZ NO SOBRECARGADA

En este caso las funciones que definen la interfaz no estarán sobrecargadas. Esto quiere decir que dos implementaciones de un mismo tipo abstracto de datos no pueden convivir a menos que la importación se haga de forma cualificada.

Introduciremos esta aproximación a través de un ejemplo. Vamos a construir un módulo que implemente un tipo abstracto de datos cola (*Cola*). Queremos disponer de funciones para crear una cola (*colaVacía*), insertar un elemento en una cola