Programación Funcional

Mario Rodríguez Artalejo

Universidad Complutense de Madrid

Enero 2004

Índice General

Capítulo 1

Conceptos fundamentales

1.1 Estilos de programación: El estilo funcional

Variedad de estilos de programación

Los muchos lenguajes de programación existentes se pueden clasificar según el *estilo de programación* que soportan. Los principales estilos (también llamados *paradigmas*) de programación son el *imperativo*, el *orientado a objetos* y el *declarativo*. Generalmente, el estilo orientado a objetos aparece en lenguajes de base imperativa, enriquecidos con las nociones de *objeto*, *clase* y *herencia*.

El estilo declarativo es muy diferente del imperativo, y bastante próximo al nivel de abstracción de los *lenguajes de especificación*. Incluye los lenguajes de *programación funcional*, objeto de este curso, y los lenguajes de *programación lógica*. En esta sección describimos brevemente las principales características, ventajas e inconvenientes de la programación funcional, en comparación con el estilo imperativo.

Lenguajes de programación imperativos

Pascal, Ada, C, C++, Java y muchos otros lenguajes de programación son *imperativos*. Un programa consiste en una secuencia de *órdenes* que se ejecutan una tras otra, modificando los valores de variables por medio de *asignaciones*. Por ejemplo, la asignación `x := x+1` incrementa en 1 el valor de la variable entera `x`.

Funciones

El concepto de *función* viene de las matemáticas. Una función f es una transformación que se puede aplicar a unos datos iniciales $x_1, \cdots, x_n$ llamados *argumentos* o *parámetros*, y devuelve un dato final $y = f(x_1, \cdots, x_n)$

llamado *resultado*. La mayoría de los lenguajes de programación disponen de funciones predefinidas para realizar diversas operaciones matemáticas (aritméticas, trigonométricas, etc.), así como de recursos que permiten a los usuarios definir otras funciones para propósitos específicos. Por ejemplo, la siguiente función escrita en Pascal calcula la suma de los cuadrados de los números enteros positivos comprendidos entre 1 y `n`:

```
function sumCuad(n : integer) : integer;
begin
   var s,i : integer;
   s := 0;
   i := 0;
   while i < n do
      begin
         i := i+1;
         s := i*i+s
      end;
   sumCuad := s
end
```

Naturalmente, las funciones utilizadas en programación no siempre operan con objetos matemáticos. Por ejemplo, para una aplicación de procesamiento de textos se podría diseñar una función de formateo. Sus parámetros serían un texto sin formatear junto con información acerca de cómo procesarlo, mientras que su resultado sería un texto formateado.

Lenguajes de programación funcionales

Los programas imperativos contienen asignaciones y otras construcciones que controlan el orden de ejecución, tales como *bucles* y llamadas a *procedimientos*. En un lenguaje funcional, un programa contiene únicamente definiciones de funciones. Concretamente, en el lenguaje funcional Haskell las funciones se definen por medio de *ecuaciones*, declarando los *tipos* de sus parámetros y de su resultado. Por ejemplo, las siguientes funciones sirven para calcular el cuadrado de un número entero y el máximo entre dos números enteros, respectivamente. Obsérvese que cada una de las dos ecuaciones de `max` tiene asociada una condición.

```
cuadrado   :: Int -> Int
cuadrado n  = n*n

max           :: Int -> Int -> Int
max n m
  | n >= m     = n
  | n  < m     = m
```

Una definición de la función `sumCuad` se puede escribir en Haskell como sigue. Observemos que la definición es *recursiva*; las condiciones asociadas a las dos ecuaciones distinguen los casos directo y recursivo.

```
sumCuad    :: Int -> Int
sumCuad n
  | n == 0  = 0
  | n  > 0  = cuadrado n + sumCuad (n-1)
```

Vemos también que la definición de `sumCuad` usa la función `cuadrado` definida previamente. En general, la definición de cualquier función puede usar en sus ecuaciones otras funciones definidas en el mismo programa.

En cierto modo, un programa funcional se comporta como una calculadora muy potente que puede usar funciones complejas definidas por el programador. Un cómputo siempre consiste en la *evaluación* de una expresión que usa las funciones definidas en el programa. Usando el intérprete Hugs 98 para el lenguaje Haskell, se pueden evaluar expresiones en las que intervengan las funciones `cuadrado`, `max` y `sumCuad`:

```
> cuadrado (2+3)
25
> cuadrado (max 2 3)
9
> sumCuad (max 2 3)
14
```

Para realizar las pruebas anteriores, es necesario previamente cargar un programa cuyo texto contenga las definiciones de las funciones `cuadrado` y `sumCuad`. No es necesario incluir `max`, ya que se trata de una función predefinida. De hecho, el intento de redefinir `max` dentro de un programa causa un mensaje de error de Hugs 98.

Características de los lenguajes funcionales

Las funciones definidas en un lenguaje imperativo pueden causar *efectos colaterales*. Por ejemplo, la siguiente función Pascal incrementa en `1` el valor de la variable global `m` cada vez que es ejecutada:

```
function siguiente(n : integer) : integer;
begin
   m := m+1;
   siguiente := n+1
end
```

Los efectos colaterales hacen más impredecible el comportamiento de los programas. Por ejemplo, los dos fragmentos de código Pascal que se muestran más abajo parecen equivalentes a primera vista. Sin embargo,

debido al efecto sobre `m` causado por la evaluación de `siguiente(x)`, el valor final de la variable `y` es `1` tras la ejecución del primer fragmento de código, y `2` tras la ejecución del segundo fragmento.

```
m := 0;                          m := 0;
x := 0;                          x := 0;
y := m+siguiente(x)              y := siguiente(x)+m
```

En *lenguajes funcionales puros* como Haskell, no existe la asignación destructiva. Por este motivo, las variables se comportan como en matemáticas. No cambian de valor, y el resultado de evaluar una expresión cualquiera siempre es el mismo, independientemente del orden de evaluación. Otras características importantes de los lenguajes funcionales puros son:

- **Nivel de abstracción alto:**
  Los programas están próximos a la lógica de los problemas a resolver, ignorando los detalles de bajo nivel. Las ecuaciones de un programa sirven para calcular y también para razonar.

- **Disciplina estática de tipos:**
  Los errores debidos a tipos incorrectos se detectan antes de la ejecución.

- **Polimorfismo:**
  Se pueden programar funciones que operan para datos de cualquier tipo. Por ejemplo, la función que calcula la longitud de una lista no depende del tipo de los elementos.

- **Funciones de orden superior:**
  Se permite que los parámetros y/o el resultado de una función f sean a su vez funciones. En este caso, se dice que f es una función de orden superior.

- **Evaluación perezosa:**
  Al calcular el valor de una expresión, se evitan automáticamente todos los cálculos innecesarios.

- **Gestión automática de la memoria:**
  La memoria que se va necesitando para el cómputo es asignada automáticamente en tiempo de ejecución. Un sistema de recogida de basura (*garbage collection*) recupera automáticamente la memoria asignada que deja de ser necesaria.

Ventajas de la programación funcional

Algunas ventajas importantes de la programación funcional son:

- **Brevedad y claridad:**
  Para un mismo problema, los programas funcionales tienden a ser mucho más cortos que los programas imperativos equivalentes. En general, también resultan más fáciles de entender. Como consecuencia, aumenta la productividad y se facilita el mantenimiento.

- **Seguridad:**
  La corrección de los programas funcionales es más fácil de verificar que en el caso imperativo. La transformación de programas en otros equivalentes y más eficientes también es más sencilla. En general, la aplicación de métodos formales para razonar acerca del comportamiento de los programas se facilita.

- **Reusabilidad:**
  Gracias al polimorfismo y a las funciones de orden superior, los lenguajes funcionales permiten escribir programas muy generales que se pueden reutilizar en diferentes aplicaciones.

- **Composicionalidad:**
  La evaluación perezosa y las funciones de orden superior ayudan a utilizar metódicamente funciones ya definidas como piezas para el diseño de otras funciones.

Inconvenientes de la programación funcional

La programación funcional también tiene inconvenientes; véase [11]. Algunos de ellos son:

- **Exotismo:**
  El estilo funcional es menos conocido que el imperativo. Hace falta un esfuerzo para adaptarse a una nueva manera de pensar.

- **Ineficiencia:**
  Los programas funcionales no permiten controlar detalles de bajo nivel. En general, su ejecuación es más lenta y consume más memoria que en el caso imperativo.

- **Comportamiento complejo:**
  La evaluación perezosa efectúa los cálculos en un orden difícil de predecir. Esto no dificulta la programación en sí misma, pero sí dificulta el análisis de la eficiencia de los programas. A veces es difícil estimar el tiempo y el espacio requeridos para el cómputo.

- **Falta de apoyo a los usuarios:**
  Las implementaciones existentes de lenguajes funcionales en general carecen de un entorno de programación que facilite el desarrollo y

mantenimiento de programas. En particular, faltan herramientas que soporten la depuración, la evaluación del rendimiento y la transformación de programas.

Aplicaciones de la programación funcional

Aunque la programación funcional no es tan popular como otros estilos de programación, la experiencia ha demostrado su utilidad para diversas aplicaciones. El uso de lenguajes de la familia del LISP es tradicional en aplicaciones relacionadas con la Inteligencia Artificial. Existen además otras experiencias industriales de uso de lenguajes funcionales, tanto para el desarrollo rápido de prototipos como para el mantenimiento de aplicaciones estables. Muchos lenguajes funcionales, en particular Haskell, están diseñados para soportar tanto el cálculo simbólico como el numérico, y disponen de mecanismos para comunicarse con programas escritos en lenguajes imperativos tales como C y C++. En `http://www.haskell.org/practice.html` y en el artículo [11] se encuentra información sobre algunas aplicaciones informáticas relevantes, desarrolladas con lenguajes funcionales.

Familias de lenguajes funcionales

La programación funcional empezó a investigarse en la década de 1960. Los principales lenguajes funcionales existentes hoy en día se pueden agrupar en tres familias:

- **Familia LISP:**
  Los lenguajes de esta familia admiten asignación y no tienen disciplina de tipos.
- **Familia ML:**
  Los lenguajes de esta familia admiten asignación. Tienen disciplina de tipos y polimorfismo, pero no usan evaluación perezosa.
- **Familia Haskell:**
  Los lenguajes de esta familia son *puros*, es decir, no admiten asignación destructiva. Además, tienen disciplina de tipos, polimorfismo y evaluación perezosa.

Haskell 98 y Hugs 98

El lenguaje funcional de referencia para este curso es Haskell. Usaremos la sintaxis de Haskell para escribir nuestros programas, y seguiremos un enfoque muy similar al del texto [1]. Existen diferentes implementaciones de Haskell disponibles en el dominio público. La más recomendable para principiantes es el intérprete Hugs 98 de la versión Haskell 98 de Haskell. En

la página web de Haskell (`http://www.haskell.org/`) se puede encontrar información y documentación sobre Haskell 98 y Hugs 98. En particular, allí se encuentran los archivos necesarios para instalar Hugs 98 en diferentes plataformas: Unix, Linux, Windows y Macintosh.

De entre los documentos que se pueden descargar de la página de Haskell, destacamos dos cuya consulta es imprescindible para la realización de prácticas personales:

- *A Gentle Introduction to Haskell 98* [5]: Documento quue explica de forma abreviada las principales características del lenguaje Haskell 98.
- *Hugs 98, A functional programming system based on Haskell 98, User Manual* [6]: Documento que sirve de manual para la instalación y utilización de Hugs 98.

1.2 Evaluación de expresiones

Cálculo del valor de expresiones

Para calcular el valor de una expresión e, se van aplicando las ecuaciones del programa, orientadas de izquierda a derecha. Este proceso se llama *reducción* o *reescritura*. En un paso de reescritura que use la ecuación orientada $l = r$, se localiza una parte l' de la expresión e con la misma forma que l y se reemplaza por otra expresión r' con la misma forma que r. Más exactamente, la parte l' de e (llamada *redex*) debe ser el resultado de aplicar una sustitución adecuada a las variables de l, y r' debe ser el resultado de aplicar esta misma sustitución a las variables de r. La sustitución representa el reemplazamiento de parámetros formales por parámetros actuales.

A efectos de la evaluación de expresiones, admitiremos que el programa contiene implícitamente todas las ecuaciones necesarias para determinar el comportamiento de las funciones predefinidas. Usaremos la notación $e \longrightarrow e'$ para indicar que e se reduce a e' mediante un paso de reescritura. Por ejemplo, suponiendo que la función `cuadrado` esté definida por la ecuación indicada en la sección 1.1, hay dos pasos de reescritura diferentes que reducen la expresión `cuadrado (2+3)`:

1. `cuadrado (2+3)` $\longrightarrow$ `cuadrado 5`
2. `cuadrado (2+3)` $\longrightarrow$ `(2+3)*(2+3)`

El paso de reescritura 1. usa la ecuación `2+3 = 5`, incluida implícitamente en el programa como parte de la definición de la función predefinida de suma de enteros. En cambio, el paso 2. usa la única ecuación incluida en la definición explícita de la función `cuadrado`. En ambos pasos, el redex reducido se ha subrayado.

En general, para evaluar una expresión e no basta con un solo paso de reescritura, sino que hay que reiterar el proceso de reducción hasta alcanzar una expresión que ya no se pueda reducir más, la cual se llama *forma normal* de la expresión inicial. La serie de pasos que conduce desde una expresión inicial hasta su forma normal se llama *secuencia de reducción*. Puesto que cada paso de reescritura se puede elegir de varias maneras, generalmente hay varias secuencias de reducción a partir de una misma expresión inicial. Como ejemplo, veamos dos secuencias de reducción diferentes que transforman la expresión `cuadrado (2+3)` a forma normal. Para mayor claridad, el redex reducido en cada paso de reescritura aparece subrayado.

```
    cuadrado (2+3)
⟶   cuadrado 5
⟶   5*5
⟶   25
```

```
    cuadrado(2+3)
⟶   (2+3)*(2+3)
⟶   5*(2+3)
⟶   5*5
⟶   25
```

En este ejemplo, intérprete Hugs 98 ejecutaría un cómputo correspondiente a la segunda de las dos secuencias de reducción anteriores, y mostraría la forma normal obtenida como el *valor* calculado para la expresión inicial:

```
> cuadrado (2+3)
25
```

En lo sucesivo utilizaremos la notación $e \longrightarrow^* e'$ para indicar que e se reduce a e' a través de un cierto número $n \geq 0$ de pasos de reducción. En cómputos que utilicen ecuaciones con condiciones, hay que tener en cuenta que una ecuación con condición asociada `c` solo se puede aplicar calculando previamente una secuencia de reducción `c` $\longrightarrow^*$ `True` para comprobar que la condición `c` se cumple. Por ejemplo, el paso de reescritura `max 7 3` $\longrightarrow$ `7` utiliza la primera ecuación de la función `max` (definida en la sección 1.1) con la sustitución $\{$`n` $\mapsto$ `7, m` $\mapsto$ `3`$\}$. La condición asociada a la ecuación, afectada de esta misma sustitución, es `7 >= 3`. Por tanto, es necesario ejecutar el cómputo auxiliar `7 >= 3` $\longrightarrow^*$ `True`, para comprobar que la ecuación puede ser aplicada en este caso.

Estrategias de reducción: evaluación impaciente y perezosa

Al no ser única la secuencia de reducción que se puede producir a partir de una expresión inicial dada, las diferentes implementaciones de lenguajes funcionales utilizan distintos criterios para restringir el cómputo a una sola posibilidad. Cada criterio bien definido para escoger solamente *uno* de los redex posibles en cada paso de reescritura, se llama *estrategia de reducción*. Las dos estrategias de reducción más importantes son:

- *Evaluación impaciente*: consiste en elegir en cada paso un redex l' de la expresión de cómputo e lo más interno posible, es decir, tal que ninguna expresión contenida estrictamente en l' sea otro redex.

- *Evaluación perezosa*: consiste en elegir en cada paso un redex l' de la expresión de cómputo e lo más externo posible, es decir, tal que ninguna parte de e que contenga estrictamente a l' sea otro redex.

Volviendo a las dos secuencias de reducción mostradas más atrás, vemos que la primera representa a un cómputo realizado con evaluación impaciente, mientras que la segunda es un cómputo realizado con evaluación perezosa. El mismo ejemplo ilustra lo siguiente:

- La evaluación impaciente solo selecciona un redex `(f e1 ... en)` cuando todos los parámetros actuales `ei` de la función `f` han sido evaluados previamente a forma normal. Esto corresponde al mecanismo llamado *paso de parámetros por valor*.

- La evaluación perezosa puede seleccionar un redex `(f e1 ... en)` aunque los parámetros actuales `ei` de la función `f` no hayan han sido evaluados todavía a forma normal. Esto corresponde al mecanismo llamado *paso de parámetros por nombre*.

Los lenguajes de la familia Haskell siempre se implementan usando *evaluación perezosa*, mientras que los lenguajes de las familias LISP y ML utilizan evaluación impaciente. El siguiente resultado, que aceptamos sin demostración, permite comparar el comportamiento de estas dos estrategias:

Propiedades fundamentales de las estrategias de reducción: Suponiendo fijados un programa funcional y una expresión de coómputo inicial e, se verifica:

(a) Todas las secuencias de reducción que comiencen con e y terminen, acaban en la misma forma normal e'. Es decir: la forma normal, en caso de existir, es única e independiente de la estrategia de reducción empleada.

(b) Siempre que e admita forma normal, la evaluación perezosa es capaz de calcularla. En algunos casos esto no es cierto para la evaluación impaciente.

Las ventajas de la evaluación perezosa frente a la impaciente se deben sobre todo al apartado (b) del resultado anterior. Para ver un ejemplo concreto, supongamos definida la función

```
primero     :: Int -> Int -> Int
primero x y  = x
```

y consideremos la evaluación de la expresión `primero (1+1) (1 ‘div‘ 0)`. La evaluación impaciente produce un cómputo que termina en error sin calcular una forma normal:

```
    primero (1+1) (1 ‘div‘ 0)
→   primero 2 (1 ‘div‘ 0)
```

El error es debido al intento de evaluar la expresión `(1 ‘div‘ 0)`. En cambio, la evaluacción perezosa calcula con éxito la forma normal `2`:

```
    primero (1+1) (1 ‘div‘ 0)
→   (1+1)
→   2
```

Obsérvese que la evaluación perezosa no intenta en ningún momento evaluar la subexpresión `(1 ‘div‘ 0)`, cuyo valor no es necesario para el cómputo. En general, la evaluación perezosa siempre evita la reducción de subexpresiones cuyo valor no es necesario. La evaluación impaciente siempre intenta reducirlas, causando (según los casos) trabajo innecesario, erores, o incluso no terminación. Para ilustrar esta última posibilidad, consideremos la siguiente definición de una *constante* (i.e., función sin parámetros) de tipo entero:

```
infinito :: Int
infinito  = 1 + infinito
```

La reducción de la expresión `primero (1+1) infinito` a forma normal con evaluación perezosa sigue siendo posible:

```
    primero (1+1) infinito
→   (1+1)
→   2
```

En cambio, el uso de evaluación impaciente conduce a una secuencia de reducción no terminante:

```
    primero (1+1) infinito
→   primero 2 infinito
→   primero 2 infinito
    ...
```

Un inconveniente de la evaluación perezosa, según la hemos explicado hasta ahora, es que en algunos casos puede requerir más pasos de reducción que la evaluación impaciente. Como ejemplo de esta situación sirve la evaluación de la expresión `cuadrado (2+3)`, mostrada nateriormente. En la práctica, este inconveniente se puede evitar utilizando una representación de la expresión de cómputo que identifique partes repetidas de la misma, evitando así que sean evaluadas varias veces. Las implementaciones de la

evaluación perezosa siempre emplean esta técnica, denominada *estructura compartida*. Por ejemplo, la evaluación perezosa de `cuadrado (2+3)` con estructura compartida produce un cómputo con el mismo número pasos de reducción que la evaluación impaciente:

```
    cuadrado (2+3)
→   (2+3)*(2+3)
→   5*5
→   25
```

De hecho, la evaluación perezosa se implementa representando la expresión `(2+3)*(2+3)` como un *grafo*, de tal manera que la estructura de `(2*3)` ocupa espacio una sola vez.

El valor indefinido

En ocasiones, puede ocurrir que una forma normal no represente un valor significativo, como ocurre por ejemplo con la expresión `(1 ‘div‘ 0)`. En estos casos, se produce un error de cómputo, y se dice que el valor calculado está indefinido. En otros casos, puede ocurrir que la secuencia de reducción no termine, ni siquiera empleando evaluación perezosa, y que el cómputo no produzca ninguna información significativa. En estas situaciones se considera también que el valor calculado es indefinido.

El símbolo $\perp$ (pronunciado *fondo*) se utiliza para nombrar el valor indefinido a efectos de razonamientos. Pero **OJO**: el símbolo $\perp$ nunca es mostrado por el intérprete como resultado de un cómputo real, ya que justamente representa la situación en la que el cómputo real no puede terminar con éxito. Sin embargo, sí es posible definir una constante `nada` cuyo valor sea $\perp$, poniendo:

```
nada :: a
nada  = error "indefinido!"
```

La definición anterior utiliza la función predefinida `error`, que espera como parámetro una cadena de caracteres y causa un error de cómputo (ésto es, un valor indefinido), mostrando además como mensaje la cadena pasada como parámetro. La declaración de tipo `nada :: a` indica que el valor de `nada` (que es $\perp$) pertenece a *cualquier tipo* `a`. Gracias a esto, `nada` se puede emplear como parámetro de cualquier función para probar ejemplos de cómputo, como veremos más abajo.

Funciones estrictas y no estrictas

En lenguajes que utilicen evaluación perezosa, puede ocurrir que el cálculo del valor de una función no necesite ninguna información acerca de su parámetro. Cuando esto sucede, se dice que la función es *no estricta*. Más exactamente: una función `f :: A -> R` se llama *estricta* si `(f` $\perp$`) =` $\perp$, y *no estricta* en caso contrario. Para una función `f` con varios parámetros, del tipo `f :: A1 -> ... -> An -> R`, se dice que `f` es *estricta con respecto al i-ésimo parámetro* ($1 \leq i \leq n$) si `(f e1 ...` $\perp$ `... en)` - con $\perp$ en el lugar del parámetro número i - da siempre resultado $\perp$, para cualquier elección posible de los restantes parámetros.

La idea es que una función estricta con respecto a su i-ésimo parámetro *siempre necesita* evaluar dicho parámetro, mientras que una función no estricta con respecto a su iésimo arámetro *no siempre necesita* evaluar dicho parámetro. Veamos algunos ejemplos concretos. En cada caso, presentamos como comprobación un cómputo ejecutable que se puede probar en Hugs 98.

1. La función `primero` definida más arriba es estricta con respecto a su primer parámetro, pero no con respecto al segundo. Comprobación:

```
> primero nada (1+1)
Program error: indefinido!
> primero (1+1) nada
2
```

2. La función `suma13` definida por

```
suma13       :: Int -> Int -> Int -> Int
suma13 x y z  = x+z
```

   es estricta con respecto a sus parámetros primero y tercero, pero no con respecto a su segundo parámetro. Comprobación:

```
> suma13 nada (2*3) (3*2)
Program error: indefinido!
> suma13 (2*3) nada (3*2)
12
> suma13 (2*3) (3*2) nada
Program error: indefinido!
```

3. La función `not` está predefinida como sigue:

```
not      :: Bool -> Bool
not True  = False
not False = True
```

Esta función es estricta, ya que sin reducir su parámetro a forma normal no se puede aplicar ninguna de las dos ecuaciones. Comprobación:

```
> not nada
Program error: indefinido!
```

Como vemos, la presencia de variables como parámetros formales en los lados izquierdos de las ecuaciones que definen una función puede indicar que la función no es estricta. La razón es que un parámetro formal de la forma `v` (variable) se puede reemplazar por *cualquier* parámetro actual sin necesidad de reducirlo. Sin embargo, este criterio no es suficiente en todos los casos, ya que el valor de una variable que aparezca como parámetro formal puede ser necesario para la evaluación del lado derecho de la ecuación correspondiente. Esto sucede para los parámetros `x`, `z` en la función `suma13`.

Razonamiento con ecuaciones

En los lenguajes funcionales puros, tales como Haskell, la evaluación de una expresión nunca produce efectos colaterales y da un resultado independiente del contexto en el que se use la expresión. Esta propiedad, llamada *transparencia referencial*, permite utilizar las ecuaciones de un programa funcional no solamente para *calcular*, como ya hemos visto, sino también para *razonar*. El cálculo manual sirve para *comprobar* el comportamiento del programa en casos particulares, mientras que el razonamiento sirve para *verificar* propiedades generales del comportamiento del programa.

Frecuentemente la propiedad a verificar toma la forma de una ecuación $e_1 = e_2$, y el razonamiento que verifica la validez de la ecuación se realiza construyendo dos secuencias de reducción que reducen los dos miembros de la ecuación a una expresión común e:

$$e_1 \longrightarrow^* e, \quad e_2 \longrightarrow^* e$$

Esta técnica de razonamiento es una forma de *cálculo simbólico*, ya que se produce sin necesidad de asignar valores concretos a las variables de las expresiones con las que se razona. En muchos casos, es necesario razonar combinando la ejecución simbólica con distinciones de casos. Supongamos por ejemplo que deseamos verificar la validez de la ecuación

```
(2*(max n m) >= n+m) = True
```

siendo `max` la función definida en la sección 1.1. Puesto que `True` está en forma normal, es suficiente demostrar que `(2*(max n m) >= n+m)` $\longrightarrow^*$ `True`. Esto se puede razonar distinguiendo dos casos:

Caso 1: `n >= m`

```
    2*(max n m) >= n+m
⟶   2*n >= n+m
⟶   n+n >= n+m
⟶   True
```

Caso 2: `n < m`

```
    2*(max n m) >= n+m
⟶   2*m >= n+m
⟶   m+m >= n+m
⟶   True
```

En ambos casos se llega a la propiedad deseada, y los dos casos cubren todas las posibilidades. Luego la propiedad queda demostrada.

En este y otros ejemplos, el cálculo simbólico puede utilizar propiedades conocidas de funciones predefinidas, tales como `2*n = n+n`. En otros ejemplos más complicados, el razonamiento con las ecuaciones de un programa debe combinar la ejecución simbólica con técnicas de inducción. Encontraremos ejemplos de esta técnica a lo largo de todo el curso, muy particularmente en el capítulo 3.

Ejercicios

1. Considera las funciones `cuadrado` y `max` estudiadas en la sección 1.1, junto con la función `cero` definida a continuación:

   ```
   cero   :: Int -> Int
   cero n  = 0
   ```

   Para cada una de las expresiones siguientes, estudia las secuencias de reducción correspondientes a las dos estrategias de evaluación impaciente y perezosa, respectivamente. Estudia en cada caso si la secuencia de reducción termina o no en una forma normal, e indica cual es el valor calculado.

   ```
   cuadrado (max 3 5)        cuadrado (max (5*2) (2+3))

   cero infinito             cuadrado infinito
   ```

   Sugerencias: Subraya el redex que se reduce en cada paso. En los pasos que usen ecuaciones con condiciones, incluye los cálculos necesarios para evaluar las condiciones.

2. La siguiente función booleana devuelve `True` si sus tres parámetros son todos iguales, y `False` en caso contrario. Su definición usa la función predefinida `&&`, que se escribe en notación infija y calcula la conjunción de valores booleanos.

```
tresIguales       :: Int -> Int -> Int -> Bool
tresIguales x y z  = (x == y) && (y == z)
```

Usando evaluación perezosa, calcula el valor de las expresiones siguientes:

```
tresIguales 3 7 7

tresIguales 4 4 4

tresIguales (max 2 5) 5 (max 4 3)
```

3. Se desea definir una función `tresDiferentes` con tres argumentos enteros, que calcule el resultado `True` si cada uno de sus tres argumentos es diferente de los otros dos, y el resultado `False` en caso contrario. La siguiente definición es incorrecta:

```
tresDiferentes       :: Int -> Int -> Int -> Bool
tresDiferentes x y z  = (x /= y) && (y /= z)
```

Comprueba la incorrección de la definición anterior calculando su resultado para un caso de prueba bien elegido. Escribe otra definición que te parezca correcta, y comprueba su corrección en algunos casos de prueba. Procura elegir casos representativos.

1.3 Definición de funciones

Ecuaciones condicionales

Como ya hemos visto en varios casos, la definición de una función en Haskell viene dada por la declaración de tipo y una o varias ecuaciones, opcionalmente condicionales. Por lo tanto, la definición de cualquier función `f` en un programa Haskell sigue un esquema como el que se indica más abajo, que corresponde a un grupo de varias *ecuaciones condicionales* con un mismo *lado izquierdo* `f p1 ... pn` (en el que aparecen los *parámetros formales* `pi`), *condiciones* `cj` (también llamadas *guardas*), y *lados derechos* `ej`. Está permitido omitir la declaración de tipo, pero es muy aconsejable incluirla.

```
f            :: A1 -> ... -> An -> R

 ...

f p1 ... pn
  | c1        = e1
  | c2        = e2
```

```
...

 | otherwise = ek

...
```

El número `n` de parámetros formales `pi` se llama `aridad` de la función `f`, y debe ser el mismo en todas las ecuaciones. Las condiciones deben ser expresiones de tipo booleano, y los lados derechos pueden ser expresiones cualesquiera, de tipo adecuado. *Siempre se exige que en un lado izquierdo no aparezcan variables repetidas.* Por ejemplo, la siguiente "definición" no es legal en un programa Haskell:

```
iguales x x = True
iguales x y = False
```

Al ejecutar una definición en la que aparezcan ecuaciones condicionales, las condiciones se evalúan en orden textual y se usa la primera ecuación cuya condición dé el valor `True`. La última condición `otherwise` es opcional y se interpreta como `True`. El caso en el que la condición `otherwise` no aparece está permitido. También se permiten ecuaciones sin condiciones, llamadas *ecuaciones incondicionales.*

La definición de la función `max` en la sección 1.1 ilustra el uso de ecuaciones condicionales sin condición `otherwise`. Otra definición equivalente que usa la condición `otherwise` es la siguiente:

```
max           :: Int -> Int -> Int
max n m
  |  n >= m    = n
  |  otherwise = m
```

La definición presentada en la sección 1.1 usa la condición `n < m` en lugar de `otherwise`, haciendo así más explícito el significado lógico de la distinción de casos.

Razonamiento con ecuaciones condicionales

El cálculo simbólico con las ecuaciones de un programa puede conducir a conclusiones incorrectas si se utilizan por separado las diferentes condiciones y se interpreta `otherwise` como `True`. Por ejemplo, utilizando la definición de `max` con `otherwise` se puede "demostrar" la ecuación `1 = 0`, razonando que

1. `max 1 0` $\longrightarrow$ `1`
   con la ecuación `max.1`, aplicable porque la condición `1 >= 0` se cumple.

2. `max 1 0` $\longrightarrow$ `0`
   con la ecuación `max.2`, aplicable porque la condición `otherwise` (equivalente a `True`) se cumple.

En la práctica, las secuencias de reducción incorrectas tales como la del item 2. del ejemplo anterior no son posibles. Esto es debido a que los intérpretes de Haskell evalúan los diferentes guardas en orden textual, hasta encontrar el primero que vale `True`.

Para evitar errores en razonamientos que utilicen un grupo de ecuaciones condicionales con un lado izquierdo común y condiciones `ci`, basta razonar reemplazando cada condición `ci` por la condición más compleja `(ci && not c1 && ... && not ci-1)`, que excluye el caso de las condiciones anteriores. En el ejemplo anterior, esta técnica conduce a razonar con la siguiente defínción de la función `max`:

```
max                              :: Int -> Int -> Int
max n m
  |  n >= m                       = n
  |  otherwise && not (n >= m)    = m
```

la cual es equivalente a la definición de la sección 1.1 y no da lugar a razonamientos incorrectos. En aquellos casos en que las condiciones `ci` sean mutuamente excluyentes, siempre es correcto razonar sin necesidad de modificarlas.

Regla de indentación

Cuando aparecen varias definiciones seguidas en el texto de un programa, es necesario determinar dónde termina cada una y comienza la siguiente. En Haskell, se utiliza un convenio basado en la *indentación* del texto: una definición termina en el primer punto posterior a su comienzo, donde el texto se sitúe en la misma columna que el comienzo de la definición o más a la izquierda.

Este convenio se explica en detalle en la sección 4.6 de [5]. Normalmente, el convenio de indentación permite presentar cómodamente las definiciones, escribiendo las ecuaciones condicionales según el esquema explicado más arriba. En caso de duda, es posible separar explícitamente las definiciones usando ";" como separador y las llaves "{" y "}" como paréntesis. En la práctica, este recurso casi nunca es necesario.

Sintaxis de los tipos

Al escribir las declaraciones de tipo de funciones en un programa Haskell, debe tenerse en cuenta que la sintaxis de los tipos es la expresada por las siguientes reglas BNF:

```
T ::=   a
      | TP
      | (T1, ..., Tn)
      | (T1 -> T)
```

El lenguaje Haskell permite aún otras posibilidades de formación de tipos, que estudiaremos en el capítulo 2. Sin embargo, para los propósitos de este capítulo bastan las reglas de formación de tipos indicadas más arriba, cuyo significado es:

- `a`
  Parámetro. Representa un tipo sin concretar. En la práctica, puede ser cualquier identificador que comience por minúscula.

- `TP`
  Tipo primitivo. Representa uno de los tipos predefinidos disponibles, tales como `Bool`, `Int`, etc. Debe ser un identificador que comience por mayúscula.

- `(T1, ..., Tn)`
  Producto de n tipos, $n \geq 0$. Es el tipo de las tuplas `(e1, ..., en)` formadas por `n` expresiones `ei` con tipos respectivos `Ti`. En el caso $n = 0$ se tiene el producto vacío `()`, que es el tipo de la tupla vacía `()`. En el caso $n = 1$, el producto unitario `(T)` se identifica con `T`.

- `(T1 -> T)`
  Tipo funcional. Representa el tipo de las funciones que esperan un parámetro de tipo `T1` y devuelven un resultado de tipo `T`.
  OJO: `T1` y `T` pueden ser a su vez tipos compuestos.

Al escribir tipos, se permite abreviar la sintaxis aceptando el convenio de que `->` asocia por la derecha y tiene precedencia sobre la coma. Este convenio permite eliminar muchos paréntesis sin peligro de ambigüedad. Por ejemplo:

- `Int -> Int -> Bool` es abreviatura de `(Int -> (Int -> Bool))`.

- `(Int -> Int) -> Bool` es abreviatura de `((Int -> Int) -> Bool)` y diferente de `Int -> Int -> Bool`.

- `(a -> Bool, Bool -> b)` es abreviatura de `((a -> Bool), (Bool -> b))`.

Sintaxis de las expresiones

Las expresiones utilizadas en programas Haskell deben escribirse en una sintaxis cuyas construcciones más comunes son las expresadas por las siguientes reglas de formación de expresiones, en notación BNF:

```
e ::=   x
      | (e e1)
      | (\x -> e)
      | (e1, ..., en)
      | if e then e1 else e2
      | let {def; ...; defn} in e
```

El significado de cada una de estas seis clases de expresiones es el siguiente:

- `x`
  Identificador. Según el contexto en el que se use, puede representar un parámetro formal, una constante, o el nombre de una función. Haskell exige que los identificadores comiencen por una letra minúscula, excepto en el caso de las constructoras de datos (véase el capítulo 2) que deben empezar por una letra mayúscula.

- `(e e1)`
  Aplicación de una expresión `e`, que hace el papel de función, a otra expresión `e1` que hace el papel de parámetro.
  OJO: `e` y `e1` pueden ser a su vez expresiones compuestas.

- `(\x -> e)`
  Función anónima con parámetro formal `x` y cuerpo `e`, también llamada λ-abstracción, o simplemente abstracción. El parámetro formal debe ser un identificador, mientras que el cuerpo puede ser cualquier expresión.

- `(e1, ..., en)`
  Tupla formada por n expresiones, $n \geq 0$. En el caso $n = 0$ se tiene la tupla vacía `()`. En el caso $n = 1$, la 1-tupla unitaria `(e)` se identifica con `e`.

- `if e then e1 else e2`
  Expresión condicional con condición `e` y ramas `e1` y `e2`.

- `let {def1; ...; defn} in e`
  Expresión `e` en el contexto de una serie de definiciones locales `defi`, que estarán disponibles a la hora de evaluar `e` (aparte de las definiciones globales incluidas en el programa).

En apartados posteriores de esta sección estudiaremos más en detalle el significado y modo de uso de las diferentes expresiones, en particular las definiciones locales y las funciones anónimas. En la práctica, la escritura de expresiones se puede abreviar utilizando los convenios siguientes:

- La pareja de paréntesis más externa se puede omitir.

- La operación de aplicación de una función a su parámetro asocia por la izquierda. En consecuencia, `(( ... ((f e1) e2) ... ) en)` se puede escribir abreviadamente en la forma `(f e1 ... en)`, donde aparecen n parámetros ei escritos uno tras otro detrás del nombre de una función. Si se la pareja de paréntesis más externa, es posible abreviar aún un poco más, escribiendo `f e1 ... en` sin paréntesis.

- La abstracción asocia por la derecha. Así, `(\x1 -> (\x2 -> (\x3 -> e)))` se puede abreviar como `\x1 -> \x2 -> \x3 -> e`, e incluso como `\x1 x2 x3 -> e`.

- La operación de aplicación tiene prioridad frente a la abstracción. Por ejemplo, `(\x -> y x)(x z)` abrevia `((\x -> (y x))(x z))`.

Veamos algunos ejemplos de escritura abreviada de expresiones. En algunos de ellos usamos usando nombres de funciones ya conocidas por ejemplos anteriores.

- `sumCuad (max 2 3)` abrevia `(sumCuad ((max 2) 3))`.

- `sumCuad (max 2 3)` no se puede abreviar como `sumCuad max 2 3`. Esta última expresión indica que la función `sumCuad` recibe tres parámetros llamados `max`, `2` y `3`, lo cual es incompatible con el tipo de `sumCuad` (que espera un parámetro de tipo entero).

- `f (g x y) x (g y z)` abrevia `(((f ((g x) y)) x) ((g y) z))`.

Funciones curryficadas y aplicaciones parciales

Más arriba hemos dicho que la sintaxis abreviada `f e1 ... en` indica una llamada a la función `f` con parámetros `e1`, ..., `en`. Esta forma de escribir las llamadas a funciones se llama *curryficada*, en recuerdo del matemático *Haskell Brooks Curry*. La curryficación es típica de los lenguajes de la familia de Haskell, y suele causar problemas a los principiantes. Para entender bien la notación curryficada, es importante observar su relación con el tipo de las funciones. Las dos situaciones siguientes son diferentes:

1. `f e1 ... en`
   Esta sintaxis indica una función `f :: T1 -> ... Tn -> R` con `n` parámetros `ei :: Ti` ($1 \leq i \leq n$). Como ejemplo más concreto:

   ```
   tresIguales        :: Int -> Int -> Int -> Bool
   tresIguales x y z   = x == y && x == z & y == z
   ```

2. `f' (e1, ..., en)`
   Esta sintaxis indica una función `f' :: (T1, ..., Tn) -> R` con un parámetro `(e1, ..., en) :: (T1, ..., Tn)` que es una `n`-tupla. Como ejemplo más concreto:

```
tresIguales'         :: (Int,Int,Int) -> Bool
tresIguales' (x,y,z)  = x == y && x == z & y == z
```

Las dos funciones `tresIguales` y `tresIguales'` terminan devolviendo el mismo resultado, pero no son la misma función. Se dice que `tresIguales` es la versión curryficada de `tresIguales'`. Son muy diferentes con respecto a su tipo. Teniendo en cuenta los convenios de escritura abreviada de tipos, vemos que el tipo de `tresIguales` se puede escribir de varias formas equivalentes:

1. `tresIguales :: Int -> (Int -> Int -> Bool)`
   Esta visión del tipo de la función sugiere aplicarla a un solo parámetro entero:

   ```
   tresIguales 2 :: Int -> Int -> Bool
   ```

2. `tresIguales :: Int -> Int -> (Int -> Bool)`
   Esta visión del tipo de la función sugiere aplicarla a dos parámetros enteros consecutivos:

   ```
   tresIguales 2 5 :: Int -> Bool
   ```

3. `tresIguales :: Int -> Int -> Int -> Bool`
   Esta visión del tipo de la función sugiere aplicarla a tres parámetros enteros consecutivos:

   ```
   tresIguales 2 5 7 :: Bool
   ```

Los ejemplos anteriores muestran aplicaciones de `tresIguales` a menos de tres parámetros. En general, una función currificada de aridad n admite varias aplicaciones de la forma `f e1 ... em` con m parámetros `ei`, siendo $0 \leq m \leq n$. En el caso $m < n$ se dice que la expresión `f e1 ... em` es una *aplicación parcial* de `f`.

La idea de aplicación parcial se comprende mejor observando que *las funciones curryficadas consumen sus parámetros uno tras otro.* Esto se entiende teniendo en cuenta los convenios de abreviatura de expresiones. Por ejemplo, la expresión `tresIguales 2 5 7` escrita sin abreviaturas queda `(((tresIguales 2) 5) 7)`, que se puede analizar paso a paso como sigue:

- `tresIguales :: Int -> (Int -> Int -> Bool)`
  Función que espera un parámetro entero y devuelve otra función.
- `tresIguales 2 :: Int -> (Int -> Bool)`
  Función que espera un parámetro entero y devuelve otra función.
- `tresIguales 2 5 :: Int -> Bool`
  Función que espera un parámetro entero y devuelve otra función.

- `tresIguales 2 5 7 :: Bool`
  Valor booleano.

Evaluación de aplicaciones parciales

Al tratar de evaluar una aplicación parcial, tal como `tresIguales 2 5`, se observa que el intérprete Hugs 98 produce un mensaje de error. Esto no se debe a que el valor de la expresión `tresIguales 2 5` esté indefinido. Dicho valor es una función, y el error se produce a nivel de entrada/salida porque el lenguaje Haskell no es capaz de representar valores funcionales en un formato visualizable.

En general, una aplicación parcial `f e1 ... em` de una función curryficada `f` de aridad mayor que m, representa un valor en forma normal siempre que los parámetros `ei` sean expresiones en forma normal. Dicho valor puede ser calculado y manejado internamente por Haskell, pero no puede ser visualizado.

Sintaxis de los patrones

Los parámetros formales utilizados en las definiciones de funciones deben ser expresiones con una forma sintáctica restringida, llamadas patrones. La sintaxis de los patrones, expresada con reglas BNF, es como sigue:

```
p ::=   x
      | c
      | (p1, ..., pn)
      | (C p1 ... pn)
```

Es decir, se admiten cuatro tipos de patrones, cuyo significado es:

- `x`
  Identificador. Representa un parámetro formal que admite a cualquier valor (del tipo adecuado) como parámetro actual.

- `c`
  Constante, como por ejemplo `2`. Representa un parámetro formal que solo admite un valor como parámetro actual.

- `(p1, ..., pn)`
  Tupla formada por n patrones, $n \geq 0$. Representa un parámetro formal que admite tuplas como parámetros actuales. En el caso $n = 0$ se tiene la tupla vacía `()`. En el caso $n = 1$, el patrón `(p)` se puede identificar con `p`.

- `(C p1 ... pn)`
  Patrón formado con una constructora de datos n-ádica `C` y n patrones `pi`, siendo $n \geq 0$. Esta clase de patrones se estudiarán en el capítulo

2, y representan parámetros formales que admiten como parámetro actual valores del tipo asociado a la constructora `C`. Por ejemplo, `True` y `False` son dos patrones correspondientes a las dos constructoras 0-ádicas del tipo `Bool`.

Como ejemplo sencillo de uso de patrones, veamos la definición de funciones que seleccionan las tres componentes de ternas de números enteros:

```
primera         :: (Int,Int,Int) -> Int
primera (x,_,_)  = x

segunda         :: (Int,Int,Int) -> Int
segunda (_,y,_)  = y

tercera         :: (Int,Int,Int) -> Int
tercera (_,_,z)  = z
```

El ejemplo anterior también ilustra el uso de la *variable anónima*, representada en Haskell por el signo de subrayado bajo `_`. Cada aparición de este signo en el lado izquierdo de una ecuación se interpreta como si fuese un identificador diferente. Por ejemplo, la ecuación que define a la función `segunda` es equivalente a:

```
segunda (x,y,z)  = y
```

Lógicamente, las variables anónimas solo se pueden escribir en el lugar de variables que no necesiten aparecer en el lado derecho de la ecuación correspondiente.

En este punto, conviene volver a recordar algo que ya se ha dicho al comienzo de esta sección: *en un lado izquierdo de una ecuación de un progama nunca pueden aparecer variables repetidas.* Por lo tanto, los patrones utilizados en dichos lados izquierdos tampoco pueden contener repeticiones de variables.

Ajuste de una expresión a un patrón

Para reducir una expresión de la forma `(f e1 ... en)`, hay que localizar una ecuación para `f` en el programa cuya condición (si la hay) se pueda evaluar a `True`, y cuyo lado izquierdo `(f p1 ... pn)` cumpla que los parámetros actuales `ei` *se ajusten* a los parámetros formales `pi`. Se dice que una expresión `e` *se ajusta* a un patrón `p` si `e` tiene la forma de `p` con las variables de `p` sustituidas adecuadamente por otras expresiones. Para comprobrar si una expresión se ajusta a un patrón, se utilizan las reglas siguientes:

- Cualquier expresión `e` se ajusta al patrón `x`, con la sustitución que reemplaza `x` por `e`, indicada como $\{\texttt{x} \mapsto \texttt{e}\}$.

- Cualquier expresión `e` se ajusta al patrón `_`, sin necesidad de sustitución.

- La expresión `e` se ajusta al patrón constante `c` si y solo si `e` es sintácticamente idéntica a `c`. No se produce ninguna sustitución.

- Una expresión `e` se ajusta al patrón $(\texttt{p}_1, \cdots, \texttt{p}_n)$ si y solo si `e` es de la forma $(\texttt{e}_1, \cdots, \texttt{e}_n)$ y cada $\texttt{e}_i$ se ajusta a $\texttt{p}_i$. La sustitución producida por el ajuste es el resultado de reunir las sustituciones que produzcan los ajustes de cada $\texttt{e}_i$ a $\texttt{p}_i$. En particular, `()` es la única expresión que se ajusta al patrón `()`.

- Una expresión `e` se ajusta al patrón $(\texttt{C}\ \texttt{p}_1\ \cdots\ \texttt{p}_n)$ si y solo si `e` es de la forma $(\texttt{C}\ \texttt{e}_1\ \cdots\ \texttt{e}_n)$ y cada $\texttt{e}_i$ se ajusta a $\texttt{p}_i$. En particular, la única expresión que se ajusta a una constructora constante `C` es la propia `C`. La sustitución producida por el ajuste es el resultado de reunir las sustituciones que produzcan los ajustes de cada $\texttt{e}_i$ a $\texttt{p}_i$.

- Una expresión solo se ajusta a un patrón en los casos señalados anteriormente.

Según las reglas anteriores, siempre que una expresión e se ajusta a un patrón p se produce una *sustitución de ajuste* $\{x_1 \mapsto e_1, \cdots, x_k \mapsto e_k\}$, donde x_i $(1 \leq i \leq k)$ son las variables de p y e_i $(1 \leq i \leq k)$ son ciertas subexpresiones de e. Por ejemplo, la expresión `(7+3, (sumCuad 5, 3*4))` se ajusta al patrón `(x, (y, z))`, y la sustitución de ajuste es $\{$`x` $\mapsto$ `7+3, y` $\mapsto$ `sumCuad 5, z` $\mapsto$ `3*4`$\}$. En cambio, la misma expresión no se ajusta al patrón `((x, y), z)`

Operadores

Algunas funciones de dos argumentos se escriben como *operadores infijos* con el primer argumento a la izquierda y el segundo argumento a la derecha. Lo mismo que otros lenguajes de programación, Haskell dispone de una serie de operadores infijos predefinidos, incluyendo los símbolos de las operaciones aritméticas, las operaciones de comparación y las operaciones boolenas que estudiaremos en el capítulo 2. En un programa Haskell se pueden incluir también declaraciones de operadores definidos por el usuario, indicando su *precedencia* y su *asociatividad*. La precedencia debe ser un número entero `p` comprendido entre 0 y 9. La asociatividad puede ser `r` (asociar por la derecha), `l` (asociar por la izquierda), o no estar definida. Según ésto, la declaración de un símbolo $\oplus$ como operador infijo puede tener una de las tres formas siguientes:

```
infix  p ⊕  -- Precedencia p, no asocia
infixr p ⊕  -- Precedencia p, asocia por la derecha
infixl p ⊕  -- Precedencia p, asocia por la izquierda
```

Frecuentemente usaremos el símbolo $\oplus$ para referirnos a un operador arbitrario, aunque este símbolo no se puede escribir en la sintaxis concreta de Haskell. Para el uso práctico de operadores son útiles las indicaciones siguientes:

- Un operador $\oplus$ encerrado entre paréntesis se comporta como un identificador de función ($\oplus$) que puede usarse en notación prefija. Por ejemplo, `3 + 5` y `(+) 3 5` son expresiones equivalentes.
- Un identificador de función `f` encerrado entre comillas inclinadas hacia atrás se convierte en un operador infijo `'f'`. Por ejemplo, `max 3 5` y `3 'max' 5` son expresiones equivalentes.
- La aplicación de funciones que no sean operadores tiene precedencia sobre la aplicación de operadores. Por ejemplo, `doble 2 + 3` se interpreta como `(doble 2) + 3`, no como `doble (2 + 3)`.
- El signo "-" tiene un significado ambigüo. Según el contexto, puede indicar un operador infijo de dos argumentos (correspondiente a la función diferencia) o un operador prefijo de un argumento (correspondiente a la función opuesto). A veces es necesario utilizar paréntesis para resolver la ambigüedad. Por ejemplo, la expresión `cuadrado (-2)` es correcta, mientras que `cuadrado -2` sería malinterpretada como la *diferencia* entre `cuadrado` (¡que no es un número!) y el número `2`.

Secciones de un operador

Haskell ofrece una notación muy útil para obtener funciones a partir de operadores. Suponiendo un operador $\oplus$ `:: A -> B -> R` y dos expresiones `a :: A` y `b :: B`, se tiene:

- `(a⊕) :: B -> R` es una función que calcula el resultado `a ⊕ y` para cualquier `y :: B`. Por ejemplo, `(2*) :: Int -> Int` es una función que calcula el doble de su argumento.
- `(⊕b) :: A -> R` es una función que calcula el resultado `x ⊕ b` para cualquier `x :: A`. Por ejemplo, `(>0) :: Int -> Bool` es una función booleana que reconoce los números enteros positivos.

Las funciones de la forma `(a⊕)` y `(⊕b)` obtenidas de esta manera, se llaman *secciones* del operador $\oplus$.

Diseño de funciones: Distinción de casos

El diseño de funciones en Haskell consiste en la construcción de las ecuaciones que las definen, empleando metódicamente distinciones de casos, y generalmente también recursión. Para expresar distinciones de casos hay varios recursos disponibles: Por un lado, la distinta forma de los patrones en los lados izquierdos de diferentes ecuaciones; por otro lado, diferentes condiciones asociadas como guardas a distintas ecuaciones con un mismo lado izquierdo; y finalmente, expresiones condicionales utilizadas como lado derecho de ciertas ecuaciones.

Como ya sabemos, `if c then e1 else e2` es la forma sintáctica de una *expresión condicional*, que no se debe confundir con una *instrucción condicional* al estilo de los lenguajes imperativos. La *condición* `c` debe ser una expresión de tipo `Bool`, y las dos *ramas* `e1` y `e2` deben ser expresiones del mismo tipo. La regla de indentación explicada más atrás se debe tener en cuenta a la hora de escribir expresiones condicionales anidadas. Para reducir una expresión condicional se debe reducir primero su condición, hasta que resulte uno de los dos valores `True` o `False`. A continuación, se podrá aplicar una de las dos ecuaciones siguientes, que no forman parte textual de los programas Haskell, pero son tenidas en cuenta por las implementaciones del lenguaje:

```
if True  then x else y = x
if False then x else y = y
```

Más atrás ya hemos visto definiciones de la función `max` que utilizaban distinciones de casos basadas en guardas. Otra definición equivalente de la misma función se puede escribir utilizando una única ecuación incondicional, cuyo lado derecho es una expresión condicional:

```
max     :: Int -> Int -> Int
max n m  = if n >= m then n else m
```

Un ejemplo típico de distinción de casos basada en patrones se presenta en la función `(&&)`, que representa la operación booleana de conjunción y está predefinida en el preludio de Haskell del siguiente modo:

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

En ejemplos más complejos, todas las técnicas de distinción de casos disponibles se pueden combinar dentro de la definición de una misma función.

Diseño de funciones: Recursión

Como hemos visto al definir la función `sumCuad` en la sección 1.1, la definición de una función en Haskell puede ser recursiva. En general, los programas Haskell deben usar recursión para todos aquellos problemas que se programarían con un bucle iterativo en un lenguaje imperativo, ya que el concepto de bucle no tiene sentido en un lenguaje funcional puro. Un método aconsejable para diseñar definiciones recursivas consiste en seguir un esquema. Se conocen diferentes esquemas, según cual sea el tipo de datos sobre el que opere la recursión. Dos esquemas muy útiles para la definición de funciones recursivas que operan con números enteros, son la *recursión natural* y la *recursión con acumulador*, que presentamos a continuación. En capítulos posteriores del curso estudiaremos técnicas de programación recursiva de funciones que operen con otros tipos de datos.

La *recursión natural* sirve para definir funciones `f :: Int -> R` según el esquema indicado más abajo. El parámetro `n` gobierna la recursión. La distinción de casos distingue el *caso directo* `n == 0`, en el cual se devuelve un resultado representado por la expresión `e :: R`, y el *caso recursivo* `n > 0`, en el cual se utiliza una función `op :: Int -> R -> R` supuestamente conocida para calcular el resultado deseado a partir de `n` y del resultado de la llamada recursiva, que es `(f (n-1))`.

```
f        :: Int -> R
f n
 | n == 0 = e
 | n  > 0 = op n (f (n-1))
 | n  < 0 = error "Argumento negativo!"
```

Se presupone que la función `f` solo va a aplicarse a enteros no negativos; una llamada `(f n)` con `n` negativo causará un error de ejecución. La última ecuación del esquema sirve para que en este caso se muestre al usuario un mensaje de error. Es posible omitirla; la única diferencia es que en este caso las llamadas a `f` con argumento negativo causarán un error sin mensaje alguno.

Un ejemplo sencillo de uso del esquema de recursión natural es la siguiente definición de la función factorial:

```
fact        :: Int -> Int
fact n
 | n == 0    = 1
 | n  > 0    = n * fact (n-1)
 | n  < 0    = error "Argumento negativo!"
```

En este caso, vemos que la expresión `n * (fact (n-1))` sustituye a lo que sería `op n (fact (n-1))` en una aplicación literal del esquema. En la

práctica, muchos usos del esquema de recursión simple omiten una definición explícita y separada de la función `op`. También resulta útil en muchos casos prácticos generalizar el esquema de recursión simple, admitiendo más parámetros adicionales de tipos arbitrarios, además del parámetro de tipo `Int` que gobierna la recursión.

En general, la recursión natural es más ineficiente de lo que sería un bucle programado en estilo imperativo, para resolver el mismo problema. En cambio, el esquema de *recursión final con acumulador* permite definir funciones que se ejecutan con una eficiencia comparable a la de los bucles imperativos. El planteamiento general de la definición de una función según este esquema es como sigue:

```
f        :: R -> Int -> R
f e n
 | n == 0 = e
 | n  > 0 = f (op e n) (n-1)
 | n  < 0 = error "Argumento negativo!"
```

El parámetro `n` gobierna la recursión, mientras que el parámetro adicional `e` hace el papel de acumulador. El esquema se puede generalizar fácilmente al caso de funciones que tengan otros parámetros además de estos dos. La distinción de casos distingue el *caso directo* `n == 0`, en el cual se devuelve como resultado el valor almacenado en el acumulador, y el *caso recursivo* `n > 0`, en el cual se utiliza una función `op :: R -> Int -> R` supuestamente conocida para calcular un nuevo valor del acumulador, con el cual se efectúa la llamada recursiva final `f (op e n) (n-1)`. Lo mismo que en el caso de la recursión natural, se presupone que la función `f` solo va a aplicarse a enteros no negativos, y la última ecuación se encarga de mostrar un mensaje de error cuando este prerequisito no se cumple.

Como ejemplo de uso de recursión final con acumulador, veamos una definición alternativa de la función `fact`, usando una función auxiliar `acuFact`:

```
fact   :: Int -> Int
fact n  = acuFact 1 n

acuFact     :: Int -> Int -> Int
acuFact e n
 | n == 0    = e
 | n  > 0    = acuFfact (e*n) (n-1)
 | n  < 0    = error "Argumento negativo!"
```

Recursión múltiple y recursión mutua

En ocasiones, la definición recursiva de una función requiere que se produzcan *varias* llamadas recursivas en el lado derecho de una misma ecuación. A ésto se le llama *recursión múltiple*. En el caso de *dos* llamadas recursivas,

se habla de *recursión doble*. Por ejemplo, la siguiente definición es un caso típico de recursión doble. `(fib n)` está definido para `n >= 0` y devuelve el término de lugar `n` de una sucesión que los matemáticos llaman *sucesión de Fibonacci*.

```
fib       :: Int -> Int
fib n
  | n == 0 = 0
  | n == 1 = 1
  | n  > 0 = fib (n-2) + fib (n-1)
```

Haskell también permite definir varias funciones usando *recursión mutua*, de tal modo que la definición de cada función pueda contener llamadas a las otras. En estos casos, el orden textual de las definiciones es irrelevante. En general, Haskell ignora el orden en el que se escriban las diferentes definiciones de funciones dentro de un mismo programa. Por ejemplo, usando recursión mutua, podemos definir dos funciones booleanas `esPar` y `esImpar` que reconocen los números enteros no negativos pares e impares, respectivamente. La definición de cada una de las dos funciones sigue un esquema similar a la recursión simple, pero incluyendo una llamada a la otra función.

```
esPar     :: Int -> Bool
esPar n
  | n == 0 = True
  | n  > 0 = esImpar (n-1)

esImpar    :: Int -> Bool
esImpar n
  | n == 0 = False
  | n  > 0 = esPar (n-1)
```

Para valores no negativos de `n`, las definiciones anteriores son equivalentes a las escritas a continuación, que usan expresiones condicionales:

```
esPar  :: Int -> Bool
esPar n = if n == 0 then True else esImpar (n-1)

esImpar  :: Int -> Bool
esImpar n = if n == 0 then False else esPar (n-1)
```

En general, las definiciones recursivas que siguen esquemas del estilo de la recursión simple o final, se pueden escribir también con expresiones condicionales.

Definiciones locales: `where`

Las definiciones que aparecen al nivel más externo de un programa Haskell son *globales*, es decir, visibles desde cualquier punto del programa. En ocasiones interesa utilizar *definiciones locales* que solo sean visibles desde los lados derechos de un grupo de ecuaciones condicionales. En Haskell, estas definiciones locales se introducen por medio de la palabra reservada `where`.

En particular, es frecuente utilizar definiciones locales de constantes para evitar que su valor se calcule varias veces. Esto ocurre en la siguiente función, que calcula las dos raíces de la ecuación de segundo grado $ax^2 + bx + c = 0$, a partir de los coeficientes a, b y c dados como parámetros. Se presupone que $a \neq 0$ y que $b^2 - 4ac \geq 0$.

```
raices      :: Float -> Float -> Float -> (Float,Float)
raices a b c = ((-b-d)/e,(-b+d)/e)
               where d = sqrt(b*b-4.0*a*c)
                     e = 2.0*a
```

Obsérvese que el resultado calculado por la función anterior es una *pareja* de números de tipo `Float`. [1] Nótese también que los parámetros formales `a`, `b` y `c` son visibles en los lados derechos de las definiciones locales de `d` y `e`.

En general, la la construcción `where` siempre introduce definiciones locales visibles desde los lados derechos de un grupo de ecuaciones con lado izquierdo común. La sintaxis a emplear es la siguiente:

```
f p1 ... pn
  | c1        = e1
  | c2        = e2

 ...

  | otherwise = ek
               where def1
                     def2
                     ...
                     defm
```

Aquí, las definiciones locales `defk` pueden definir funciones cualesquiera (en particular, constantes) y son visibles únicamente a efectos de la evaluación de las expresiones `ej`. Los lados derechos de las definiciones locales pueden utilizar las variables que aparezcan en los parámetros formales `pi` del lado izquierdo al que esté asociado el `where`. Se admiten definiciones locales anidadas, que conviene escribir teniendo en cuenta la regla de indentación explicada más atrás. También hay que tener en cuenta el siguiente criterio:

[1] Véase la sección 2.1 para una explicación de los tipos numéricos disponibles en Haskell.

Alcance de las definiciones: En cada punto de un programa donde aparezca un identificador de función o constante, éste se refiere a la definición *más local* de dicho identificador. No obstante, se recomienda no repetir un mismo identificador para una definición local más interna, pues esta práctica solo causa confusión.

La siguiente definición de una función `f` ilustra el anidamiento de definiciones locales y el uso de la regla de indentación:

```
f     :: Int -> Int -> Int
f x y  = g (x+w)
         where g u = u+v
                     where v = u*u
               w   = y+y
```

Equivalentemente, se puede presentar la definición anterior usando explícitamente el separador ";" y los delimitadores "{" y "}", para indicar la estructura de las definiciones locales:

```
f     :: Int -> Int -> Int
f x y  = g (x+w) where {g u = u+v where {v = u*u}; w = y+y}
```

Definiciones locales: `let`

En Haskell también es posible expresar definiciones locales asociadas a una expresión `e`, que solamente se utilizan para la evaluación de `e` y no son visibles desde el resto del programa. Para esto se utiliza la sintaxis

```
let {def1; ...; defm} in e
```

donde `defk` son definiciones locales de constantes o funciones, posiblemente recursivas. El uso explícito de ";", "{" y "}" se puede evitar mediante la regla de indentación. Por ejemplo, la siguiente expresión (cuyo valor es `False`) incluye una definición local de la función recursiva `par`:

```
let {par 0 = True; par n | n > 0 = not (par (n-1))}
    in par 3
```

Diferencia entre `let` y `where`

Aunque `let` y `where` parecen construcciones similares, `where` es más general en el sentido de que introduce deficiones locales cuyo ámbito de validez incluye *varios lados derechos*. Por ejemplo, la definición local usada a continuación no se podría expresar con `let`:

```
f           :: Int -> Int -> Int
f x y
  | y >  z   = 1
  | y == z   = 0
  | y <  z   = -1
               where z = x*x
```

Otra diferencia es que la construcción `let` da lugar a *expresiones*, mientras que la construcción `where` no sirve para construir una expresión independiente; siempre debe acompañar a una definición de función. [2]

Funciones de orden superior

En Haskell y otros lenguajes funcionales, se permiten *funciones de orden superior*. Se llama así a aquellas funciones que admiten como parámetro o devuelven como resultado a otras funciones. Como ejemplo de uso de funciones de orden superior, consideremos el problema de calcular el valor del sumatorio $\Sigma_{i=a}^{i=b}(f\ i)$, que representa la suma de todos los valores de una función f a lo largo de un intervalo de números enteros con extremos a y b. Tomando a f, a y b como parámetros, la siguiente función de orden superior resuelve el problema:

```
sigma       :: (Int -> Int) -> Int -> Int -> Int
sigma f a b
  | a >  b   = 0
  | a <= b   = f a + sigma f (a+1) b
```

Las funciones de orden superior favorecen la reutilización de código. Esto es debido a que distintos reemplazamientos de un parámetro formal que sea una función por parámetros actuales, producen comportamientos distintos pero similares, que no es preciso programar uno por uno. Por ejemplo, las dos funciones que se muestran más abajo sirven para calcular la suma de los n primeros números pares y la suma de los n primeros números impares, respectivamente. Como vemos, su definición es muy sencilla; basta reutilizar la función de orden superior `sigma`, y no es necesario volver a utilizar explícitamente recursión.

```
sumaPares   :: Int -> Int
sumaPares n  = sigma par 0 (n-1)
               where par i = 2*i

sumaImpares   :: Int -> Int
sumaImpares n  = sigma impar 0 (n-1)
                 where impar i = 2*i+1
```

[2] Más adelante aprenderemos que la construcción `where` también puede acompañar a una expresión `case`.

Funciones anónimas

En el ejemplo anterior, las funciones locales `par` e `impar` solo se necesitan como parámetros para `sigma`. Haskell permite también representar la función `impar` sin asociarle un identificador, mediante la notación `(\i -> 2*i+1)`, que significa: "la función con parámetro `i` que devuelve `2*i+1`". La misma técnica se puede aplicar a la función `par`. Siguiendo esta idea se pueden escribir definiciones alternativas de las funciones `sumaPares` y `sumaImpares`:

```
sumaPares   :: Int -> Int
sumaPares n  = sigma (\i -> 2*i) 0 (n-1)

sumaImpares   :: Int -> Int
sumaImpares n  = sigma (\i -> 2*i+1) 0 (n-1)
```

En general, dados un identificador `x` y una expresión `e`, se puede formar la expresión `(\x -> e)`, que representa una función con parámetro formal `x` y cuerpo `e`. Las expresiones de esta forma se llaman *λ-abstracciones*. Puesto que sirven para introducir una función sin asociarle un nombre, se llaman también *funciones anónimas*.

Cuando una función anónima `(\x -> e)` se aplica a un parámetro actual dado por la expresión `a`, resulta la expresión `(\x -> e) a`, cuyo valor se calcula reemplazando el parámetro formal `x` por el actual `a` dentro del cuerpo `e` y reduciendo la expresión resutante. Por ejemplo:

`(\i -> 2*i+1) (5+3)` $\longrightarrow$ `2*(5+3)+1` $\longrightarrow$ `2*8+1` $\longrightarrow$ `16+1` $\longrightarrow$ `17`

La evaluación de llamadas a funciones anónimas se estudiará con más detalle en la sección 1.4.

Funciones polimórficas

Otra técnica de reutilización de código se basa en el *polimorfismo*. Una función se llama *polimórfica* o *genérica* cuando su tipo depende de parámetros formales que representan tipos. Un ejemplo muy simple de función polimórfica es la identidad:

```
id   :: a -> a
id x  = x
```

En cada uso particular de `id`, el parámetro `a` se reemplazará automáticamente por el tipo que convenga, sin que el usuario tenga que intervenir. Así:

- `id (3+2) ::  Int`
  Expresión que usa `id :: Int -> Int` y vale `5`.

- `id (not True) ::  Bool`
  Expresión que usa `id :: Bool -> Bool` y vale `False`.

En general, en cada uso particular de una función polimórfica, los parámetros que intervengan en su tipo se reemplazan automáticamente por los tipos que convengan para que la expresión total de que se trate quede bien tipada.

Otros ejemplos más interesantes de funciones polimóficas son las funciones predefinidas `fst` y `snd`, que extraen las dos componentes de un par ordenado:

```
fst       :: (a,b) -> a
fst (x,_)  = x

snd       :: (a,b) -> b
snd (_,y) = y
```

Un ejemplo de función polimórfica y a la vez de orden superior, es la *composición de funciones*. En Haskell está declarada como operador infijo. y predefinida del siguiente modo:

```
infixr 9 .
(.)    :: (b -> c) -> (a -> b) -> (a -> c)
f . g   = \x -> f (g x)
```

Otra definición de `(.)` equivalente a la anterior es

```
infixr 9 .
(.)       :: (b -> c) -> (a -> b) -> a -> c
(f . g) x  = f (g x)
```

Obsérvese que los tipos declarados en las dos definiciones de `(.)` son el mismo, gracias al convenio de que `->` asocia a la derecha en la construcción sintáctica de los tipos. Se han escrito de diferente manera tan solo para que resulte más claro cual es la aridad de la función en cada caso (2 en la primera definición, y 3 en la segunda).

Como ejemplo de uso de `(.)` en la definición de otras funciones, podemos considerar

```
fun :: Int -> Int
fun  = doble . (+1)
```

que equivale a definir

```
fun   :: Int -> Int
fun n  = doble (n+1)
```

aunque la definición anterior es más concisa. Obsérvese que el tipo de `(.)` dentro de la expresión `(doble . (+1))` es un caso particular del tipo genérico de `(.)`, concretamente:

```
(.) :: (Int -> Int) -> (Int -> Int) -> Int -> Int
```

Equivalencia entre funciones: principio de extensionalidad

Varias veces a lo largo de esta sección hemos hablado de definiciones equivalentes de una misma función. En general, se entiende que dos funciones son equivalentes si tienen el mismo tipo y dan siempre el mismo resultado, cualesquiera que sean los parámetros a que se apliquen. Cuando dos funciones `f` y `g` sean equivalentes en este sentido, escribiremos `f = g`. Por ejemplo, las funciones

```
doble   :: Int -> Int          dobla    :: Int -> Int
doble x  = 2*x                 dobla x   = x+x
```

cumplen que `doble x = dobla x` para cualquier `x :: Int`. Por lo tanto, podemos afirmar que `doble = dobla`. Del mismo modo se pueden demostrar otras equivalencias entre funciones que hemos dado por buenas en ejemplos anteriores.

La equivalencia entre funciones, entendida en el sentido que acabamos de explicar, se llama *igualdad extensional*. En general, para demostrar la igualdad extensionalidad entre dos funciones `f` y `g` que esperen `n` parámetros, hay que considerar nuevas variables `x1, ..., xn` que representan valores cualesquiera de los parámetros, y utilizar las definiciones de `f` y `g` para demostrar la ecuación `f x1 ... xn = g x1 ... xn`. Cuando se razona de esta manera, se dice que se ha aplicado el *principio de extensionalidad*. Dos funciones extensionalmente equivalentes nunca difieren en los resultados que calculan, pero pueden diferir en otras propiedades, en particular la eficiencia.

Transformación de programas

Las técnicas de *transformación de programas* persiguen transformar un programa dado en otro equivalente y posiblemente más eficiente. En el caso de los programas funcionales, se requiere que las funciones del programa transformado sean extensionalmente equivalentes a las funciones del programa original, y se utilizan técnicas de transformación basadas en el cálculo simbólico con ecuaciones.

En particular, es muy útil el método de *plegado-desplegado*, que se remonta al trabajo pionero de Burstall y Darlington [3]. Este método se emplea para transformar una función `fun` con definición ejecutable conocida pero ineficiente, con la intención de mejorar la eficiencia. Los pasos a seguir son, a grandes rasgos, los siguientes:

1. Etapa inicial de *generalización*: Se especifica una función `genFun` más general que `fun`, de manera que `fun` admita una definición sencilla a partir de `genFun`. Generalmente, la especificación de `genFun` se escribe como una ecuación, pero ésta no se admite aún como código ejecutable.

2. Etapa de *transformación*: Usando la definición ineficiente de `fun` y la especificación de `genFun`, se deriva una definición ejecutable de `genFun` mediante cálculo simbólico con ecuaciones. En este proceso, se distinguen varios tipos de pasos:

   - *Desplegado:* se llaman así a los pasos de reescritura, reemplazando una expresión que se ajuste al lado izquierdo de una eccuación por el correspondiente lado derecho, afectado de la sustitución de ajuste.
   - *Plegado:* se llaman así a los pasos de "reescritura a la inversa", reemplazando una expresión que se ajuste al lado derecho de una eccuación por el correspondiente lado izquierdo, afectado de la sustitución de ajuste. Generalmente, los pasos de plegado siempre se dan con la ecuación que especifica a `genFun`.
   - *Equivalencia:* se llaman así a los pasos que reemplazan una expresión por otra que sea equivalente a ella en virtud de alguna ley de equivalencia conocida.
   - *Abstracción:* se llaman así a los pasos que reemplazan una o varias subexpresiones `ei` de una expresión por nuevas variables `ui` ($1 \leq i \leq n$), introduciendo a la vez una definición local de la forma `(u1, ..., un) = (e1, ..., en)`.

3. Finalmente, se reemplaza la definición inicial de `fun` por una ecuación sencilla que defina `fun` en términos de `genFun`, junto con la definición de `genFun` derivada por plegado-desplegado.

Se puede demostrar que el método de plegado-desplegado siempre conduce a un programa funcional equivalente al de partida. En este curso encontraremos con frecuencia situaciones en las que dicho método conduce además a optimizaciones importantes. En algunos casos, la etapa inicial de generalización es innecesaria, y basta con transformar directamente la función que se desea optimizar. En otros casos, sin embargo, la generalización inicial de la función de partida es esencial para que el resto de la transformación consiga una buena optimización.

Un buen ejemplo de plegado-desplegado con generalización de la función de partida, es la optimización de la definición doblemente recursiva de la función de Fibonacci:

```
fib       :: Int -> Int
fib n
  | n == 0 = 0
  | n == 1 = 1
  | n  > 0 = fib (n-2) + fib (n-1)
```

El cálculo de `fib n` empleando esta definición requiere $\mathcal{O}(2^n)$ pasos de reducción. La transformación por plegado-desplegado presentada a continuación termina con una definición muy optimizada de la función de Fibonacci, tal que el número de pasos de reducción empleados en calcular `fib n` desciende drásticamente a $\mathcal{O}(n)$.

1. Etapa de *generalización*:
   La forma del caso recursivo de `fib` motiva considerar una función más general `dosFib`, que devuelva como resultado la pareja formada por dos números de Fibonacci consecutivos. *Especificamos*:

   ```
   dosFib   :: Int -> (Int,Int)
   dosFib n  = (fib n, fib (n+1))
   ```

   No admitimos esta especificación como código ejecutable, pues tendría la misma ineficiencia que la definición de `fib` de que partimos.

2. Etapa de *transformación*:
   Derivamos una definicón recursiva para `dosFib`, planteando una distinción de casos de la forma:

   ```
   dosFib    :: Int -> (Int,Int)
   dosFib n
     | n == 0 = ...
     | n > 0  = ...
   ```

   Para completar los lados derechos, calculamos en cada caso encadenando una serie de ecuaciones según el método de plegado-desplegado. Anotamos cada paso con la inicial de una de las palabras *Desplegado*, *Plegado*, *Equivalencia* o *Abstracción*, según corresponda. Además, en los pasos que utilicen una ecuación, indicamos cual; entendiendo que `dosFib` hace referencia a la ecuación que especifica `dosFib`, y que `fib.1`, `fib.2`, `fib.3` hacen referencia a las tres ecuaciones que definen `fib` en el programa de partida (en orden textual).

   (a) Caso `n == 0`:

   ```
                    dosFib 0
   =(D, dosFib)
                    (fib 0, fib (0+1))
   =(E)
                    (fib 0, fib 1)
   =(D, fib.1,fib.2)
                    (0,1)
   ```

(b) Caso `n > 0`:

```
                dosFib n
=(D, dosFib)
                (fib n, fib (n+1))
=(D, fib.3)
                (fib n, fib ((n+1)-2) + fib ((n+1)-1))
=(E)
                (fib n, fib (n-1) + fib n)
=(A)
                (v,u+v) where {(u,v) = (fib (n-1), fib n)}
=(E)
                (v,u+v) where {(u,v) = (fib (n-1), fib ((n-1)+1))}
=(P, dosFib)
                (v,u+v) where {(u,v) = dosFib (n-1)}
```

3. Etapa *final*:
   La definición inicial de `fib` queda transformada en la definición derivada para `dosFib` junto con la definición obvia de `fib` a partir de `dosFib`, usando la función predefinida `fst` que calcula la primera componente de una pareja:

```
dosFib    :: Int -> (Int,Int)
dosFib n
  | n == 0 = (0,1)
  | n  > 0 = (v,u+v)
             where (u,v) = dosFib (n-1)

fib :: Int -> Int
fib  = fst . dosFib

--  O equivalentemente:
--
--  fib n = fst (dosFib n)
```

   En general, siempre que una transformación de programas emplee una generalización `genFun` de la función `fun` que se desea transformar, debe justificarse la existencia de una definición correcta y simple de `fun` a partir de `genFun`.

Ejercicios

1. Define funciones `f :: Int -> Int -> Bool` y `g :: (Int -> Int) -> Bool`. Usando las definiciones, explica la diferencia entre el tipo de `f` y el tipo de `g`.

2. Define una función `max3 :: Int -> Int -> Int -> Int` que calcule como resultado el máximo de sus tres argumentos.

3. Define una función `cuantosIguales :: Int -> Int -> Int -> Int` que se comporte como sigue:

   - Cuando sus tres argumentos son iguales, la función calcula el resultado `3`.
   - Cuando dos de sus argumentos son iguales y diferentes del otro, la función calcula el resultado `2`.
   - En otro caso, la función calcula el resultado `1`

4. Muestra que la definición recursiva de la función `sumCuad` estudiada en la sección 1.1 se ajusta al esquema de recursión simple. Busca otra definición equivalente que utilice el esquema de recursión final.

5. Razonando por inducción sobre `n`, demuestra que la función `acuFact` definida en esta sección verifica que `acuFact m n = m * fact n` (para cualquier `n :: Int, n` $\geq$ `0`).

6. Se dispone de una función `ventas :: Int -> Int` ya definida, tal que `(ventas n)` calcula el importe de las ventas de un negocio en la semana número n, para enteros $n \geq 0$. Puedes imaginar que `ventas` está definida mediante una serie de ecuaciones, como por ejemplo:

   ```
   ventas 0 = 57000
   ventas 1 = 48000
   ventas 2 = 60000
          ...
   ```

   Usando `ventas` y el esquema de recursión simple, define otra función `maxVentas :: Int -> Int` tal que `(maxVentas n)` calcule el importe de la mayor venta producida durante las semanas del intervalo $[0..n]$, para $n \geq 0$. Prueba la función en algún cálculo.

7. Escribe otra definición de la función `maxVentas`, usando ahora el esquema de recursión final. Prueba la función en algún cálculo.

8. Reconsidera los dos ejercicios anteriores, definiendo `maxVentas` como una función de orden superior que reciba a `ventas` como parámetro.

9. Calcula el valor de `(fib 3)` utilizando la definición doblemente recursiva de la función `fib`.

10. Calula el valor de las siguientes expresiones `let`:

```
let x = 5          let x = 3
    y = 3              in let f n = (n+x)*(n-x)
    s = x+y               in  (f 5, f 7)
    d = x-y
    in  s*d
```

11. Supón que `h x y = f (g x y)`. Estudia si las igualdades siguientes son o no válidas:

    (i) `h = f . g` (ii) `h x = f . (g x)` (iii) `h x y = (f . g) x y`

12. La composición de una función (curryficada) de dos argumentos, `g`, y una función de un argumento `f`, es la función `(f .. g)` que cumple la ecuación `(f .. g) x y = f (g x y)`. Escribe la definición del operador `(..)`, incluyendo una declaración de tipo.

13. la siguiente ecuación no ejecutable especifica el comportamiento deseado para una función `leastForm`:

    `leastFrom n p  =  min i : i` $\geq$ `n :  p i`

    Se supone que `p :: Int -> Bool`. Escribe una definición recursiva ejecutable de `leastFrom`, incluyendo una declaración de tipo.

14. Utilizando la función del ejercicio anterior y sin usar más recursión explícita, define otra función `least` que calcule el menor entero no negativo que cumpla una propiedad `p` dada como parámetro. Incluye en tu definición la declaración del tipo de `leastFrom`.

15. Considera una función para multiplicar números enteros, definida como sigue:

    ```
    producto     :: Int -> Int -> Int
    producto x y  = if x == 0 then x else x*y
    ```

    Suponiendo que las funciones `(==)` y `(*)` necesiten evaluar sus argumentos a forma normal, estudia si la función `producto` es estricta o no, con respecto a cada uno de sus dos argumentos.

16. Demuestra que la función composición `(f . g)` es estricta siempre que `f` y `g` lo sean.

1.4 El cálculo λ

Propósito del cálculo λ

El cálculo λ fué desarrollado por el matemático Alonzo Church durante el periodo 1930–1940, buscando que sirviese como modelo del comportamiento de las funciones desde un punto de vista computacional. Posteriormente, este cálculo ha servido como de fundamento para el diseño e implementación de lenguajes de programación funcionales.

Básicamente, el cálculo λ establece una sintaxis para representar expresiones y unas reglas de reducción para transformarlas. Estas ideas ya han aparecido en las secciones 1.2 y 1.3. En esta sección vamos a ampliar algunos detalles, particularmente en lo que se refiere a la reducción de llamadas a funciones anónimas. Una exposición detallado del cálculo λ cae fuera de los objetivos de este curso. En el capítulo 3 de [8] se puede encontrar una ampliación de este tema, así como referencias bibliográficas para profundizar en su estudio.

Sintaxis de las expresiones

Para los propósitos de esta sección, suponemos expresiones formadas de acuerdo con las siguientes reglas BNF:

```
e ::=   x
      | (e e1)
      | (\x -> e)
      | (e1, ..., en)
      | if e then e1 else e2
      | let {x1 = e1; ...; xn = en} in e
      | letrec {x1 = e1; ...; xn = en} in e
```

Vemos que esta sintaxis es muy similar a la explicada en la sección 1.3, excepto que:

- Solamente admitimos definiciones locales de la forma `xi = ei`, consistentes en una única ecuación que define el comportamiento de un identificador.
- Distinguimos dos clases de expresiones con definiciones locales asociadas: Expresiones `let`, donde se supone que las definiciones locales no pueden ser recursivas; y expresiones `letrec`, cuyas definiciones locales pueden ser recursivas. Como ya sabemos, en Haskell solamente existen definiciones locales con sintaxis `let`, que se interpretan como la sintaxis `letrec` del cálculo λ.

Estas variaciones en la sintaxis se introducen para simplificar el estudio teórico del cálculo λ. Una gran parte de los resultados matemáticos conocidos se refieren al llamado cálculo λ *puro*, que utiliza solamente expresiones de los tres primeros tipos (es decir, `x`, `(e e1)` y `(\x -> e)`), y no permite

identificadores que representen funciones predefinidas. El cálculo λ extendido con el resto de las construcciones sintácticas que estamos considerando se llama *aplicado*, y permite identificadores que representen funciones predefinidas. Se ha demostrado matemáticamente que el cálculo λ puro es capaz de simular el comportamiento del cálculo λ aplicado, pero la simulación es muy engorrosa, y en la práctica es necesario utilizar una sintaxis más amplia.

El significado de cada una de las clases de expresiones y los convenios para escribir expresiones en forma abreviada ya los hemos explicado en la sección 1.3. Al escribir expresiones del cálculo λ aplicado admitiremos también el uso de operadores infijos $\oplus$, que siempre se pueden convertir en funciones prefijas ($\oplus$) según un convenio que ya conocemos.

Expresividad de la sintaxis

Desde el punto de vista de la complejidad de su sintaxis, el cálculo λ aplicado se encuentra a medio camino entre el cálculo λ puro y los lenguajes funcionales reales, tales como Haskell. Aunque Haskell admite muchas construcciones sintácticas que no existen en el cálculo λ aplicado, es posible traducir programas Haskell a expresiones de dicho cálculo. Suponiendo un programa Haskell $\mathcal{P}$ que contenga las definiciones de n funciones `f1`, ..., `fn` y una expressión `e`, se puede construir una única expresión del cálculo λ aplicado, de la forma `letrec {f1 = d1; ...; fn = dn} in e`, que representa el problema de evaluar `e` en presencia de las definiciones de $\mathcal{P}$.

Por ejemplo, supongamos que $\mathcal{P}$ contenga la definición de la función factorial mediante recursión final, tal como la vimos en la sección 1.3:

```
fact   :: Int -> Int
fact n  = acuFact 1 n

acuFact     :: Int -> Int -> Int
acuFact e n
 | n == 0    = e
 | n  > 0    = acuFact (e*n) (n-1)
 | n  < 0    = error "Argumento negativo!"
```

Supongamos además que `e` sea una llamada a `fact`, tal como `(fact 5)`. La expresión única del cálculo λ aplicado que representa el problema de la evaluación de `e` usando $\mathcal{P}$ en este caso puede ser la siguiente, ignorando las declaraciones de tipo y el mensaje de error:

```
letrec {fact    = \n -> acuFact 1 n ;
        acuFact = \e -> \n -> if n == 0
                              then e
                              else acuFact (e*n) (n-1)

        }
 in fact 5
```

Identificadores libres y ligados; renombramiento

Cada aparición de un identificador `x` dentro de una expresión `e` puede estar *ligada* o *libre*. Como veremos enseguida, esta distinción es necesaria para definir las reglas de reducción de expresiones en el cálculo λ. Se considera que:

- Están ligadas todas la apariciones de `x` en la expresión `(\x -> e)`.
- Están ligadas todas la apariciones de `x1`, ..., `xn` en la expresión `let {x1 = e1; ...; xn = en} in e`.
- Están ligadas todas la apariciones de `x1`, ..., `xn` en la expresión `letrec {x1 = e1; ...; xn = en} in e`.
- Una aparición de un identificador en una expresión solo está ligada si aparece en un contexto correspondiente a alguno de los tres casos anteriores; en caso contrario, está libre.

Por ejemplo, en la expresión `(\x -> y x)(x z)`, la tercera aparición de `x` es libre, mientras que las otras dos son ligadas. Las apariciones de `y`, `z` son libres. Si se reemplazan las apariciones ligadas de `x` por un nuevo identificador `x'` resulta la expresión `(\x' -> y x')(x z)` que es intuitivamente equivalente a la de partida.

En general, siempre que una expresión `e'`, se obtenga a partir de otra expresión `e` renombrando algunos identificadores ligados de `e`, se dice que `e'` es una *variante* de `e`. Cualquier expresión se considera equivalente a cualquiera de sus variantes, ya que los nombres de los identificadores ligados no afectan al significado de las expresiones.

En lo sucesivo, a veces nos interesará construir una variante `e'` de una expresión dada `e`, de tal manera que ningún identificador ligado de `e'` aparezca libre en otra expresión dada `e1`. En tales casos, diremos que `e'` es una *variante de* `e` *que evita conflictos con* `e1`.

La regla de reducción β

Una regla de reducción muy importante, tanto para el cáculo λ puro como para el cáculo λ aplicado, es la que especifica como reducir una expresión que tenga la forma de una función anónima aplicada a un parámetro. Por razones históricas, los pasos de reducción dados conforme a esta regla se llaman pasos de reducción β. Se especifican del modo siguiente:

(β) `(\x -> e) e1` $\longrightarrow_\beta$ `e' [x` $\mapsto$ `e1]`

En la regla (β), `e'` representa cualquier variante de `e` que evite conflictos con `e1`. Por otra parte, la notación `e' [x` $\mapsto$ `e1]` indica la expresión resultante de sustituir todas las apariciones libres de `x` dentro de `e'` por `e1`.

Según esto, un paso de reducción β da como resultado el cuerpo de la función anónima de que se trate, pero reemplazando el parámetro formal por el correspondiente parámetro actual. En lo sucesivo usaremos la notación `e0` $\longrightarrow_\beta$ `e1` para indicar que `e0` se reduce a `e1` aplicando la regla β a alguna subsexpresión de `e`. Los ejemplos siguientes muestran algunos casos típicos de reducción β:

1. `(\x -> x+x) 3` $\longrightarrow_\beta$ `3+3`
   En efecto, `(x+x) [x` $\mapsto$ `3]` $\equiv$ `3+3`.

2. `(\x -> x + (\x -> x) y) 3` $\longrightarrow_\beta$ `3 + (\x -> x) y`
   En efecto, `(x + (\x -> x) y) [x` $\mapsto$ `3]` $\equiv$ `3 + (\x -> x) y`, teniendo en cuenta que solo se sustituyen las apariciones libres de `x`.

3. `(\x -> \y -> x+2*y) (3*y)` $\longrightarrow_\beta$ `\y' -> 3*y+2*y'`
   En efecto, `(\y' -> x+2*y') [x` $\mapsto$ `3*y]` $\equiv$ `\y' -> 3*y+2*y'`. En este caso, `(\y -> x+2*y)` se ha reemplazado por la variante `(\y' -> x+2*y')`, a fin de evitar conflictos con `3*y`.

La reglas de reducción η

Junto con (β), la regla (η) es suficiente para investigar el comportamiento de la reducción de expresiones en el cálculo λ puro. Se formula del siguiente modo:

(η) `\x -> (e x)` $\longrightarrow_\eta$ `e`
supuesto que `x` no aparezca libre en `e`

Esta regla se justifica por el *principio de extensionalidad* que ya hemos estudiado en la sección 1.3. En efecto, considerando cualquier identificador nuevo `z` que no aparezca en `\x -> (e x)` ni en `e`, resulta

`(\x -> (e x)) z` $\longrightarrow_\beta$ `(e z)`

debido a que `(e x) [x` $\mapsto$ `z]` $\equiv$ `(e z)`. Por lo tanto, `\x -> (e x)` y `e` dan el mismo resultado para cualquier argumento `z`, por lo cual es correcto considerarlas equivalentes.

La reducción η se puede emplear también en el cáculo λ aplicado. En lo sucesivo usaremos la notación `e` $\longrightarrow_\eta$ `e'` para indicar que `e` se reduce a `e'` aplicando la regla η a alguna subsexpresión de `e`, y emplearemos notaciones similares para las demás reglas de reducción que vamos a considerar en el

resto de la sección. Un ejemplo concreto de uso de la regla η es

`\x -> (\y -> x y)` $\longrightarrow_\eta$ `\x -> x`

donde el paso de reducción η se ha aplicado a la subexpresión `\y -> x y`.

Otras reglas de reducción

Las reglas siguientes especifican como reducir expresiones condicionales y expresiones con definiciones locales, respectivamente:

(IF_1) `if True then e1 else e2` $\longrightarrow_{IF}$ `e1`

(IF_2) `if False then e1 else e2` $\longrightarrow_{IF}$ `e2`

(LET) `let x1 = e1 in e` $\longrightarrow_{LET}$ `e' [x1` $\mapsto$ `e1]`

($LETREC$) `letrec x1 = e1 in e` $\longrightarrow_{LETREC}$
`e' [x1` $\mapsto$ `e1'[x1` $\mapsto$ `letrec x1 = e1 in x1]]`

En las reglas (LET) y ($LETREC$), se supone que `e'` y `e1` son variantes elegidas para evitar conflictos en las sustituciones. Por simplicidad, estas dos reglas se han limitado al caso de que se defina localmente un solo identificador. El caso general se puede tratar con una regla análoga, algo más compleja. Según la regla (LET), una expresión de la forma `let x1 = e1 in e` se reduce del mismo modo que `(\x1 -> e) e1`. La regla ($LETREC$) es más complicada; especifica que `x1` se debe sustituir *recursivamente* por `e1` durante la evaluación de `e`. En la práctica, las expresiones con definiciones locales (recursivas o no) se pueden reducir transformando su cuerpo con pasos de reescritura, y consultando las definiciones locales en aquellos pasos que así lo requieran.

Una última regla de reducción es (δ), que refleja el comportamiento de aquellos identificadores que representen funciones para las que se presupone una definición conocida. La notación `e` $\longrightarrow_\delta$ `e'` indica que el paso de `e` a `e'` se realiza sustituyendo una subexpresión de `e` que tenga la forma de una llamada a una función con definición conocida, por el resultado de reducirla. Por ejemplo, suponiendo conocida la definición de `(+)` se tiene `3+2` $\longrightarrow_\delta$ `5`.

En lo sucesivo, usaremos las notaciones siguientes:

- `e` $\longrightarrow$ `e'`
  Indica que `e` se reduce a `e'` en un paso, mediante alguna de las reglas de reducción.

- `e` $\longrightarrow^*$ `e'`
  Indica que `e` se reduce a `e'` en un número finito n de pasos, $n \geq 0$.

- `e` $\longrightarrow^+$ `e'`

  Indica que `e` se reduce a `e'` en un número finito n de pasos, $n > 0$.

Las notaciones `e` $\longrightarrow^*_\beta$ `e'` y `e` $\longrightarrow^+_\beta$ `e'` tiene un significado similar, pero considerando únicamente pasos de reducción β.

Propiedades de confluencia

Aceptamos sin demostración el siguiente resultado:

Propiedad de confluencia de la reducción β: siempre que `e` $\longrightarrow^*_\beta$ `e1` y `e` $\longrightarrow^*_\beta$ `e2`, existe `e'` tal que `e1` $\longrightarrow^*_\beta$ `e'` y `e2` $\longrightarrow^*_\beta$ `e'`.

Por ejemplo, tomando

`e` $\equiv$ `(\x -> (\y -> y x) z) v`

`e1` $\equiv$ `(\y -> y v) z` `e2` $\equiv$ `(\x -> z x) v`

se tiene que `e` $\longrightarrow_\beta$ `e1` y `e` $\longrightarrow_\beta$ `e2`. Por la propiedad de confluencia, debe existir `e'` tal que `e1` $\longrightarrow^*_\beta$ `e'` y `e2` $\longrightarrow^*_\beta$ `e'`. Y en efecto, tomando `e'` $\equiv$ `z v`, se cumple que `e1` $\longrightarrow_\beta$ `e'` y `e2` $\longrightarrow_\beta$ `e'`.

El resultado anterior se puede generalizar del siguiente modo:

Propiedad de confluencia: suponiendo que las funciones predefinidas y las funciones definidas localmente no introduzcan ambigüedades, siempre que `e` $\longrightarrow^*$ `e1` y `e` $\longrightarrow^*$ `e2`, existe `e'` tal que `e1` $\longrightarrow^*$ `e'` y `e2` $\longrightarrow^*$ `e'`.

La "ausencia de ambigüedades" significa que no debe ser posible aplicar dos ecuaciones diferentes para reescribir una misma llamada a una función. En teoría, es posible escribir definiciones ambigüas, como por ejemplo:

```
pim   :: Int -> Int
pim n  = 2*n
pim n  = 3*n
```

En presencia de estas ecuaciones la propiedad de confluencia se perdería en teoría, debido a las dos reducciones `pim 1` $\longrightarrow$ `2` y `pim 1` $\longrightarrow$ `3`. En la práctica, Hugs 98 solamente ejecutaría la primera de las dos reducciones, ya que el intérprete ensaya las ecuaciones del programa en orden textual.

Propiedades de terminación

Dada una expresión `e`, definimos:

1. La reducción β a partir de `e` termina si no existe ninguna sucesión infinita de expresiones $\mathtt{e}_i$ tales que

   $\mathtt{e} \equiv \mathtt{e}_0 \longrightarrow_\beta \mathtt{e}_1 \ \ldots \ \longrightarrow_\beta \mathtt{e}_i \longrightarrow_\beta \mathtt{e}_{i+1} \ \ldots$

2. `e` tiene forma normal `e'` con respecto a la reducción β si $\mathtt{e} \longrightarrow_\beta^* \mathtt{e'}$ y `e'` no es reducible con (β).

3. La reducción a partir de `e` termina si no existe ninguna sucesión infinita de expresiones $\mathtt{e}_i$ tales que

   $\mathtt{e} \equiv \mathtt{e}_0 \longrightarrow \mathtt{e}_1 \ \ldots \ \longrightarrow \mathtt{e}_i \longrightarrow \mathtt{e}_{i+1} \ \ldots$

4. `e` tiene forma normal `e'` si $\mathtt{e} \longrightarrow^* \mathtt{e'}$ y `e'` no es reducible.

Claramente, 1. $\Rightarrow$ 2., 3. $\Rightarrow$ 4. y 3. $\Rightarrow$ 1. En general, 3. no se cumple debido a que las definiciones recursivas pueden causar no terminación, tanto si se trata de definiciones locales como de definiciones que se supongan disponibles para pasos de reducción δ. Tampoco 1. se cumple siempre. Por ejemplo, la reducción β a partir de la expresión `(\x -> x x) (\x -> x x)` no termina, ya que

`(\x -> x x) (\x -> x x)` $\longrightarrow_\beta$ `(\x -> x x) (\x -> x x)` $\longrightarrow_\beta \ldots$

Aceptamos sin demostración la propiedad siguiente:

Propiedad de terminación para la reducción β: La reducción β termina a partir de cualquier expresión `e` que admita tipo en el sistema de tipos de Milner, que estudiaremos en la sección 1.5.

La propiedad anterior puede fallar cuando `e` no admite tipo en el sistema de Milner, como ocurre por ejemplo para $\mathtt{e} \equiv$ `(\x -> x x) (\x -> x x)`. Además, el que la reducción de una expresión termine o no depende de la estrategia de reducción empleada. Como ya se ha dicho en la sección 1.2, la *evaluación perezosa* es la estrategia de reducción que elige en cada paso un redex lo más externo posible, cuya reducción sea necesaria para que el cómputo progrese. También aceptamos sin demostración:

Propiedad de normalización de la reducción perezosa: Si una expresión `e` tiene forma normal, entonces la estrategia de reducción perezosa la calcula. Esta propiedad vale para la reducción general, no solo para la reducción β.

Por ejemplo, dada la expresión

```
(\y -> z) ((\x -> x x) (\x -> x x))
```

la evaluación perezosa calcula la forma normal `z` en un solo paso de reducción β:

`(\y -> z) ((\x -> x x) (\x -> x x))` $\longrightarrow_\beta$ `z`

En cambio, la estrategia de reducción que selecciona en cada paso un redex lo más interno posible, no temina para esta misma expresión:

`(\y -> z) ((\x -> x x) (\x -> x x))` $\longrightarrow_\beta$

`(\y -> z) ((\x -> x x) (\x -> x x)) ...`

Ejercicios

1. Dada la expresión `(\x -> \y -> y x) ((\z -> z y) x)`:
   (a) Señala los identificadores libres y los identificadores ligados.
   (b) Construye una variante de la expresión, en la cual ningún identificador aparezca libre y ligado a la vez.
   (c) Reduce la expresión a forma β-normal. ¿Es única la serie de pasos de reducción que llega a esta forma normal?

2. Comprueba que la forma normal de la expresión
   ```
   let diag = \f -> \x -> f x x in diag (*) 3
   ```
   es `9`. ¿Es única la secuencia de reducción que calcula la forma normal?

3. Comprueba que la forma normal de la expresión
   ```
   let {twice = \f -> \x -> f (f x); doble = \n -> n+n}
   in twice doble 2
   ```
   es `8`. ¿Es única la secuencia de reducción que calcula la forma normal?

4. Comprueba que la forma normal de la expresión
   ```
   let twice = \f -> \x -> f (f x) in twice twice
   ```
   es `\f -> \x -> f (f (f (f x)))`. ¿Es única la secuencia de reducción que calcula la forma normal?

1.5 Disciplina de tipos

Propósito de la disciplina de tipos

En los lenguajes de programación con disciplina de tipos, cada tipo representa una colección de valores (datos) similares. Una función cuyo tipo

sea `A1 -> ... An -> R` espera `n` parámetros con tipos `A1, ..., An` y devuelve un resultado de tipo `R`. El conocer los tipos de las funciones ayuda a documentar los programas y a evitar errores en tiempo de ejecución.

Haskell y otros lenguajes funcionales utilizan el *sistema de tipos de Milner*, que tiene dos características fundamentales:

- *Disciplina estática de tipos:* Los programas bien tipados se pueden reconocer en tiempo de compilación. Un programa bien tipado se puede utilizar sin efectuar comprobaciones de tipo en tiempo de ejecución, estando garantizado que no se producirán errores de tipo durante el cómputo. En particular, el tipo de los resultados calculados por programas bien tipados siempre es el previsto.

- *Polimorfismo:* Un programa bien tipado puede incluir definiciones de funciones polimórficas, como ya hemos visto en la sección 1.3. El polimorfismo permite que una misma función se pueda aplicar a parámetros de diferentes tipos, dependiendo del contexto en el que la función se utilice.

En esta sección vamos a estudiar las *reglas de inferencia de tipos*, que permiten calcular el tipo de las expresión dada, siempre que la expresión admita tipo en el sistema de Milner. Estudiaremos además las *clases de tipos*, una extensión muy útil del sistema de tipos de Milner, que está soportada por el lenguaje Haskell.

Inferencia de tipos

El problema de la *inferencia de tipos* se puede formular del siguiente modo: calcular el tipo `T` de una expresión dada `e`, suponiendo que los identificadores $\texttt{x}_i$ que aparezcan libres en `e` tengan tipos $\texttt{T}_i$. Vamos a utilizar la notación $\Gamma \vdash$ `e :: T` para indicar que es posible inferir el tipo `T` para la expresión `e`, bajo los supuestos $\texttt{x}_i$ `::` $\texttt{T}_i$ incluidos en el conjunto Γ. Por ejemplo:

(a) $\{$`x :: Int, 0 :: Int, (>) :: Int -> Int -> Bool`$\}$
$\vdash$ `x > 0 :: Bool`

(b) $\{$`x :: a`$\} \vdash$ `(x,x) :: (a,a)`

(c) $\{\ \} \vdash$ `\x -> x ::` $\forall$`a (a -> a)`

En el item (c) del ejemplo, la cuantificación universal $\forall$`a` indica que en el tipo inferido `a -> a` el parámetro `a` se puede sustituir por cualquier tipo concreto. En cambio, en el item (b) el tipo inferido no se puede generalizar a $\forall$`a (a,a)`, porque la inferencia *depende del supuesto* de que se tenga `x :: a`.

El ejemplo motiva que al formular reglas de inferencia de tipos es necesario indicar explítamente (por medio de una cuantificación universal) aquellos parámetros de un tipo que se pueden particularizar reemplazándolos por tipos concretos cualesquiera. Con esta motivación, en el apartado siguiente vamos a distinguir entre los *tipos simples* ya introducidos en la sección 1.3, y los *tipos genéricos* que incluyen parámetros cuantificados universalmente.

En el resto de esta sección supondremos además expresiones con la sintaxis del cálculo λ aplicado, explicada en la sección 1.4.

Tipos simples y genéricos

Llamaremos *tipo simple* a cualquier tipo `T` que se pueda formar utilizando las siguientes reglas en notación BNF:

```
T ::=   a
      | TP
      | (T1, ..., Tn)
      | (T1 -> T)
```

Las reglas anteriores ya han sido presentadas y explicadas en la sección 1.3, donde los tipos simples se llamaban "tipos" a secas. Recordemos que al escribir tipos simples se abrevia la sintaxis aceptando el convenio de que `->` asocia por la derecha. Por ejemplo, lo que sigue son tres tipos simples escritos en notación abreviada:

```
(Int,Bool)   (a,b)   Int -> Int -> Bool   (a -> a) -> a -> a
```

Por otra parte, llamaremos *tipo genérico* a cualquier tipo `S` que tenga una de las dos formas siguientes:

- Un tipo simple `T`.
- Un tipo de la forma ∀`a S1`, siendo `S1` otro tipo genérico.

Es decir, un tipo genérico siempre es de la forma ∀$\mathtt{a}_1$ ... ∀$\mathtt{a}_n$ `T`, siendo $n \geq 0$ y `T` un tipo simple. En un tipo genérico `S`, las cuantificaciones universales ∀$\mathtt{a}_i$ indican parámetros $\mathtt{a}_i$ que se pueden reemplazar por tipos simples cualesquiera, mientras que los parámetros que aparezcan en `S` sin estar afectados por ninguna cuantificación universal se suponen "congelados". Por ejemplo, los siguientes tipos son genéricos, pero no simples:

```
∀a a -> a            ∀a (a -> a) -> a -> a

∀a (a,b) -> a        ∀a ∀b (a,b) -> (b,a)
```

En lo que sigue, usaremos las notaciones $\mathcal{TS}$ y $\mathcal{TG}$ para referirnos al conjunto de todos los tipos simples y al conjunto de todos los tipos genéricos, respectivamente. Obsérvese que $\mathcal{TS} \subset \mathcal{TG}$, ya que cualquier tipo simple es un caso particular de tipo genérico, pero no a la inversa.

Los *parámetros libres* de un tipo genérico `S1` son aquellos parámetros `a` que aparezcan dentro de `S1` sin estar afectados por una cuantificación $\forall$`a`. Usaremos la notación `S1 [a` $\mapsto$ `T]` para indicar el resultado de sustituir en el tipo genérico `S1` todas las apariciones libres del parámetro `a` por el tipo simple `T`, renombrando si es necesario los parámetros cuantificados en `S1` para evitar que los parámetros de `T` queden capturados por una cuantificación después de la sustitución.

La operación de sustitución la vamos a utilizar para construir casos particulares de un tipo genérico dado. Más concretamente, siempre que tengamos un tipo genérico `S` $\equiv \forall$`a S1` y un tipo simple `T`, entenderemos que la sutitución `S1 [a` $\mapsto$ `T]` nos da un tipo que es un *caso particular* de `S`. Por ejemplo, considerando `S` $\equiv \forall$`a` $\forall$`b (a,b) -> (b,a)`, resulta que:

- $\forall$`b (a,b) -> (b,a) [a` $\mapsto$ `Int]` $\equiv$
  $\forall$`b (Int,b) -> (b,Int)` es un caso particular de `S`.
- $\forall$`b (a,b) -> (b,a) [a` $\mapsto$ `(b,b)]` $\equiv$
  $\forall$`c ((b,b),c) -> (c,(b,b))` es otro caso particular de `S`.

Reglas de inferencia de tipos

El tipo de una expresión dada `e` depende de los tipos que se supongan para los identificadores que aparezcan libres en `e`. En lo que sigue, llamaremos *contexto* a cualquier conjunto finito Γ formado por supuestos de tipo $\mathtt{x}_i$ `::` $\mathtt{S}_i$, que no contenga varios supuestos diferentes para un mismo identificador $\mathtt{x}_i$. Usaremos la notación $\Gamma \vdash$ `e :: S` para indicar que a partir del contexto Γ se puede deducir que `e` tiene tipo `S`, utilizando la notación abreviada $\vdash$ `e :: S` en el caso de que se tenga un contexto vacío $\Gamma = \emptyset$.

Para deducir cuando se verifica $\Gamma \vdash$ `e :: S`, usaremos las reglas de inferencia de tipos que vamos a estudiar a continuación. Antes de formularlas, necesitamos dos definiciones auxiliares:

- Un parámetro `a` *está libre en un contexto* Γ syss `a` aparece libre en $\mathtt{S}_i$ para alguno de los supuestos $\mathtt{x}_i$ `::` $\mathtt{S}_i$ pertenecientes a Γ.
- Dados dos contextos Γ y Γ', su *combinación* $\Gamma \oplus \Gamma'$ se construye como la unión de $\{$ $\mathtt{x}_i$ `::` $\mathtt{S}_i \in \Gamma$ | $\mathtt{x}_i$ no aparece en Γ' $\}$ con Γ'.

Ahora ya estamos en condiciones de formular las *reglas de inferencia de tipos*. Cada una de ellas va a ser de la forma

$$\frac{\Gamma_1 \vdash \mathtt{e}_1 \ :: \ \mathtt{S}_1 \qquad \cdots \qquad \Gamma_n \vdash \mathtt{e}_n \ :: \ \mathtt{S}_n}{\Gamma \vdash \mathtt{e} \ :: \ \mathtt{S}}$$

con n *premisas* $\Gamma_i \vdash \mathtt{e}_i \ :: \ \mathtt{S}_i$ $(n \geq 0)$ y una *conclusión* $\Gamma \vdash \mathtt{e} \ :: \ \mathtt{S}$. En el caso $n = 0$ no escribiremos las premisas. Para cada una de las clases de expresiones previstas por la sintaxis del cálculo λ aplicado (ver sección 1.4) hay una regla cuya conclusión corresponde a una expresión de esa forma. Estas reglas se muestran a continuación, junto con algunas reglas auxiliares. En cada regla, convenimos en que las letras $\mathtt{T}$, $\mathtt{T}_i$ indican tipos simples, mientras que $\mathtt{S}$, $\mathtt{S}_i$ indican tipos genéricos.

(IDE) Identificador

$$\Gamma \vdash \mathtt{x} \ :: \ \mathtt{S}$$
siempre que $\mathtt{x} \ :: \ \mathtt{S} \in \Gamma$

(AP) Aplicación

$$\frac{\Gamma \vdash \mathtt{e}_1 \ :: \ \mathtt{T}_1 \qquad\qquad \Gamma \vdash \mathtt{e} \ :: \ \mathtt{T}_1 \ \mathtt{->} \ \mathtt{T}}{\Gamma \vdash \mathtt{e} \ \mathtt{e}_1 \ :: \ \mathtt{T}}$$

La regla **(APn)** formulada a continuación no es imprescindible, pero resulta muy útil para abreviar la aplicación reiterada de la regla **(AP)** `n` veces consecutivas:

(APn) Aplicación con n parámetros

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_i \ :: \ \mathtt{T}_i \ \ (1 \leq i \leq n) \\ \Gamma \vdash \mathtt{e} \ :: \ \mathtt{T}_1 \ \mathtt{->} \ \ldots \ \mathtt{->} \ \mathtt{T}_n \ \mathtt{->} \ \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{e} \ \mathtt{e}_1 \ \ldots \ \mathtt{e}_n \ :: \ \mathtt{T}}$$

(ABS) Abstracción (función anónima)

$$\frac{\Gamma \oplus \{\mathtt{x} \ :: \ \mathtt{T}_1\} \vdash \mathtt{e} \ :: \ \mathtt{T}}{\Gamma \vdash \mathtt{\backslash x \ -> \ e} \ :: \ \mathtt{T}_1 \ \mathtt{->} \ \mathtt{T}}$$

La siguiente regla es útil para abreviar la aplicación reiterada de **(ABS)** `n` veces consecutivas:

(ABSn) Abstracción reiterada n veces

$$\frac{\Gamma \oplus \{\mathtt{x}_1 \ :: \ \mathtt{T}_1, \ \ldots, \ \mathtt{x}_n \ :: \ \mathtt{T}_n\} \vdash \mathtt{e} \ :: \ \mathtt{T}}{\Gamma \vdash \mathtt{\backslash x_1 \ -> \ \ldots \ -> \backslash x_n \ -> \ e} \ :: \ \mathtt{T}_1 \ \mathtt{->} \ \ldots \ \mathtt{->} \ \mathtt{T}_n \ \mathtt{->} \ \mathtt{T}}$$
siempre que los parámetros formales $\mathtt{x}_i$ $(1 \leq i \leq n)$ sean todos diferentes

(TUP) Tupla

$$\frac{\Gamma \vdash \mathtt{e}_1 \ :: \ \mathtt{T}_1 \qquad \cdots \qquad \Gamma \vdash \mathtt{e}_n \ :: \ \mathtt{T}_n}{\Gamma \vdash (\mathtt{e}_1, \cdots, \mathtt{e}_n) \ :: \ (\mathtt{T}_1, \cdots, \mathtt{T}_n)}$$

(IF) Expresión condicional

$$\frac{\Gamma \vdash \mathtt{e} \ :: \ \mathtt{Bool} \qquad \Gamma \vdash \mathtt{e}_1 \ :: \ \mathtt{T} \qquad \Gamma \vdash \mathtt{e}_2 \ :: \ \mathtt{T}}{\Gamma \vdash \mathtt{if\ e\ then\ e}_1 \ \mathtt{else\ e}_2 \ :: \ \mathtt{T}}$$

(LET) Definición local

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_i \ :: \ \mathtt{S}_i \ \ (1 \leq i \leq n) \\ \Gamma \oplus \{\mathtt{x}_1 \ :: \ \mathtt{S}_1, \cdots, \mathtt{x}_n \ :: \ \mathtt{S}_n\} \vdash \mathtt{e} \ :: \ \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{let}\ \{\mathtt{x}_1 = \mathtt{e}_1; \cdots; \mathtt{x}_n = \mathtt{e}_n\}\ \mathtt{in\ e} \ :: \ \mathtt{T}}$$

(LETREC) Definición local recursiva

$$\frac{\begin{array}{c}\Gamma \oplus \{\mathtt{x}_1 \ :: \ \mathtt{S}_1, \cdots, \mathtt{x}_n \ :: \ \mathtt{S}_n\} \vdash \mathtt{e}_i \ :: \ \mathtt{S}_i \ \ (1 \leq i \leq n) \\ \Gamma \oplus \{\mathtt{x}_1 \ :: \ \mathtt{S}_1, \cdots, \mathtt{x}_n \ :: \ \mathtt{S}_n\} \vdash \mathtt{e} \ :: \ \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{letrec}\ \{\mathtt{x}_1 = \mathtt{e}_1; \cdots; \mathtt{x}_n = \mathtt{e}_n\}\ \mathtt{in\ e} \ :: \ \mathtt{T}}$$

Las siguientes reglas auxiliares sirven para manejar los tipos genéricos. El superíndice "†" que acompaña a la regla (GEN) advierte que esta regla solo puede aplicarse cuando se cumple la condición de que `a` no esté libre en Γ. Si esta condición no se satisface, la aplicación de (GEN) puede inferir a tipos incorrectos.

(PART) Particularización

$$\frac{\Gamma \vdash \mathtt{e} \ :: \ \forall \mathtt{a}\ \mathtt{S}}{\Gamma \vdash \mathtt{e} \ :: \ \mathtt{S}\ [\mathtt{a} \mapsto \mathtt{T}]}$$

para cualquier $\mathtt{T} \in \mathcal{T}S$

(GEN)† Generalización

$$\frac{\Gamma \vdash \mathtt{e} \ :: \ \mathtt{S}}{\Gamma \vdash \mathtt{e} \ :: \ \forall \mathtt{a}\ \mathtt{S}}$$

siempre que `a` no esté libre en Γ

Uso de las reglas de inferencia de tipos

A continuación, mostramos algunos ejemplos de inferencia de tipos. En cada caso, se trata de demostrar una afirmación de la forma $\Gamma \vdash \mathtt{e} \ :: \ \mathtt{S}$ en varios pasos, utilizando las reglas que acabamos de formular. Los dos

primeros ejemplos ilustran el uso de las reglas más básicas.

Ejemplo 1. Para el contexto Γ = {`(+) :: Int -> Int -> Int`}, se verifica que $\Gamma \vdash$ `\x -> x+x :: Int -> Int`.

Demostración: Razonamos teniendo en cuenta que la expresión `x+x` equivale a `(+) x x`.

1. $\Gamma \vdash$ `\x -> x+x :: Int -> Int` (ABS), 2.
2. $\Gamma' = \Gamma \oplus$ {`x :: Int`} $\vdash$ `(+) x x :: Int` (AP2), 3, 3, 4.
3. $\Gamma' \vdash$ `x :: Int` (IDE).
4. $\Gamma' \vdash$ `(+) :: Int -> Int -> Int` (IDE).

Ejemplo 2. $\vdash$ `\f -> \x -> f (f x) :: ∀a (a -> a) -> a -> a`.

Demostración: Similar a la anterior, pero utilizando la regla (GEN), ya que el tipo derivado es genérico.

1. $\vdash$ `\f -> \x -> f (f x) :: ∀a (a -> a) -> a -> a` (GEN), 2.
2. $\vdash$ `\f -> \x -> f (f x) :: (a -> a) -> a -> a` (ABS2), 3.
3. $\Gamma' = \Gamma \oplus$ {`f :: a -> a, x :: a`} $\vdash$ `f (f x) :: a` (AP), 4, 5.
4. $\Gamma' \vdash$ `f x :: a` (AP), 6, 5.
5. $\Gamma' \vdash$ `f :: a -> a` (IDE).
6. $\Gamma' \vdash$ `x :: a` (IDE).

El ejemplo siguiente muestra que la regla (LET) soporta el polimorfismo, ya que permite derivar tipos genéricos S_i para los identificadores x_i definidos localmente en una expresión `let`, y particularizarlos según convenga para derivar el tipo de la expresión `e` que se tenga como cuerpo del `let`. En el caso de este ejemplo, se deriva el tipo genérico `∀a a -> a` para `id`, que luego se particulariza de dos maneras diferentes para las dos apariciones de `id` en el cuerpo de la expresión `let`.

Ejemplo 3. Para el contexto Γ = {`2 :: Int, True :: Bool`}, se verifica que $\Gamma \vdash$ `let id = \x -> x in (id 2,id True) :: (Int,Bool)`.

Demostración:

1. $\Gamma \vdash$ `let id = \x -> x in (id 2,id True) :: (Int,Bool)` (LET), 2, 3.
2. $\Gamma \vdash$ `\x -> x :: ∀a a -> a` (GEN), 4.
3. $\Gamma' = \Gamma \oplus$ {`id :: ∀a a -> a`} $\vdash$ `(id 2,id True) :: (Int,Bool)` (TUP), 6,7.
4. $\Gamma \vdash$ `\x -> x :: a -> a` (ABS), 5.

5. $\Gamma'' = \Gamma \oplus \{\texttt{x :: a}\} \vdash \texttt{x :: a}$ (IDE).
6. $\Gamma' \vdash \texttt{id 2 :: Int}$ (AP), 8,9.
7. $\Gamma' \vdash \texttt{id True :: Bool}$ (AP), 11,12.
8. $\Gamma' \vdash \texttt{2 :: Int}$ (IDE).
9. $\Gamma' \vdash \texttt{id :: Int -> Int}$ (PART) con $\texttt{a} \mapsto \texttt{Int}$, 10.
10. $\Gamma' \vdash \texttt{id :: } \forall \texttt{a a -> a}$ (IDE).
11. $\Gamma' \vdash \texttt{True :: Bool}$ (IDE).
12. $\Gamma' \vdash \texttt{id :: Bool -> Bool}$ (PART) con $\texttt{a} \mapsto \texttt{Bool}$, 10.

Finalmente, mostramos un ejemplo sencillo que ilustra el uso de la regla (LETREC).

Ejemplo 4. Suponiendo que Γ incluya los supuestos de tipo siguientes

```
(>) :: Int -> Int -> Bool
(*) :: Int -> Int -> Int
(-) :: Int -> Int -> Int
0 :: Int
1 :: Int
2 :: Int
3 :: Int
```

se tiene que

$\Gamma \vdash$ `letrec fact = \n -> if n > 0 then n*fact(n-1) else 1`
`in fact 3 :: Int`

Demostración: Según la regla (LETREC), para obtener la afirmación propuesta es suficiente demostrar otras dos afirmaciones:

1. $\Gamma \oplus \{\texttt{fact :: Int -> Int}\} \vdash$
   `\n -> if n > 0 then n*fact(n-1) else 1 :: Int -> Int`
2. $\Gamma \oplus \{\texttt{fact :: Int -> Int}\} \vdash \texttt{fact 3 :: Int}$

las cuales se demuestran fácilmente, usando razonamientos similares a los que han aparecido en los ejemplos anteriores.

Aunque el ejemplo anterior se refiere a una función monomórfica, la regla (LETREC) también soporta el polimorfismo, lo mismo que la regla (LET). Como dijimos al comienzo de esta sección, el polimorfismo es una de las carcaterísticas más más importantes del sistema de tipos de Milner. El polimorfismo tiene un gran valor práctico, porque permite programar funciones genéricas y reutilizables. En el resto de este curso irán apareciendo muchas funciones polimórficas interesantes, aunque por ahora solo hayamos visto ejemplos simples.

Existencia de expresiones no tipables

No todas las expresiones admiten tipo en el sistema de Milner. Seguidamente mostramos dos contraejemplos. En cada caso, veremos que las reglas de inferencia de tipos no permiten deducir un tipo para la expresión propuesta.

Contraejemplo 1. La expresión `\x -> (suc x, not x)` no admite tipo en el contexto Γ = {`suc :: Int -> Int, not :: Bool -> Bool`}.

Demostración: Si esta expresión admitiese tipo en este contexto, deberían existir tipos `T`, $\mathtt{T}_1$ y $\mathtt{T}_2$ tales que

$\Gamma \vdash$ `\x -> (suc x, not x) :: T -> (`$\mathtt{T}_1$`,`$\mathtt{T}_2$`)`

Demostrar lo anterior solo sería posible aplicando la regla (ABS) y demostrando

$\Gamma \oplus$ {`x :: T`} $\vdash$ `(suc x, not x) :: (`$\mathtt{T}_1$`,`$\mathtt{T}_2$`)`

Para deducir ésto, sería necesario aplicar (TUP) y demostrar

1. $\Gamma \oplus$ {`x :: T`} $\vdash$ `suc x ::` $\mathtt{T}_1$

2. $\Gamma \oplus$ {`x :: T`} $\vdash$ `not x ::` $\mathtt{T}_2$

Tal como son los supuestos incluidos en Γ, 1. solo se podría demostrar si ocurriese que `T` $\equiv$ $\mathtt{T}_1$ $\equiv$ `Int`; y 2. solo se podría demostrar si fuese cierto que `T` $\equiv$ $\mathtt{T}_2$ $\equiv$ `Bool`. Como `T` no puede ser al mismo tiempo `Int` y `Bool`, podemos concluir que efectivamente la expresión dada no admite tipo en el contexto dado.

Contraejemplo 2. La expresión `\x -> x x` no admite tipo en el contexto vacío.

Demostración: Si `\x -> x x` admitiese tipo en el contexto vacío, debería admitir un tipo simple de la forma $\mathtt{T}_1$ `-> T`, y la demostración de la afirmación $\vdash$ `\x -> x x ::` $\mathtt{T}_1$ `-> T` debería tener la estructura siguiente:

1. $\vdash$ `\x -> x x ::` $\mathtt{T}_1$ `-> T` (ABS), 2.
2. {`x ::` $\mathtt{T}_1$} $\vdash$ `x x :: T` (AP), 3., 4.
3. {`x ::` $\mathtt{T}_1$} $\vdash$ `x ::` $\mathtt{T}_1$ (IDE).
4. {`x ::` $\mathtt{T}_1$} $\vdash$ `x ::` $\mathtt{T}_1$ `-> T` (IDE).

Ahora bien, una demostración así solo se podría construir si se cumpliese que $\mathtt{T}_1 \equiv \mathtt{T}_1$ `->` `T`, lo cual es imposible. Por lo tanto, `\x -> x x` no admite tipo en el contexto vacío. Como consecuencia, tampoco admite tipo la expresión `(\x -> x x) (\x -> x x)`, como ya habíamos advertido en la sección 1.4.

Tipo principal de una expresión

Una expresión que admita tipo en el sistema de Milner puede admitir a veces varios tipos. Por ejemplo, la expresión `\x -> x` que representa la función identidad, admite infinitos tipos diferentes en el contexto vacío; entre otros:

$\vdash$ `\x -> x :: ∀a a -> a`
$\vdash$ `\x -> x :: Int -> Int`
$\vdash$ `\x -> x :: ∀a ∀b (a,b) -> (a,b)`

Intuitivamente, el primero de los tres tipos anteriores es más general que los otros dos. Para precisar cuando se considera que un tipo es más general que otro, damos la siguiente definición:

Tipo más general: Dados dos tipos genéricos `S` y `S'`, diremos que `S` es *más general* que `S'` si se puede demostrar que $\{$`x :: S`$\}$ $\vdash$ `x :: S'`, siendo `x` cualquier identificador.

Por ejemplo, `∀a a -> a` es más general que `∀a ∀b (a,b) -> (a,b)`, ya que:

1. $\{$`x :: ∀a a -> a`$\}$ $\vdash$ `x :: ∀a ∀b (a,b) -> (a,b)` (GEN), 2.
2. $\{$`x :: ∀a a -> a`$\}$ $\vdash$ `x :: ∀b (a,b) -> (a,b)` (GEN), 3.
3. $\{$`x :: ∀a a -> a`$\}$ $\vdash$ `x :: (a,b) -> (a,b)` (PART) con `a` $\mapsto$ `(a,b)`, 4.
4. $\{$`x :: ∀a a -> a`$\}$ $\vdash$ `x :: ∀a a -> a` (IDE).

Cualquier expresión que admita tipo en el sistema de Milner tiene un tipo más general que todos los demás. Esto se sabe gracias al siguiente teorema, que admitimos sin demostración:

Teorema de existencia de tipo principal: Siempre que la expresión `e` admita algún tipo en el contexto Γ, existe un tipo genérico `S`, llamado *tipo principal* de `e`, que verifica:

1. $\Gamma \vdash$ `e :: S`.
2. `S` es más general que cualquier otro `S'` tal que $\Gamma \vdash$ `e :: S'`.

Inferencia de tipos en Haskell

Las lenguajes de programación con disciplina estática de tipos siempre comprueban que no hay errores de tipos en un programa antes de ejecutarlo. En particular, las implementaciones de Haskell utilizan un algoritmo basado en las reglas de inferencia de tipos de Milner, con el fin de decidir si las funciones definidas dentro de un programa dado admiten tipo. En caso afirmativo, el sistema infiere el tipo principal. En el caso de que el usuario haya incluido declaraciones de tipo en el programa, el sistema se encarga de comprobar que ésta sean consistentes con los tipos inferidos. Si el usuario no ha incluido declaraciones de tipo en el programa, el sistema las infiere por sí mismo.

Además, dada cualquier expresión `e`, un usuario de Haskell puede solicitar el cálculo del tipo principal de `e` mediante la orden `:type e`, que se puede abreviar como `:t e`. Para ejecutar dicha orden, el sistema aplica un algoritmo basado en las reglas de inferencia de tipos a la expresión `letrec` $\{$`def`$_1$`; ...; def`$_n\}$ `in e`, donde `def`$_i$ representan las definiciones incluidas en el programa que esté cargado en el momento de ejecutarse la orden.

Advertencia: Los tipos calculados por Haskell son genéricos, pero el sistema los muestra sin hacer explícita la cuantificación universal de los parámetros. En cualquier tipo `T` que aprezca declarado en el texto de un programa Haskell, o que sea calculado mediante la orden `:type`, hay que entender que todos los parámetros que aparezcan en `T` llevan una cuantificacción universal implícita. Por ejemplo, el intérprete Hugs 98 calcula el tipo principal de la expresión `\x -> x` como sigue:

```
Prelude> :type \x -> x
\x -> x :: a -> a
```

y hay que entender que se trata del tipo genérico ∀`a a -> a`.

Clases de tipos y tipos cualificados

En uno de los ejemplos de esta sección hemos supuesto que la operación de orden tiene tipo `(>) :: Int -> Int -> Bool`. Evidentemente, tiene sentido admitir `(>) :: T -> T -> Bool` para algunos otros tipos `T` diferentes de `Int`. Por otro lado, es deseable que `(>)` tenga un tipo principal, y éste no puede ser `(>) :: ∀a a -> a -> Bool` porque no es cierto que *cualquier* tipo `a` admita una operación de orden.

Para resolver este problema, Haskell admite que el tipo principal de la operación `(>)` es ∀`a Ord a => a -> a -> Bool`. Esto se interpreta como la afirmación de que `(>) :: a -> a -> Bool` se cumple para cualquier tipo `a` que pertenezca a la clase de tipos `Ord`, la cual agrupa a todos los tipos que disponen de operaciones de orden.

En general, Haskell permite agrupar en una *clase de tipos* $\mathcal{C}$ a una familia de tipos que tengan en común el disponer de una serie de operaciones, llamadas *métodos de la clase*, que se nombran con los mismos símbolos o identificadores para todos los tipos que pertenezcan a la clase. Y se permite utilizar *tipos cualificados* de la forma

$\forall$`a`$_1$ ... $\forall$`a`$_n\forall$`b`$_1$ ... $\forall$`b`$_m$ (`C`$_1$ `a`$_1$, ..., `C`$_n$ `a`$_n$) => `T`

siendo $n, m \geq 0$, a_i, b_j parámetros diferentes, $\mathcal{C}_i$ clases de tipos y `T` un tipo simple. Un tipo cualificado como éste indica que cada parámetro `a`$_i$ se puede reemplazar por cualquier tipo simple que pertenezca a la clase de tipos $\mathcal{C}_i$, mientras que los parámetros b_j se pueden reemplazar por tipos simples arbitrarios. En la práctica, Haskell permite escribir tipos cualificados sin indicar explícitamente la cuantificación universal.

Las reglas de derivación de tipos del sistema de Milner se pueden extender para tratar con tipos cualificados, de tal manera que la propiedad de existencia de tipo principal se preserva para cualquier expresión que admita tipo en el sistema extendido. En el resto de esta sección, vamos a dar algunas indicaciones acerca de la definición de clases de tipos y el uso de tipos cualificados en Haskell.

Declaraciones de clase y de ejemplar

Las clases de tipos se definen mediante *declaraciones de clase*, de la forma

```
class CL a where

    ...
```

La palabra reservada `class` encabeza la declaración de clase, y `CL` debe se un identificador que comience por mayúscula, que nombra la clase declarada. Detrás de la palabra reservada `where` deben aparecer las *declaraciones de los métodos* propios de la clase. La declaración de cada método se compone de los siguientes elementos:

1. El símbolo o identificador que sirve de *nombre* del método, junto con el tipo de los parámetros y el resultado del método, y un comentario que explique el comportamiento esperado del método.

2. Opcionalmente, una *definición genérica* del método, escrita utilizando los recursos estudiados en la sección 1.3 para construir definiciones de funciones.

3. Un comentario que especifique un *conjunto suficiente* de métodos de la clase, diseñado de tal manera que incluya todos aquellos métodos que

no tengan una definición genérica especificada en la propia declaración de clase. Como veremos más abajo, cada ejemplar concreto de la clase debe aportar definiciones específicas para los métodos de este conjunto suficiente.

Por ejemplo, la clase `Eq` de los *tipos con igualdad* se puede definir por medio de la siguiente declaración:

```
class Eq a where

 -- (==) debe ser una relacion de equivalencia
 -- (/=) debe ser la negacion de (/=)

    (==), (/=) :: a -> a -> Bool
    x == y      = not (x /= y)
    x /= y      = not (x == y)

-- Conjunto suficiente de metodos: (==) o (/=)
```

La declaración anterior está predefinida en el preludio de Haskell, y no es necesario que los usuarios la incluyan en sus programas. La declaración indica que los ejemplares de `Eq` son tipos que disponen de los métodos `(==)` y `(/=)`. Los nombres de estos métodos están *sobrecargados*, porque son los mismos para todos los ejemplares de la clase. Los comportamientos de `(==)` y `(/=)` pueden ser diferentes para cada ejemplar de la clase, aunque se espera que siempre se comporten como una relación de equivalencia y su negación, respectivamente. Se espera además que cada ejemplar de `Eq` se declare aportando una definición específica de uno de los dos métodos `(==)` o `(/=)`; para el otro se utilizará la definición genérica incluida en la propia clase.

En general, los nombres de los métodos de cualquier clase de tipos siempre están *sobrecargados* en el sentido de que se refieren a diferentes funciones con distinto comportamiento en los distintos ejemplares de la clase. Dichos ejemplares deben declararse escribiendo *declaraciones de ejemplar* con la siguiente sintaxis:

```
instance CL EJ where

    ...
```

La palabra reservada `instance` encabeza la declaración de ejemplar, y los identificadores `CL` y `EJ` (ambos comenzando por mayúscula) nombran, respectivamente, la clase y el ejemplar que se está declarando. La palabra reservada `where` debe ir seguida por las definiciones de un *conjunto suficiente* de métodos de la clase. Para aquellos métodos de la clase `CL` que no

figuren en este conjunto, el ejemplar `EJ` utilizará las definiciones genéricas especificadas en la declaración de la clase.

Por ejemplo, la siguiente declaración (predefinida en el preludio de Haskell) convierte al tipo `Bool` en ejemplar de la clase `Eq`:

```
instance Eq Bool where
  False == False  = True
  False == True   = False
  True  == False  = False
  True  == True   = True
```

La declaración anterior ha elegido definir el método `(==)`, que constituye un por sí solo conjunto suficiente de métodos de la clase. Puesto que lo mismo ocurre para el método `(/=)`, la siguiente declaración de ejemplar también es legal, y causa un comportamiento equivalente a la anterior:

```
instance Eq Bool where
  False /= False  = False
  False /= True   = True
  True  /= False  = True
  True  /= True   = False
```

Subclases y herencia

Una clase de tipos puede ser *subclase* de otras. En este caso, los ejemplares de la subclase heredan todos los métodos de las superclases, junto con nuevos métodos propios de la subclase. La declaración de una subclase tiene la forma

```
class (CL1 a, ..., CLn a) => CL a where

    ...
```

donde el identificador `CL` nombra la subclase, los identificadores `CLi` nombran las superclases, y la palabra `where` va seguida de las declaraciones de los métodos propios de `CL`, especificando sus nombres, sus tipos, su comportamiento esperado, y un conjunto suficiente de entre ellos.

Por ejemplo, la siguientes declaraciones (predefinidas en el preludio de Haskell) introducen la clase `Ord` de los *tipos con orden* como subclase de `Eq`. El tipo `Ordering` empleado en esta declaración es un tipo predefinido de la clase `Eq`; contiene tres valores `EQ`, `LT` y `GT`, que deben interpretarse como "igual", "menor que" y "mayor que", respectivamente.

```
class Eq a => Ord a where

-- Metodo de comparacion:

   compare       :: a -> a -> Ordering
   compare x y
     | x == y     = EQ
     | x <= y     = LT
     | otherwise  = GT

-- Relaciones de orden total:

   (>=), (<), (>=), (>) :: a -> a -> Bool
   x <= y                = compare x y /= GT
   x <  y                = compare x y == LT
   x >= y                = compare x y /= LT
   x >  y                = compare x y == GT

--  Metodos para el calculo de maximos y minimos:

   max, min             :: a -> a -> a
   x 'min' y  = if x <= y then x else y
   x 'max' y  = if x <= y then y else x

--  Conjunto suficiente de metodos: (<=) o compare
```

Debido al hecho de que las ecuaciones de un programa siempre se ensayan en orden textual, la definición genérica del método `compare` formulada más arriba se comporta del mismo modo que la siguiente definición:

```
compare x y
  | x == y     = EQ
  | x <  y     = LT
  | x >  y     = GT
```

Aunque el sentido lógico de esta segunda definición es más claro, la definición presentada más arriba es la que se usa en la práctica, debido a que puede ser utilizada por aquellos ejemplares de `Ord` que opten por declarar una definición específica de `(<=)`. De este modo se consigue de que cualquiera de los dos métodos `compare` o `(<=)` forme por sí solo un conjunto suficiente de métodos para la clase.

Los ejemplares de una clase declarada como subclase de otras se declaran con la misma sintaxis que ya hemos explicado, pero deben haber sido declarados previamente como ejemplares de las superclases. Por ejemplo, la declaración de `Bool` como ejemplar de la clase `Ord` (predefinida en el preludio) se puede escribir de una de las dos maneras indicadas a continuación, según que se decida definir el método `(<=)` o el método `compare`:

```
instance Ord Bool where
  False <= False  = True
  False <= True   = True
  True  <= False  = False
  True  <= True   = True
```

o bien:

```
instance Ord Bool where
  compare False False  = EQ
  compare False True   = LT
  compare True  False  = GT
  compare True  True   = EQ
```

En las dos declaraciones anteriores se ha tomado la decisión arbitraria de considerar que `False` precede a `True`. En general, al construir una declaración de ejemplar para `Ord` siempre hay que optar por definir `(<=)` o `compare`, de forma que resulte el comportamiento de un orden total.

Ejemplos de uso de tipos cualificados

Cualquier identificador que nombre uno de los métodos de una clase de tipos tiene siempre un tipo cualificado, como ocurre por ejemplo en los casos

```
(==) :: Eq a  => a -> a -> Bool

(<=) :: Ord a => a -> a -> Bool
```

Más en general, cualquier función cuya definición dependa de métodos propios de ciertas clases de tipos, tendrá un tipo cualificado. Esto ocurre en los dos ejemplos siguientes:

```
capricho       :: Ord a => a -> b -> a -> (a,b)
capricho x y z
  | x <= z      = (z,y)
  | x  > z      = (x,y)

veleidad         :: (Eq a, Ord b) => a -> a -> b -> b -> Bool
veleidad x y u v
  | x == y        = u < v
  | x /= y        = u > v
```

Los ejemplos muestran que un tipo cualificado puede tener algunos parámetros completamente genéricos (como el parámetro `b` en el tipo de `capricho`) y otros restringidos de manera que solo se pueden sustituir por ejemplares de ciertas clases de tipos. Nótese que esto permite una cierta reutilización de código, aunque restringida por la cualificación de los tipos. En capítulos

posteriores encontraremos ejemplos más interesantes de programación con tipos cualificados.

Algunas clases de tipos predefinidas en Haskell

Haskell dispone de un cierto número de clases de tipos predefinidas, cuyo conocimiento es útil para la programación. Las más importantes se resumen a continuación, indicando en cada caso el nombre de la clase y algunos de sus métodos, pero sin dar detalles acerca del modo de declarar la clase y sus ejemplares. Algunas de estas clases las estudiaremos más detalladamente en capítulos posteriores.

- `Eq`, `Ord`
  Ya explicadas.

- `Num`
  Los ejemplares de esta clase son los diferentes tipos numéricos. Los métodos propios de la clase incluyen:

```
(+), (*), (-) :: Num a => a -> a -> a
```

- `Enum`
  Los ejemplares de esta clase son tipos isomorfos a un intervalo formado por números enteros consecutivos. Los métodos propios de la clase incluyen:

```
fromEnum   :: Enum a => a -> Int
toEnum     :: Enum a => Int -> a
succ, pred :: Enum a => a -> a
```

- `Bounded`
  Los ejemplares de esta clase tienen dos valores extremos, dados por dos métodos sin parámetros:

```
minBound, maxBound :: Bounded a => a
```

- `Ix`
  Los ejemplares de esta clase son tipos cuyos valores se pueden utilizar como índices de vectores, como veremos en la sección 4.3.

- `Read`
  Los ejemplares de esta clase son tipos cuyos valores se pueden reconstruir a partir de una cadena de caracteres. El método principal de la clase es:

```
read :: Read a => String -> a
```

- Show
  Los ejemplares de esta clase son tipos cuyos valores se pueden representar en forma de cadena de caracteres. El método principal de la clase es:

```
show :: Show a => a -> String
```

Ejercicios

1. Usando las reglas de inferencia de tipos del sistema de Milner, demuestra que cada una de las expresiones siguientes admite el tipo indicado en el contexto dado:

   (a) $\Gamma \vdash$ `\x -> let y = x+x in x*y :: Int -> Int`
   siendo Γ = {`(+) :: Int -> Int -> Int`,
   `(*) :: Int -> Int -> Int`}

   (b) $\vdash$ `\f -> \g -> \x -> f x (g x)`
   `:: ∀a ∀b ∀c (a -> b -> c) -> (a -> b) -> a -> c`

   (c) $\vdash$ `let id = \x -> x in id id :: ∀a a -> a`

   (d) $\vdash$ `let twice = \f -> \x -> f (f x) in twice twice`
   `:: ∀a (a -> a) -> a -> a`

2. Suponiendo que `e` sea la expresión

```
letrec par    = \n -> if n < 2
                      then n
                      else impar (n-1)
       impar  = \n -> if n < 2
                      then 1-n
                      else par (n-1)
       in (par, impar)
```

   construye un contexto Γ adecuado, y usa las reglas de inferencia de tipos de Milner para demostrar $\Gamma \vdash$ `e :: (Int -> Int, Int -> Int)`.

3. En cada uno de los casos siguientes, encuentra el tipo principal de la expresión en el contexto vacío, y usa las reglas de inferncia de Milner para demostrar que el tipo es correcto.

   (a) `\k -> \x -> k`

   (b) `\f -> \x -> \y -> f y x`

(c) `\f -> \g -> \x -> \y -> f (g x y)`

4. La función `while` definida a continuación se puede usar para expresar declarativamente el cambio de estado producido por la ejecución de un bucle imperativo:

```
while test body state =
   if test state
      then while test body (body state)
      else state
```

   Deriva el tipo principal de `while` en el contexto vacío. Para ello, deberás reformular la definición de `while` utilizando una expresión `letrec`, de manera similar a lo que hemos hecho en el ejemplo 4. y en el ejercicio 2. de esta sección.

5. Con ayuda de la función `while` del ejercicio anterior, y sin usar más recursión, construye una definición ejecutable para la función `leastFrom` del ejercicio 13 de la sección 1.3.

6. Considera el siguiente programa imperativo:

```
var
  x, y, z :: Int;
begin
  z := x;
  while x <= y do
    z := z*x;
    x := x+1
  endwhile
end
```

   Usando la función `while` del ejercicio 4, define una función:

```
prog :: (Int,Int,Int) -> (Int,Int,Int)
```

   tal que `prog(u,v,w)` devuelva el estado final que produciría la ejecución del programa imperativo a partir de un estado inicial en el cual las variables `x, y, z` tengan valores `u, v, w`, respectivamente.

7. Por analogía con el ejercicio 4, define una función `ifThenElse` que sirva para expresar declarativamente el comportamiento de la estructura de control `if then else` de los lenguajes imperativos. Deriva el tipo principal de `ifThenElse` en el contexto vacío.

8. Define una función `ordenaTerna :: Ord a => (a,a,a) -> (a,a,a)` que devuelva el resultado de ordenar la terna dada como parámetro en orden no decreciente.

Capítulo 2

Tipos de datos

2.1 Valores numéricos y booleanos

Los datos más básicos para cualquier problema de programación son los valores numéricos y booleanos. En esta sección estudiamos los tipos predefinidos que ofrece el lenguaje Haskell para manejar dichos valores. Comenzamos considerando los principales tipos numéricos disponibles.

Como criterio general, hay que tener en cuenta que todos los tipos numéricos existentes en Haskell, así como el tipo de los booleanos, son ejemplares de las clases de tipos `Eq`, `Ord`, `Read` y `Show`. En particular, las operaciones de comparación `(==)`, `(/=)`, `(<=)`, `(<)`, `(>=)` y `(>)` se pueden utilizar para comparar valores de cualquier tipo numérico, así como valores booleanos. Las funciones `max` y `min` también se pueden aplicar a valores de todos estos tipos.

Números enteros

En Haskell existen dos tipos de números enteros: el tipo `Int` de los enteros de precisión limitada, y el tipo `Integer` de los enteros de precisión arbitraria. En este curso usaremos preferentemente `Int`. Los valores de este tipo comprenden los enteros negativos como `-1057`, el cero `0` y los enteros positivos como `1782`. El tipo `Int` es ejemplar de las clases `Enum` e `Ix`. Además, `Int` es ejemplar de `Bounded`, por lo cual las constantes `minBound` y `maxBound` representan, respectivamente, el menor y el mayor número de tipo `Int` soportado por el sistema. Los valores de estas constantes son dependientes de la implementación.

Para realizar cálculos aritméticos con números enteros, Haskell ofrece un repertorio de operadores y funciones predefinidas:

`+,*`	Operadores de adición y multiplicación
`^`	Operador de exponenciación
`-`	Operador infijo de sustracción; operador prefijo de cambio de signo
`div`	Función que calcula el cociente de la división entera
`mod`	Función que calcula el resto de la división entera
`abs`	Función que calcula el valor absoluto
`negate`	Función que calcula el opuesto
`signum`	Función que calcula el signo (-1, 0 o 1)

Los enteros no negativos (desde `0` inclusive en adelante) se llaman *números naturales*. En Haskell no existe un tipo de datos que contenga únicamente los números naturales. Sin embargo, en muchos casos se utilizan funciones que solo están definidas para los números naturales. Esto ocurre por ejemplo para las funciones definidas siguiendo los esquemas de *recursión simple* y *recursión final* que hemos estudiado en la sección 1.3.

Números fraccionarios de punto flotante

El tipo `Float` de Haskell agrupa a los números fraccionarios con representación en el formato llamado de *punto flotante*, que tienen una precisión limitada. Estos números se pueden escribir en notación decimal, con parte entera, punto decimal, parte fraccionaria y (si se desea) signo negativo:

```
0.0230879      -1025.37      409.347      -1512.0
```

Para escribir números de tipo `Float` también se puede utilizar la llamada *notación científica*, indicando un exponente entero que puede ser positivo o negativo, que se interpreta con respecto a la base `10`. Así:

```
507.43e5     equivale a   50743000.0
209.472e-7   equivale a   0.0209472
```

Asociadas al tipo `Float`, Haskell ofrece un repertorio de funciones predefinidas, que incluyen las operaciones aritméticas habituales. En particular, hay un operador infijo de división `/` y dos operadores de exponenciación: `^` para la exponenciación con exponente entero, y `**` para la exponenciación con exponente de tipo `Float`. Algunas otras funciones de interés, junto con sus tipos, se muestran en la tabla siguiente:

`negate`	`Float -> Float`	Opuesto
`abs`	`Float -> Float`	Valor absoluto
`signum`	`Float -> Int`	Signo (-1, 0 o 1)
`fromInt`	`Int -> Float`	Conversión
`ceiling`	`Float -> Int`	Redondeo por exceso
`floor`	`Float -> Int`	Redondeo por defecto
`round`	`Float -> Int`	Redondeo
`sqrt`	`Float -> Float`	Raíz cuadrada positiva
`exp`	`Float -> Float`	Potencia de e
`log`	`Float -> Float`	Logaritmo con base e
`logBase`	`Float -> Float -> Float`	Logaritmo con base dada
`pi`	`Float`	Aproximación de π
`sin, cos, tan`	`Float -> Float`	Funciones trigonométricas
`asin, acos, atan`	`Float -> Float`	Funciones trigonométricas inversas

Números fraccionarios de mayor precisión

La representación interna de los números de tipo `Float` impone un límite fijo al número de cifras sigificativas. Para cálculos numéricos que requieran mayor precisión, Haskell ofre el tipo `Double` de los números de punto flotante con *precisión doble*, cuyo número de cifras significativas es el doble que en el caso de los números de tipo `Float`. Para cálculos que requieran precisión ilimitada, Haskell ofrece el tipo `Rational`, cuyos números equivalen a fracciones con numerador y denominador de tipo `Integer`.

La clase de tipos `Num`

Todos los tipos numéricos de Haskell son ejemplares de la clase de tipos predefinida `Num`, declarada en el preludio como sigue:

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    fromInt        :: Int -> a

 -- Conjunto suficiente de metodos: todos, excepto negate o (-)

    x - y           = x + negate y
    fromInt         = fromIntegral
    negate x        = 0 - x
```

Como vemos, los métodos propios de la clase `Num` incluyen las operaciones binarias `(+)`, `(*)` y `(-)`, así como la operación `negate` que calcula el

opuesto de un número de cualquier tipo. Gracias a estos métodos, muchas funciones basadas en operaciones aritméticas sencillas se pueden programar genéricamente, con un tipo cualificado por `Num`. Por ejemplo, la función que calcula el cubo de un número admite un tipo cualificado:

```
cubo   :: Num a => a -> a
cubo x  = x*x*x
```

Nótense también los métodos `fromInteger` y `fromInt`, que permiten convertir números enteros en valores de cualquier otro tipo numérico. Gracias a los métodos de la clase `Num`, Haskell puede interpretar expresiones construidas con `(+)`, `(*)`, `(-)` y constantes enteras como valores de cualquier tipo numérico. Las siguientes pruebas con el intérprete Hugs 98 ilustran este hecho:

```
Prelude> :type -1025 + 3*4
(-1025) + 3 * 4 :: Num a => a

Prelude>  1 / (-1025 + 3*4)
-0.000987167
```

La segunda de las dos pruebas anteriores muestra que Haskell convierte implícitamente el entero `(-1025 + 3*4)` a un valor numérico fraccionario, con el cual tiene sentido la operación de división `(/)`. El método `fromInteger` hace posible la conversión.

Símbolos tales como `(+)`, `(*)`, `(-)` y otros están *sobrecargados*, ya que representan diferentes operaciones en diferentes tipos numéricos. A veces ocurre que el uso de símbolos sobrecargados en una expresión causa ambigüedades que un intérprete o compilador no es capaz de resolver. En estos casos está permitido emplear *anotaciones de tipo*, utilizando la sintaxis `e :: T`, que indica una expresión `e` cuyo tipo se declara como `T`. Haskell admite expresiones de esta forma en cualquier contexto, siempre que `T` no contradiga el tipo principal inferido para `e`.

Por ejemplo, al evaluar la expresión `(3 :: Float) + 2`, Hugs 98 obtiene el resultado `5.0`, ya que la anotación `(3 :: Float)` fuerza al sistema a operar con números fraccionarios. En este caso, la expresión `3 + 2` no es ambigüa, y se puede evaluar también dando el resultado entero `5`. En otros casos, las anotaciones de tipo son esenciales para reslver ambigüedades. Se pueden utilizar con cualquier tipo, no solamente con tipos numéricos.

Programación numérica

Las definiciones de funciones con parámetros numéricos pueden usar como patrones tanto variables como constantes. La programación de algoritmos numéricos frecuentemente exige definir funciones recursivas. Un ejemplo interesante, inspirado en el capítulo 3 de [1], es el cálculo aproximado de

ceros de funciones derivables mediante el algoritmo de Newton. El programa Hugs 98 cuyo texto aparece en el apéndice A.1 resuelve este problema utilizando una función de orden superior `newton` que espera dos parámetros: la función `f` cuyo cero se desea calcular, y una aproximación inicial `x` de dicho cero. Particularizando el parámetro funcional `f` de distintas maneras, se obtienen inmediatamente a partir de `newton` funciones que calculan raíces cuadradas, cúbicas, etc.

El programa del apéndice A.1 consta de un solo módulo, que podría editarse en un archivo llamado `Newton.hs`. En casos más complicados, un programa suele estar compuesto por varios módulos. El sistema de módulos de Haskell dispone de mecanismos de *exportación* e *importación*, como veremos en capítulos psoteriores del curso. Si no hay ninguna mención explícita de exportaciones ni importaciones, como ocurre en este caso, se entiende que el módulo no importa nada y exporta todo lo que define.

Una vez activado Hugs 98, se puede utilizar la orden `:l Nombre` para `cargar` el módulo almacenado en el archivo `Nombre.hs`. En nuestro caso, debemos ejecutar la orden `l: Newton`. A partir de ese momento, se puede continuar la interacción con el intérprete, solicitando la evaluación de cualquier expresión que utilice identificadores definidos en `Newton` (o en el preludio de Haskell). En el apéndice A.1 se pueden consultar varias pruebas de ejecución, que se han incluido como comentario en el texto del módulo.

Valores booleanos

Todos los lenguajes de programación utilizan los *valores booleanos* (también llamados *valores lógicos*) para tomar decisiones que conducen a distinciones de casos. En Haskell el tipo de los valores booleanos se llama `Bool` y contiene dos valores correspondientes a *cierto* y *falso*, representados por los identificadores `True` y `False`, que se consideran *constructoras* del tipo `Bool`. Como ya sabemos, Haskell exige que todos los identificadores correspondientes a tipos y constructoras comienzan por una letra mayúscula.

Programación con valores booleanos

Como ya hemos dicho en la sección 1.5, el tipo `Bool` es ejemplar de las clases `Eq` y `Ord`. Por lo tanto, todos los métodos de estas clases se pueden usar al definir funciones que trabajen con valores booleanos. Además, las operaciones booleanas de *negación* `not`, *conjunción* `(&&)` y *disyunción* `(||)` están predefinidas en Haskell, de tal modo que `(&&)` y `(||)` son operadores infijos, con asociatividad `r` y precedencias `3` y `2`, respectivamente. Sus definiciones, que se encuentran en el preludio del lenguaje y no deben ser incluidas en ningún programa, son las siguientes:

```
not       :: Bool -> Bool
not True   = False
not False  = True

(&&)      :: Bool -> Bool -> Bool
False && x  = False
True  && x  = x

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

En general, los patrones de tipo `Bool` utilizables como parámetros formales en definiciones de funciones, pueden ser o bien variables, o bien las constructoras `True`, `False`. Otro ejemplo de función que opera con valores booleanos es la *disyunción exclusiva*, cuyo valor es `True` si uno de los dos argumentos (pero no ambos) es `True`, y `False` en caso contrario. Esta función no está predefinida, y se puede definir como sigue:

```
xor         :: Bool -> Bool -> Bool
xor False x  = x
xor True  x  = not x
```

Ejercicios

1. Define `(&&)`, `(||)` utilizando expresiones condicionales.

2. Construye diferentes definiciones para `(&&)`, `(||)` usando patrones, de manera que las funciones resultantes sean:

   (a) Estrictas con respecto al segundo parámetro,
   pero no con respecto al primero.
   (b) Estrictas con respecto a ambos parámetros.
   (c) No estrictas con respecto a ambos parámetros.

3. Define una función `nand :: Bool -> Bool -> Bool` que calcule el resultado `True` en todos los casos, excepto si sus dos parámetros valen ambos `True`.

4. Define una función `nor :: Bool -> Bool -> Bool` que calcule el resultado `False` en todos los casos, excepto si sus dos parámetros valen ambos `False`.

5. Suponemos disponible una función `ventas :: Int -> Int` tal que `(ventas i)` da el importe de las ventas de una tienda en la semana `i`. Define dos funciones

   ```
   algunaNula ::  Int -> Bool
   ```

```
todasNulas ::  Int -> Bool
```

tales que:

`(algunaNula n)` da `True` syss *alguna* semana del intervalo $[0..n]$ tiene venta nula;

`(todasNulas n)` da `True` syss *todas* las semanas del intervalo $[0..n]$ tienen ventas nulas.

6. Suponiendo diponible la función `ventas` del ejercicio anterior, define dos funciones

```
algunExceso ::  Int -> Int -> Bool
todoExcesos ::  Int -> Int -> Bool
```

tales que:

`(algunExceso v n)` da `True` syss *alguna* semana de $[0..n]$ tiene venta con importe mayor que v;

`(todoExcesos v n)` da `True` syss *todas* las semanas de $[0..n]$ tienen ventas con importe mayor que v.

7. Modifica el planteamiento de los dos ejercicios anteriores de manera que la función `ventas` venga dada como parámetro.

2.2 Caracteres y cadenas

Caracteres

En Haskell, los caracteres son datos del tipo predefinido `Char`, que es ejemplar de las clases de tipos `Eq`, `Ord`, `Read`, `Show`, `Ix`, `Enum` y `Bounded`.

La sintaxis para representar caracteres concretos en Haskell consiste en escribirlos encerrados entre comillas simples. Un caracter escrito de este modo puede aparecer también como patrón en los lados izquierdos de ecuaciones de un programa. Por ejemplo, `'h'`, `'M'` y `'5'` son caracteres.

Los caracteres que representan dígitos, como `'5'`, *no son equivalentes* a los correspondientes números enteros (de hecho, tienen diferente tipo). Algunos caracteres tienen una representación especial, que comienza por el símbolo `\`; la mayoría de ellos son caracteres de control. Por ejemplo:

'\t'	tabulador
'\n'	nueva línea
'\\'	barra hacia atrás (\)
'\'	comilla simple (')
'\"'	comilla doble (")

Existen funciones de conversión

```
chr ::  Int -> Char      ord ::  Char -> Int
```

que pasan de enteros a caracteres y viceversa, según el código ASCII. En particular, se tiene:

```
ord '0'  =  48        ord 'A'  =  65        ord 'a'  =  97
ord '9'  =  57        ord 'Z'  =  90        ord 'z'  =  122
```

La declaración predefinida de `Char` como ejemplar de las clases `Eq` y `Ord` es tal que las comparacciones entre caracteres son equivalentes a la comparación de sus códigos numéricos. Utilizando las operaciones de conversión y comparación, es fácil definir otras funciones que operan con caracteres, como por ejemplo las que se proponen en los ejercicios de este tema.

Para imprimir caracteres en pantalla se puede utilizar la función predefinida `putChar :: Char -> IO ()`. Una llamada a esta función debe ser de la forma `(putChar e)`, siendo `e` cualquier expresión de tipo `Char`. El resultado de la llamada es un *proceso de entrada/salida* cuyo efecto es mostrar en pantalla el valor de `e`. La notación `IO ()` indica el tipo de los procesos de entrada/salida en Haskell, que estudiaremos más en detalle en el capítulo 3.

Es importante observar que la evaluación de la expresión `(putChar e)` no tiene el mismo comportamiento que la evaluación de `e`. Al evaluar `e`, el caracter obtenido se muestra en pantalla encerrado entre comillas simples, incluso en el caso de que se trate de un caracter de control. En cambio, al evaluar `(putChar e)` el caracter obtenido aparece sin comillas, y si se trata de un caracter de control se ejecuta la acción correspondiente (como por ejemplo un salto de línea, si se trataba del caracter `'\n'`). Las siguientes pruebas, ejecutadas en Hugs 98, ilustran lo que acabamos de decir. Para interpretarlas, debe tenerse en cuenta que `97` y `10` son los números de orden de los caracteres `'a'` y `'\n'`, respectivamente.

```
Prelude> putChar (chr 97)
a
Prelude> chr 97
'a'
Prelude> putChar (chr 10)

Prelude> chr 10
'\n'
```

Cadenas de caracteres

El tipo de las cadenas de caracteres en Haskell se llama `String`. Como veremos en el capítulo 3, `String` es un sinónimo del tipo de las *listas de*

caracteres, por lo cual todas las funciones de procesamiento de listas se pueden aplicar a cadenas de caracteres. El tipo `String` es ejemplar de las clases de tipos `Eq`, `Ord`, `Read` y `Show`.

Cualquier serie de caracteres encerrada entre comillas dobles representa un dato de tipo `String`, y puede aparecer también como patrón en los lados izquierdos de ecuaciones de un programa. Por ejemplo, son cadenas de caracteres:

```
"-1025"                  "Pedro"
"Pedro\nPablo\nPilar"    "Pedro\tPablo\tPilar"
```

La evaluación de una expresión `e :: String` en Haskell muestra la cadena de caracteres resultante en forma literal, encerrada entre dobles comillas y con todos sus caracteres visibles, incluso los de control. Por ejemplo, suponiendo un programa que contenga las definiciones

```
amigosN :: String
amigosN  = "Pedro\nPablo\nPilar"

amigosT :: String
amigosT  = "Pedro\tPablo\tPilar"
```

la evaluación de `amigosN` y `amigosT` en Hugs 98 se comporta del modo siguiente:

```
Prelude> amigosN
"Pedro\nPablo\nPilar"
Prelude> amigosT
"Pedro\tPablo\tPilar"
```

Para imprimir cadenas de caracteres en pantalla de manera que no aprezcan las dobles comillas y se ejecuten las acciones corrrespondientes a los caracteres de control, hay que usar `putStr :: String -> IO()`. Una llamada a esta función predefinida da como resultado un proceso de entrada/salida que imprime en pantalla la cadena de caracteres pasada como parámetro. Por ejemplo:

```
Prelude> putStr amigosN
Pedro
Pablo
Pilar
Prelude> putStr amigosT
Pedro   Pablo   Pilar
```

Otras funciones predefinidas útiles para trabajar con cadenas de caracteres son las siguientes:

- `length :: String -> Int`
  `(length xs)` calcula la longitud de la cadena `xs`.

- `(!!) :: String -> Int -> Char`
  `(xs !! i)` calcula el caracter de la posición `i` de la cadena `xs`, siempre que se tenga `0 <= i && i < length xs`. Por convenio, el primer caracter de una cadena (contando desde la izquierda) ocupa la posición 0. El operador infijo `!!` asocia por la izquierda y tiene precedencia 9.

- `(++) :: String -> String -> String`
  `xs ++ ys` calcula la concatenación de las cadenas `xs` e `ys`. El operador infijo `++` asocia por la derecha y tiene precedencia 5.

La *cadena vacía* de caracteres, que no contiene ningún caracter, se escribe `""` o también `[]`. Se comporta como elemento neutro para la operación de concatenación.

Para programar funciones que operen con cadenas de caracteres, se debe tener en cuenta que la declaración de `String` como ejemplar de las clases `Eq` y `Ord` es tal que las operaciones de comparación entre cadenas de caracteres se realizan con el siguiente criterio:

- La comparación `(xs == ys)` calcula `True` si `xs` e `ys` son idénticas, y `False` en otro caso.

- La comparación `(xs <= ys)` busca la menor posición `i` tal que se tenga `xs !! i /= ys !! i`, y da el resultado `xs !! i <= ys !! i`. En el caso de que `xs` sea un segmento inicial de `ys`, el resultado de la comparación `(xs <= ys)` es `True`.

Las siguientes pruebas en Hugs 98 sirven como ejemplo del comportamiento de la comparación entre cadenas de caracteres:

```
Prelude> "Perfecta" <= "Peste"
True
Prelude> "Arte" <= "Artemisa"
True
Prelude> "Burdeos" <= "Barcelona"
False
```

Ejercicios

1. Define funciones `isAlpha, isDigit :: Char -> Bool` que reconozcan los caracteres alfabéticos y los dígitos, respectivamente. Compara tus definiciones con las que se encuentran en el preludio standard de Haskell.

2. Las siguientes funciones están también predefinidas en Haskell. Escribe tus propias definiciones, y compáralas con las que aparecen en el preludio.

   - `isLower :: Char -> Bool`
     `(isLower c)` da resultado `True` syss `c` es una letra minúscula.
   - `isUpper :: Char -> Bool`
     `(isUpper c)` da resultado `True` syss `c` es una letra mayúscula.
   - `isAlpha :: Char -> Bool`
     `(isAlpha c)` da resultado `True` syss `c` es una letra.
   - `isAlphaNum :: Char -> Bool`
     `(isAlphaNum c)` da resultado `True` syss `c` es una letra o un dígito.

3. Define funciones `toUpper, toLower :: Char -> Char` que conviertan las letras minúsculas en mayúsculas y las mayúsculas en minúsculas, respectivamente, dejando inalterados los demás caracteres. Estas funciones también esán predefinidas. Compara tus definiciones con las que se encuentran en el preludio de Haskell.

4. Define una función `sigLet :: Char -> Char` que transforme cada letra en la letra siguiente, dejando inalterados aquellos caracteres que no sean letras. Se entiende que las funciones pedidas deben cumplir `sigLet 'z' = 'a'` y `sigLet 'Z' = 'A'`.

5. Suponiendo disponible una función `espacios :: Int -> String`, tal que `(espacios n)` devuelva una cadena de longitud `n` formada por `n` blancos (`n >= 0`), define las tres funciones especificadas a continuación:

   - `ajustaD :: Int -> String -> String`
     `(ajustaD n xs)` da como resultado la cadena `xs` rellenada con blancos por la izquierda, de modo que la longitud de la nueva cadena sea `n`.
   - `ajustaI :: Int -> String -> String`
     `(ajustaI n xs)` da como resultado la cadena `xs` rellenada con blancos por la derecha, de modo que la longitud de la nueva cadena sea `n`.
   - `ajustaC :: Int -> String -> String`
     `(ajustaC n xs)` da como resultado la cadena `xs` rellenada con blancos por ambos lados, de modo que la longitud de la nueva cadena sea `n`.

   Explica qué hacen tus funciones en el caso de que la longitud de la cadena `xs` sea mayor que `n`.

2.3 Tipos enumerados

Declaración de tipos enumerados

Además de los tipos predefinidos, Haskell proporciona medios potentes y cómodos para que el programador defina sus propios tipos de datos. Los *tipos enumerados* son los tipos de datos más sencillos que puede definir un usuario. Se definen enumerando una serie de identificadores elegidos por el programador, que deben empezar por una letra mayúscula y representan los valores posibles de los datos de ese tipo (o, como suele decirse, los *valores* del tipo). Para definir un tipo de datos enumerado se escribe una declaración con la siguiente sintaxis:

$$\texttt{data T = } \texttt{C}_0 \texttt{ | } \cdots \texttt{ | } \texttt{C}_{n-1}$$

donde `T` es el nombre del tipo (un identificador que empiece por mayúscula) y $\texttt{C}_0 \cdots \texttt{C}_{n-1}$ son los identificadores de los valores del tipo. Una vez dada esta declaración, se entiende que $\texttt{C}_i$ `::` `T` para todo $0 \leq i < n$, y que `T` no contiene ningún otro valor (excepto el valor indefinido $\perp$, que pertenece a todos los tipos). A los identificadores C_i se les llama *constructoras* de `T`. Por ejemplo, los días de la semana se pueden representar como valores de un tipo enumerado declarado como sigue:

```
data DiaSemana = Lunes | Martes | Miercoles | Jueves |
                 Viernes | Sabado | Domingo
```

La clase de tipos `Enum`

Normalmente, los tipos enumerados se declaran como ejemplares de la clase de tipos `Enum`, cuya declaración predefinida en el preludio es como sigue:

```
class Enum a where
  succ, pred           :: a -> a
  toEnum               :: Int -> a
  fromEnum             :: a -> Int
  enumFrom             :: a -> [a]             -- [n..]
  enumFromThen         :: a -> a -> [a]        -- [n,m..]
  enumFromTo           :: a -> a -> [a]        -- [n..m]
  enumFromThenTo       :: a -> a -> a -> [a]   -- [n,n'..m]

-- Conjunto suficiente de metodos: toEnum, fromEnum

  succ                  = toEnum . (1+)       . fromEnum
  pred                  = toEnum . subtract 1 . fromEnum
  enumFrom x            = map toEnum [fromEnum x ..]
```

```
enumFromThen x y      = map toEnum [fromEnum x, fromEnum y ..]
enumFromTo x y        = map toEnum [fromEnum x .. fromEnum y ]
enumFromThenTo x y z  = map toEnum
                            [fromEnum x, fromEnum y .. fromEnum z]
```

Los ejemplares de `Enum` son tipos cuyos valores se pueden poner en correspondencia biyectiva con un intervalo de valores enteros consecutivos, mediante los dos métodos `toEnum` y `fromEnum`. Los métodos `succ` y `pred`, por su parte, calculan el valor siguiente resp. el valor anterior a un valor dado de un tipo enumerado (que pueden estar indefinidos para los valores extremos). Los restantes métodos de la clase `Enum` sirven para construir intervalos representados como listas; los estudiaremos en la sección 3.3.

Cada declaración de ejemplar para la clase `Enum` debe incluir definiciones para los dos métodos `toEnum` y `fromEnum`. Para un tipo enumerado declarado del modo indicado al comienzo de esta sección, la opción más natural consiste en hacer corresponder el número i a cada constructora $\mathtt{C}_i$, como se observa en el siguiente ejemplo:

```
instance Enum DiaSemana where
   fromEnum Lunes      = 0
   fromEnum Martes     = 1
   fromEnum Miercoles  = 2
   fromEnum Jueves     = 3
   fromEnum Viernes    = 4
   fromEnum Sabado     = 5
   fromEnum Domingo    = 6

   toEnum 0 = Lunes
   toEnum 1 = Martes
   toEnum 2 = Miercoles
   toEnum 3 = Jueves
   toEnum 4 = Viernes
   toEnum 5 = Sabado
   toEnum 6 = Domingo
```

Los tipos predefinidos `Int`, `Bool` y `Char` están declarados implícitamente como ejemplares de `Enum`. En el caso de `Int`, los métodos `fromEnum` y `toEnum` se comportan como la identidad. En el caso de `Char`, `fromEnum` y `toEnum` coinciden con las funciones `ord` y `chr` explicadas en la sección 2.2, mientras que en el caso de `Bool` el método `fromEnum` transforma `False` y `True` en `0` y `1`, respectivamente.

Por lo general, los tipos enumerados también se declaran como ejemplares de las clases `Eq` y `Ord`, definiendo los métodos de comparación de tal manera que se comparen los números enteros que `fromEnum` hace corresponder a los valores del tipo. Por ejemplo, en el caso del tipo `DiaSemana` podemos declarar

```
instance Eq DiaSemana where
   x == y = fromEnum x == fromEnum y

instance Ord DiaSemana where
   x <= y = fromEnum x <= fromEnum y
```

También es posible y conveniente declarar a los tipos enumerados como ejemplares de las clases `Read`, `Show`, `Bounded` e `Ix`, como vamos a ver inmediatamente.

Derivación automática de declaraciones de ejemplar

En la práctica, resulta muy tedioso para los usuarios tener que escribir siempre explícitamente declaraciones de ejemplar para las clases `Enum`, `Eq`, `Ord`, etc., cada vez que se declara un tipo enumerado. Haskell permite que cualquier declaración de un tipo enumerado o construido `T` vaya acompañada de una cláusula opcional de la forma `deriving (`$\mathtt{CL}_1$`, ..., `$\mathtt{CL}_m$`)`. Si esta cláusula está presente, al procesar el programa el intérprete o compilador generará automáticamente declaraciones adecuadas para convertir a `T` en ejemplar de las clases $\mathtt{CL}_1$`, ..., `$\mathtt{CL}_m$.

El uso de cláusulas `deriving` para tipos de datos construidos lo estudiaremos más adelante en la sección 2. En el caso de los tipos enumerados, es recomendable utilizar declaraciones de la forma

```
data T  =  C_0 | ··· | C_{n-1}
           deriving (Enum,Bounded,Eq,Ord,Show,Read,Ix)
```

Las declaraciones de ejemplar que se generan a partir de una declaración como ésta están construidas conforme a los siguientes criterios:

- `fromEnum` hace corresponder el número i a cada constructora $\mathtt{C}_i$, y `toEnum` se comporta como inversa de `fromEnum`.
- `minBound` y `maxBound` se definen como $\mathtt{C}_0$ y $\mathtt{C}_{n-1}$, respectivamente.
- `(==)` y `(<=)` comparan de acuerdo con las imágenes de `fromEnum`.
- `show` convierte cada constructora a la cadena de caracteres que representa su identificador, mientras que `read` realiza la operación inversa.

Por ejemplo, el preludio de Haskell contiene la siguiente declaración del tipo enumerado `Ordering` (usado anteriormente en la sección 1.5):

```
data Ordering = LT | EQ | GT
      deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

Dominio de un tipo enumerado

Dado un tipo enumerado `T` con constructoras $\mathtt{C}_0$, ..., $\mathtt{C}_{n-1}$, el *dominio* $\mathtt{D}_\mathtt{T}$ se define como el conjunto $\{\mathtt{C}_0, \cdots, \mathtt{C}_{n-1}\} \cup \{\bot\}$, entendiendo que el *valor indefinido* $\bot$ es diferente de todas las constructoras $\mathtt{C}_i$. En $\mathtt{D}_\mathtt{T}$ se considera además el orden parcial $\sqsubseteq_\mathtt{T}$ definido por la condición:

$$x \sqsubseteq_T y \Longleftrightarrow_{def} x = \bot \vee x = y$$

A este orden se le llama *orden de información* asociado a `T`. Su definición expresa que:

- El valor ficticio $\bot$ tiene menos información que los restantes valores de tipo `T`.
- Los valores de `T` diferentes de $\bot$ son incomparables entre sí; ninguno de ellos tiene ni más ni menos información que los otros.

Dominios de otros tipos de datos

En general, cualquier tipo de datos `T` tiene asociado un *dominio* $\mathtt{D}_\mathtt{T}$ que se define como el conjunto de todos los valores de tipo `T`, y un *orden de información* $\sqsubseteq_\mathtt{T}$ definido de manera que $x \sqsubseteq_\mathtt{T} y$ se cumple en el caso de que el valor y tenga al menos tanta información como el valor x. Los dominios asociados a los tipos numéricos, al tipo `Bool` y al tipo `Char` se definen con el mismo criterio que los dominios de los tipos enumerados. Por ejemplo, para el tipo `Bool` se tiene:

$\mathtt{D}_\mathtt{Bool} = \{\mathtt{False}, \mathtt{True}\} \cup \{\bot\}$ con
$\bot \sqsubseteq_\mathtt{Bool} \mathtt{False}$, $\bot \sqsubseteq_\mathtt{Bool} \mathtt{True}$, $\mathtt{False} \not\sqsubseteq_\mathtt{Bool} \mathtt{True}$ y $\mathtt{True} \not\sqsubseteq_\mathtt{Bool} \mathtt{False}$

Y para el tipo `Int` se tiene:

$\mathtt{D}_\mathtt{Int} = \{\mathtt{n}, \mathtt{n}\text{valor entero}\} \cup \{\bot\}$ con
$\bot \sqsubseteq_\mathtt{Int} \mathtt{n}$ para todo entero `n`, $\mathtt{n} \not\sqsubseteq_\mathtt{Int} \mathtt{m}$ para enteros diferentes `n`, `m`.

Obsérvese que el orden de información $\sqsubseteq_\mathtt{Int}$ no coincide con el orden `<=`. En general, para tipos de datos que sean ejemplares de la clase `Ord`, el orden de información $\sqsubseteq_\mathtt{T}$ no se debe confundir con el orden `<=`.

El modo de definir los dominios asociados a otros tipos de datos lo iremos estudiando en el resto de este capítulo. A veces escribiremos simplemente $\sqsubseteq$ en lugar de $\sqsubseteq_\mathtt{T}$, cuando no sea necesario identificar explícitamente el tipo de datos `T`.

Programación con tipos enumerados

Como ejemplo sencillo de programación con tipos de datos enumerados, consideremos el pequeño programa Hugs 98 cuyo texto aparece en el apéndice A.2. Este programa consta de un solo módulo, que podría editarse en un archivo llamado `DiaSemana.hs`, y que incluye las definiciones del tipo `DiaSemana` y de dos funciones booleanas que reconocen los días laborables y festivos, respectivamente. Como se ve, Haskell permite la coincidencia entre el nombre de un módulo y el nombre de un tipo de datos definido dentro de él, ya que ésto no puede dar lugar a ambigüedades.

Una vez activado Hugs 98 y ejecutada la orden de carga `l: DiaSemana`, se puede continuar la interacción con el intérprete, solicitando la evaluación de cualquier expresión que utilice identificadores definidos en `DiaSemana` (o en el preludio de Haskell). Puesto que el tipo `DiaSemana` se ha definido incluyéndolo en varias clases de tipos, los métodos propios de estas clases también se pueden utilizar. En el apéndice A.2 se pueden consultar varias pruebas de ejecución, que se han incluido como comentario en el texto del módulo. Una de las pruebas evalúa la expresión `(toEnum 2) :: DiaSemana`, que incluye una *anotación de tipo*, necesaria en este caso para resolver el significado ambigüo del identificador sobrecargado `toEnum`. El uso de anotaciones de tipo en Haskell ya lo habíamos explicado en la sección 2.1.

Ejercicios

1. Define un tipo enumerado `Direccion` con cuatro valores, que representen los puntos cardinales. Define también una función que calcule la dirección opuesta de una dirección dada como parámetro.

2. Define un tipo enumerado `Mes` con doce valores que representen los doce meses del año. Define también una función que calcule el número de días de un mes dado como parámetro. Febrero presenta un pequeño problema. ¿Qué soluciones se te ocurren?.

3. Define un tipo enumerado `Estacion` cuyos valores representen las cuatro estaciones del año, y una función `estacionDe :: Mes -> Estacion` que calcule la estación propia de cada mes.

2.4 Tuplas

Tuplas y tipos producto

Como ya hemos estudiado en el capítulo 1, las *tuplas* en Haskell son expresiones de la forma $(e_1, \cdots, e_n)$, y juegan un papel análogo al de

registros con n campos, que se distinguen por su posición dentro de la tupla. También hemos aprendido que el tipo de una tupla con componentes `e`$_i$ `::` `T`$_i$ es el tipo producto `(T`$_1$`, ···, T`$_n$`)`, correspondiente al concepto matemático de producto cartesiano.

Ejemplo. El tipo producto `(String,String,Int)` es análogo al tipo de los registros con dos campos de tipo `String` y un campo de tipo `Int` en lenguajes del estilo de Pascal. Un dato de este tipo se puede utilizar para representar el nombre, el DNI y la edad de una persona; e.g.

```
("Luisa Gonzalez Martin","52601309-H",63) :: (String,String,Int)
```

El uso más común de n-tuplas corresponde al caso $n \geq 2$. El caso $n = 1$ no se utiliza, ya que Haskell identifica una 1-tupla `(x)` con el propio `x`. En el caso $n = 0$, existe una única 0-tupla, la *tupla vacía* `()`, cuyo tipo es el *producto vacío* `()`. Se tiene pues `() :: ()`.

Patrones con forma de tupla

Al definir funciones que tengan parámetros de tipo tupla, se pueden utilizar *patrones en forma de tupla*, que se construyen usando variables (sin repetir) y la notación de tupla, anidada si es necesario. A continuación se muestran varias funciones polimórficas definidas con ayuda de patrones en forma de tupla.

1. Funciones que seleccionan las dos componentes de una pareja. Están predefinidas en Haskell.

```
fst         ::  (a,b) -> a
fst (x, _)  =   x

snd         ::  (a,b) -> a
snd (_, y)  =   y
```

2. Funciones que seleccionan las tres componentes de una terna.

```
primero           ::  (a,b,c) -> a
primero (x, _, _) =   x

segundo           ::  (a,b,c) -> b
segundo (_, y, _) =   y

tercero           ::  (a,b,c) -> c
tercero (_, _, z) =   z
```

3. Función que reorganiza una pareja anidada.

```
reorganiza             ::  ((a,b),c) -> (a,(b,c))
reorganiza ((x,y),z)   =   (x,(y,z))
```

4. Función que aplana una pareja anidada, dando como resultado una terna.

```
aplana             ::  ((a,b),c) -> (a,b,c)
aplana ((x,y),z)   =   (x,y,z)
```

Las funciones `curry` y `uncurry`

Como hemos estudiado en la sección 1.3, Haskell permite definir funciones en forma *curryficada* o *no curryficada*. Las funciones de orden superior predefinidas `uncurry` y `curry` sirven para convertir una función dada de forma *curryficada* a forma *no curryficada*, y viceversa. Más exactamente:

- `(uncurry f)` da como resultado la versión no curryficada de la función curryficada `f`, dada como parámetro.
- `(curry g)` da como resultado la forma curryficada de la función no curryficada `g`, dada como parámetro.

Las definiciones de `curry` y `uncurry` en el preludio de Haskell son:

```
curry       :: ((a,b) -> c) -> (a -> b -> c)
curry g x y  = g (x,y)

uncurry         :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y)  = f x y
```

Las siguientes pruebas de ejecución en Hugs 98 muestran casos sencillos de uso de las funciones `uncurry` y `curry`:

```
Prelude> uncurry (+) (3,5)
8
Prelude> curry snd 3 5
5
```

Igualdad y orden entre tuplas

Cualquier producto de tipos que sean ejemplares de las clases `Eq` y `Ord` es siempre ejemplar de estas mismas clases. Esto se logra por medio de las siguientes declaraciones de ejemplar, predefinidas en Haskell:

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (u,v) = x == u && y == v

instance (Ord a, Ord b) => Ord (a,b) where
   (x,y) <= (u,v) = x < u || (x == u && y <= v)
```

Nótese que estas declaciones de ejemplar están *parametrizadas* por `a` y `b`, que representan tipos cualesquiera. Las declaraciones indican como definir los métodos de la clase `Eq` (respectivamente, `Ord`) para `(a,b)` bajo el supuesto de que `a` y `b` sean ejemplares de `Eq` (respectivamente, `Ord`). Obsérvese que la comparación de orden entre dos parejas se realiza comparando primero las primeras componentes. Si éstas son iguales, se comparan las segundas componentes. A esta manera de definir un orden entre parejas se le llama *orden lexicográfico*. La misma idea es aplicable a tuplas de tamaño mayor que dos. Por ejemplo, para el caso de ternas, Haskell tiene predefinidas las siguientes declaraciones:

```
instance (Eq a, Eq b, Eq c) => Eq (a,b,c) where
   (x,y,z) == (u,v,w) = x == u && y == v && z == w

instance (Ord a, Ord b, Ord c) => Ord (a,b,c) where
   (x,y,z) <= (u,v,w) = x < y || (x == u && y < v)
                              || (x == u && y == v & z <= w)
```

Conversión de tuplas a cadenas y viceversa

En Haskell existen también declaraciones predefinidas que hacen que un producto de tipos de la clase `Show` sea siempre ejemplar de esta clase; y análogamente para la clase `Read`. El método `show` convierte tuplas a cadenas de caracteres, convirtiendo las componentes y añadiendo paréntesis y comas. El método `read` realiza la transformación inversa. Por ejemplo:

```
Prelude> show (3+5, not True)
"(8,False)"
Prelude> read "(8,False)" :: (Int,Bool)
(8,False)
```

Dominio de un tipo producto

El dominio de un tipo producto se define de manera bastante natural como el producto de los dominios de los dos tipos componentes, pero añadiendo el valor $\bot$:

$$\mathtt{D}_{(\mathtt{a},\mathtt{b})} = \{(x, y) \in \mathtt{D}_\mathtt{a} \times \mathtt{D}_\mathtt{b}\} \cup \{\bot\}$$

con el orden de información $\sqsubseteq_{(\mathtt{a},\mathtt{b})}$ definido como el menor orden parcial sobre $\mathtt{D}_{(\mathtt{a},\mathtt{b})}$ que verifique las dos condiciones siguientes:

- $\bot \sqsubseteq_{(\mathtt{a},\mathtt{b})} (u, v)$, siempre que $u \in \mathtt{D_a}$, $v \in \mathtt{D_b}$.
- $(x, y) \sqsubseteq_{(\mathtt{a},\mathtt{b})} (u, v)$, siempre que $x \sqsubseteq_{\mathtt{a}} u$, $y \sqsubseteq_{\mathtt{b}} v$.

Es importante observar que cualquier pareja de la forma (u, v), incluso $(\bot, \bot)$, tiene más información que $\bot$. Para comprobar ésto experimentalmente, basta utilizar un programa que contenga las definiciones

```
nada :: a
nada  = error "indefinido!"

parTonto :: (a,b)
parTonto  = (nada,nada)

esPareja       :: (a,b) -> Bool
esPareja (u,v)  = True
```

según las cuales el valor de `nada` es $\bot$, mientras que el valor de `parTonto` es $(\bot, \bot)$. Los siguientes ejemplos de ejecución muestran que Hugs 98 distingue estos dos valores

```
> esPareja nada
Program error: indefinido!

> esPareja parTonto
True
```

Nótese que ninguna de las dos pruebas causa un error de tipo, ya que `nada :: ∀a a` implica que también se tiene `nada :: ∀a∀b (a,b)`. En general, `nada` resulta útil para probar el comportamiento de cualquier cómputo en el que intervenga el valor indefinido.

Funciones asociadas al producto de tipos

El producto `(a,b)` de dos tipos dados va asociado de manera natural a las dos funciones `fst` y `snd` ya definidas más atrás, que se suelen llamar *proyecciones* y sirven para calcular las dos componentes de una pareja. De su definición se deduce que las proyecciones son funciones estrictas. Se cumple que `(fst ⊥) = ⊥` y `(snd ⊥) = ⊥`.

Otra operación natural asociada al producto de tipos es la combinación de dos funciones dadas `f` y `g` para obtener una nueva función que devuelva parejas formadas por una imagen de `f` y una imagen de `g`. La función de orden superior `pair` realiza esta combinación, recibiendo como parámetros a `f` y `g`:

```
pair         :: (a -> b, a -> c) -> (a -> (b,c))
pair (f,g) x  = (f x, g x)
```

La función `pair (f,g)` se llama *emparejamiento* de `f` y `g`; solo tiene sentido cuando `f` y `g` esperan un parámetro del mismo tipo. Por ejemplo:

```
> pair ((+1),(*2)) 3
(4,6)
```

La siguiente función de orden superior es similar a `pair`, con la diferencia de que las funciones-parámetro `f` y `g` admiten parámetros de distinto tipo:

```
cross       :: (a -> b, c -> d) -> ((a,c) -> (b,d))
cross (f, g) = pair(f . fst, g . snd)
```

A `cross (f,g)` se le llama *producto* de `f` y `g`. De la definición de `cross` se deduce que `cross (f,g) (x,y) = (f x, g y)`, como muestra el ejemplo siguiente:

```
> cross (not, (*2)) (True,3)
(False,6)
```

Advertencia: las funciones `pair` y `cross` *no están predefinidas* en Haskell, por lo cual es necesario cargar un módulo que contenga sus definiciones antes de poder ejecutar las pruebas indicadas más arriba. Para este fin se puede utilizar el módulo que aparece en el apéndice A.3. Otras propiedades interesantes de `pair` y `cross` aparecen en los ejercicios de esta sección.

Programación en estilo combinatorio

Utilizando funciones de orden superior tales como `pair`, `cross` y otras, es posible programar escribiendo ecuaciones en las que no aparezca ningún parámetro explícito. Por ejemplo, en el programa del apéndice A.2 aparece la siguiente definición de una función booleana que reconoce los días laborables:

```
diaLaborable   :: DiaSemana -> Bool
diaLaborable d  = Lunes <= d && d <= Viernes
```

Esta definición utiliza una ecuación en la que aparece el parámetro `d`. Una definición alternativa que no usa parámetros explícitos se puede escribir del siguiente modo:

```
diaLaborable :: DiaSemana -> Bool
diaLaborable  = uncurry (&&) . pair ((Lunes <=),(<= Viernes))
```

El comprobar la equivalencia entre las dos definiciones anteriores se deja como ejercicio para el lector. En general, la programación funcional con ecuaciones que no utilicen parámetros explícitos se llama *programación en*

estilo combinatorio. Este estilo tiende a producir definiciones más compactas, aunque también más difíciles de entender.

Programación con tuplas

El programa Hugs98 del apéndice A.4 sirve como ejemplo interesante de uso de tuplas para resolver un problema de cálculo numérico, a saber, el cómputo del mayor entero `n` que cumpla `n <= x`, para `x :: Float` dado. Para resolver este problema se han utilizado dos algoritmos diferentes inspirados en la sección 3.6 del texto de Richard Bird [1]: un algoritmo basado en *búsqueda secuencial*, y otro más eficiente basado en *búsqueda binaria.* La programación en Haskell de estos algoritmos emplea varias funciones de orden superior, entre ellas la función `cross` explicada más arriba. Por este motivo, el módulo del apéndice A.4 incluye la orden de importación `import Product`.

Una vez activado Hugs 98 y ejecutada la orden `l: Floor` (que carga el módulo del apéndice A.4) se pueden realizar las pruebas de ejecución incluidas en el comentario final del texto del módulo, u otras similares elegidas por el usuario. Las pruebas permiten observar que el número de pasos de reducción y celdas de memoria requeridos por el cómputo decrece notablemente al emplear el algoritmo de búsqueda binaria. Para poder realizar estas observaciones hay que ejecutar previamente la orden `:s +s`, solicitando al intérprete que muestre el número de pasos de reducción y el número de celdas de memoria consumidos.

Ejercicios

1. Define `raices :: (Float,Float,Float) -> (Float,Float)` tal que `(raices a b c)` calcule la pareja formada por las dos raíces reales de la ecuación de segundo grado $ax^2 + bx + c = 0$, si existen. En el caso de que la ecuación no tenga raíces reales, las función deberá abortar, produciendo un mensaje de error.

   Sugerencia: Usa definiciones locales para evitar la cálculos repetidos.

2. ¿Es posible declarar al tipo producto `(a,b)` como ejemplar de la clase de tipos `Enum`, siempre que `a` y `b` sean ambos ejemplares de `Enum`?

3. Usando las definiciones de las funciones `pair` y `cross` y cálculo simbólico, demuestra:

   ```
   cross(f,g) (x,y) = (f x, g y)
   ```
   `cross(f,g)` $\perp \neq \perp$.

4. Usando cálculo simbólico, demuestra que `pair` y `cross` verifican las siguientes leyes:

```
fst . pair(f,g)          =  f
snd . pair(f,g)          =  g
pair(f,g) . h            =  pair(f . h, g . h)
cross(f,g) . pair(h,k)   =  pair(f . h, g . k)
cross(f,g) . cross(h,k)  =  cross(f . h, g . k)
```

Nota: Las tres primeras leyes se pueden demostrar usando extensionalidad, y las dos últimas se pueden demostrar a partir de las anteriores, razonando con ecuaciones entre funciones.

2.5 Tipos construidos

Declaración de tipos construidos

La definición de un tipo enumerado equivale a declarar una serie de constantes de ese tipo, que hemos llamado *constructoras* del tipo en la sección 2.3. Los tipos construidos siguen una idea similar, aunque más general. Se permite que las constructoras del tipo que se está definiendo tengan parámetros, especificando los tipos de éstos. La forma general de la definición de un tipo construido se escribe con la siguiente sintaxis:

$$\texttt{data T } \overline{a} \texttt{ = C}_0\ \overline{A}_0 \texttt{ | } \cdots \texttt{ | C}_{n-1}\ \overline{A}_{n-1}$$

Aquí, `T` es el nombre del tipo que se está definiendo y $\overline{a}$ abrevia una serie de variables $\texttt{a}_1, \cdots, \texttt{a}_k$ que representan tipos y se llaman *parámetros formales* de `T`. Puede ocurrir que $k = 0$, en cuyo caso `T` no tiene parámetros. Cada alternativa $\texttt{C}_i\ \overline{A}_i$ del lado derecho de la definición se debe entender como abreviatura de $\texttt{C}_i\ \texttt{A}_{i,1} \cdots \texttt{A}_{i,m_i}$, y significa que $\texttt{C}_i$ se declara como una constructora cuyo tipo es $\texttt{C}_i\ \texttt{::}\ \texttt{A}_{i,1}\ \texttt{->} \cdots \texttt{->}\ \texttt{A}_{i,m_i}\ \texttt{->}\ \texttt{T}\ \overline{a}$. Los tipos $\texttt{A}_{i,j}$ de los parámetros de $\texttt{C}_i$ pueden incluir variables, siempre que éstas se encuentren entre los parámetros formales $\overline{a}$. Obviamente, los tipos enumerados se pueden considerar como un caso particular de tipo construido.

Normalmente, interesa que los tipos construidos se incluyan en las clases de tipos `Eq`, `Ord`, `Read` y `Show`. Para lograrlo basta modificar la definición de tipo, añadiendo una cláusula `deriving`:

$$\texttt{data T } \overline{a} \texttt{ = C}_0\ \overline{A}_0 \texttt{ | } \cdots \texttt{ | C}_{n-1}\ \overline{A}_{n-1}$$
```
            deriving (Eq, Ord, Read, Show)
```

La cláusula `deriving` hace que el sistema genere automáticamente definiciones de las funciones `read` y `show` y de las operaciones de igualdad y orden para el tipo `T`. Las definiciones de `read` y `show` se basan en la transformación de los identificadores de las constructoras de `T` en cadenas de caracteres. La

definición de `(==)` se hace con un criterio de igualdad sintáctica. La definición de `(<=)` se basa en considerar que cada constructora $\texttt{C}_i$ precede a las constructoras $\texttt{C}_j$ con índice $j > i$. Dos valores de tipo `T` construidos con una misma constructora se comparan comparando las tuplas formadas por los correspondientes parámetros actuales. Según hemos estudiado en la sección 2.4, el orden empleado para esta comparación será lexicográfico. Si el tipo `T` tiene parámetros, las operaciones de comparación están limitadas por la condición de que los parámetros correspondan a tipos de las clases `Eq` resp. `Ord`.

Un tipo construido para representar figuras planas

Al definir un tipo construido, resulta útil imaginar que las diferentes constructoras que aparezcan en su declaración corresponden a diferentes *variantes* posibles para los valores de dicho tipo. Por ejemplo, supongamos que necesitamos diseñar un tipo `Figura` cuyos valores representen figuras geométricas planas. Supongamos además que solo estamos interesados en dos variantes posibles para una figura:

- Un *rectángulo*, deteminado por las coordenadas de su vértice inferior izquierdo y su vértice superior derecho.
- Un *círculo*, determinado por las coordenadas de su centro y el valor de su radio.

En estas condiciones, resulta natural declarar `Figura` como un tipo construido con dos constructoras, una para construir rectángulos y otra para construir círculos. Los tipos de los parámetros de estas constructoras se deducen fácilmente del planteamiento del problema. La declaración del tipo queda:

```
data Figura =   Rect (Float,Float) (Float,Float)
              | Circ (Float,Float) Float
                deriving (Eq, Ord, Read, Show)
```

En este caso, el tipo no tiene parámetros. La cláusula `deriving` induce la generación automática de declaraciones de ejemplar, que incluyen en particular el código indicado a continuación:

```
instance Eq Figura where
   Rect _ _   == Circ _ _    = False
   Circ _ _   == Rect _ _    = False
   Rect p1 q1 == Rect p2 q2  = (p1,q1) == (p2,q2)
   Circ c1 r1 == Circ c2 r2  = (c1,r1) == (c2,r2)
```

```
instance Ord Figura where
   Rect _ _   <= Circ _ _    = True
   Circ _ _   <= Rect _ _    = False
   Rect p1 q1 <= Rect p2 q2  = (p1,q1) <= (p2,q2)
   Circ c1 r1 <= Circ c2 r2  = (c1,r1) <= (c2,r2)
```

Gracias a la cláusula `deriving`, el usuario no necesita escribir explícitamente las declaraciones anteriores. En ellas aparece un uso típico de patrones formados con las constructoras del tipo `Figura`. En general, al definir funciones que operen con valores de un tipo construido se suelen utilizar patrones formados con las constructoras de dicho tipo (recuérdese la sintaxis de los patrones, introducida en la sección 1.3). Lo que sigue ilustra el uso de patrones para programar algunas otras funciones que operan con figuras, cuyos significados están claros a partir de sus definiciones.

```
esRect, esCirc    :: Figura -> Bool

esRect (Rect _ _)  = True
esRect (Circ _ _)  = False

esCirc (Rect _ _)  = False
esCirc (Circ _ _)  = True

centro                   :: Figura -> (Float,Float)

centro (Rect (x,y) (u,v)) = ((x+u)/2,(y+v)/2)
centro (Circ c r)         = c


base, altura, radio, area :: Figura -> Float

base (Rect (x,y) (u,v))    = u-x
base (Circ c r)            = error "Base de un circulo!"

altura (Rect (x,y) (u,v))  = v-y
altura (Circ c r)          = error "Altura de un circulo!"

radio (Rect (x,y) (u,v))   = error "Radio de un rectangulo!"
radio (Circ c r)           = r

area fig
   | esRect fig            = b*h
   | esCirc fig            = pi*r^2
                           where b  = base fig
                                 h  = altura fig
                                 r  = radio fig
                                 pi = 3.1416
```

El tipo `Maybe`

Como ejemplo de tipo construido que utiliza un parámetro, consideremos la siguiente definición, que forma parte del preludio de Haskell:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

Según la definición, un dato de tipo `Maybe a` puede ser de la forma `Nothing` o de la forma `Just x`, siendo `x` un dato de tipo `a`. Por ejemplo, vemos que `Nothing :: Maybe Int` y `Just 12 :: Maybe Int`. El uso más frecuente del tipo `Maybe a` es para expresar el resultado de un cómputo que pueda ser un valor de tipo `a` o un error. Por ejemplo, el resultado de la división entera puede ser o bien un número entero o un error (cuando el divisor es `0`). Podemos hacer explícito este comportamiento definiendo una versión especial de la función de división, que devuelva un resultado de tipo `Maybe Int`:

```
divisionSegura     :: Int -> Int -> Maybe Int
divisionSegura m n  = if n == 0
                         then Nothing
                         else Just (m 'div' n)
```

Esta función tiene la ventaja de que siempre termina sin error, incluso si el denominador es `0`. A cambio, tiene el inconveniente de que el resultado que calcula no es un número entero. En general, `Just x :: Maybe a` no es lo mismo que `x :: a`. Por esta razón, son útiles las siguientes funciones, cuyas definiciones se encuentran en el módulo `Maybe` (disponible en la biblioteca de módulos de Haskell 98):

```
isJust                :: Maybe a -> Bool
isJust Nothing         = False
isJust (Just _)        = True

isNothing             :: Maybe a -> Bool
isNothing Nothing      = True
isNothing (Just _)     = False

fromJust              :: Maybe a -> a
fromJust Nothing       = error "Maybe.fromJust: Nothing"
fromJust (Just x)      = x

fromMaybe             :: a -> Maybe a -> a
fromMaybe y Nothing    = y
fromMaybe y (Just x)   = x
```

Dada cualquier expresión `e :: Maybe a`, si `(isNothing e) = False` sabemos que `(fromJust e) :: a` está bien definido y nos da el valor de

tipo `a` "encerrado" en `e`. Para definir funciones con un parámetro de tipo `Maybe a` es útil también la siguiente función de orden superior, que efectúa una distinción de casos natural con respecto a dicho parámetro:

```
maybe             :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x
```

La definición de la función `maybe` forma parte del preludio de Haskell. Las siguientes pruebas en Hugs 98 ilustran el funcionamiento de `maybe`:

```
Prelude> maybe 0 (+1) Nothing
0
Prelude> maybe 0 (+1) (Just 3)
4
```

El tipo `Either`

También está predefinido en Haskell el siguiente tipo construido, que depende de dos parámetros:

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

Según la declaración anterior, un dato de tipo `Either a b` puede ser de la forma `Left x` siendo `x :: a`, o de la forma `Right y` siendo `y :: b`. Esto quiere decir que `Either a b` representa la *suma disjunta* de los dos tipos `a` y `b`. Por ejemplo, los dos valores `False` y `'m'` no admiten un tipo común; pero usando las constructoras `Left` y `Right` es posible convertirlos en valores de un tipo suma, donde coexisten copias de los valores booleanos y de los caracteres. Así, se tiene:

```
Left True   :: Either Bool Char
Right 'm'   :: Either Bool Char
```

De hecho, los valores anteriores admiten tipos más generales, como muestra el siguiente experimento ejecutado en Hugs 98:

```
Prelude> :t Left True
Left False :: Either Bool a
Prelude> :t Right 'm'
Right 'm' :: Either a Char
```

La suma disjunta de tipos tiene un comportamiento dual al del producto de tipos, estudiado en la sección 2.4. En particular, las funciones `join` y `plus` definidas a continuación son duales de las funciones `pair` y `cross` estudiadas en la sección 2.4:

```
join                 :: (a -> c, b -> c) -> (Either a b -> c)
join (f,g) (Left x)   = f x
join (f,g) (Right y)  = g y

plus        :: (a -> c, b -> d) -> (Either a b -> Either c d)
plus (f,g)   = join (Left . f, Right . g)
```

Las funciones `join (f,g)` y `plus (f,g)` se llaman *fusión* y *suma* de las dos funciones `f` y `g`, respectivamente. `join` y `plus` *no están predefinidas* en Haskell, aunque sí lo está la versión curryficada semejante de `join`, definida en el preludio como sigue:

```
either              :: (a -> c) -> (b -> c) -> (Either a b -> c)
either f g (Left x)  = f x
either f g (Right y) = g y
```

Cargando el módulo que aparece en el apéndice A.5 se pueden realizar las siguientes pruebas en Hugs 98, que ilustran el comportamiento de las funciones `either`, `join` y `sum`:

```
Sum> either not isDigit (Left True)
False
Sum> either not isDigit (Right 'm')
False
Sum> join (not,isDigit) (Left True)
False
Sum> join (not,isDigit) (Right 'm')
False
Sum> plus (not,ord) (Left True)
Left False
Sum> plus (not,ord) (Right 'm')
Right 109
```

En la sección 2.5 del texto [1], `join` aparece con el nombre `case`. Pero en Haskell no es posible programarla con este nombre, ya que se trata de una palabra reservada del lenguaje. De las definiciones de `either`, `join` y `plus` se deduce fácilmente que las siguientes ecuaciones son válidas:

```
either               = curry join
plus (f,g) (Left x)  = Left (f x)
plus (f,g) (Right y) = Right (g y)
```

Otras propiedades interesantes de `join` y `plus` aparecen en los ejercicios al final de esta sección.

Tipos construidos recursivos

Un tipo de datos construido se llama recursivo en el caso de que dependa de sí mismo a través de sus constructoras. Más exactamente, dados dos tipos construidos `T` y `S`, diremos que:

- `T` *depende directamente* de `S` syss `T` tiene alguna constructora que tenga un parámetro de tipo `S`.
- `T` *depende* de `S` syss `T` depende directamente de `S`, o bien `T` depende de algún otro tipo que a su vez dependa de `S`.
- `T` es un *tipo recursivo* syss `T` depende de sí mismo.

Un ejemplo sencillo de tipo recursivo es el siguiente:

```
data Nat = Cero | Suc Nat
           deriving (Eq, Ord, Read, Show)
```

Los valores de este tipo representan números naturales represenrtados al estilo de Peano:

```
Cero            :: Nat
Suc Cero        :: Nat
Suc (Suc Cero)  :: Nat

          ...
```

Típicamente, las funciones que operan con valores de un tipo recursivo se definen recursivamente, utilizando las diferentes constructoras del tipo como apoyo para la distinción de casos de la definición recursiva. Por ejemplo, las declaraciones de ejemplar para las clases `Eq` y `Ord` generadas automáticamente por la cláusula `deriving` de la declaración de `Nat` contienen definiciones recursivas para los métodos `(==)` y `(<=)`:

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

instance Ord Nat where
   Cero  <= Cero   = True
   Cero  <= Suc y  = True
   Suc x <= Cero   = False
   Suc x <= Suc y  = x <= y
```

Para demostrar propiedades de funciones que operen con valores de un tipo recursivo tiene importancia las técnicas de *razonamiento por inducción*, que estudiaremos en los capítulos 3 y 4. Algunos ejemplos sencillos de inducción para el tipo recursivo `Nat` aparecen en los ejercicios de esta sección.

Un tipo recursivo construido para representar muñecas rusas

En la práctica, muchos problemas de programación exigen trabajar con datos cuya representación más natural conduce a la definición de un tipo de datos recursivo. Pensemos por ejemplo en las conocidas *muñecas rusas*. Cada una de estas figuras tiene una cierta forma y puede ser maciza o estar rellena con otra muñeca. Esta descripción intuitiva de las muñecas rusas se puede trasladar inmediatamente a la declaración de un tipo recursivo `Rusa` con dos constructoras, junto con un tipo auxiliar enumerado `Forma` para representar las formas posibles de las muñecas. Para nuestro ejemplo, elegimos como formas imágenes de políticos.

```
data Rusa = Maciza Forma | Rellena Forma Rusa
            deriving (Eq, Ord, Read, Show)

data Forma = Lenin | Stalin | Gorbachov | Yeltsin | Putin
             deriving (Enum, Bounded, Ix, Eq, Ord, Read, Show)
```

Obsérvese que en un lenguaje de programación imperativo la definición de un tipo de datos semejante a este sería más complicada. Posiblemente, `Rusa` se tendría que declarar como un tipo de registros con variantes, incluyendo un campo con un puntero a registros del mismo tipo. La sencillez y potencia de los tipos de datos construidos es una de las ventajas de los lenguajes funcionales del estilo ML o Haskell.

Una vez definido el tipo `Rusa`, se puede utilizar recursión para definir diferentes funciones que operen con muñecas rusas. Por ejemplo, la siguiente función booleana reconoce las muñecas rusas que solo incluyen imágenes de políticos "vetustos":

```
arcaica               :: Rusa -> Bool
arcaica (Maciza f)     = vetusta f
arcaica (Rellena f r)  = vetusta f && arcaica r

vetusta   :: Forma -> Bool
vetusta f  = f == Lenin || f == Stalin
```

Aunque ninguno de los dos tipos `Rusa` y `Nat` tiene parámetros, otros tipos recursivos sí pueden tenerlos, como veremos en los capítulos 3 y 4.

Revisión de la sintaxis de los tipos

La sintaxis de los *tipos simples*, introducida en las secciones 1.3 y 1.5, necesita ser ampliada para tener en cuenta el posible uso de tipos construidos declarados por un usario. La nueva sintaxis queda:

```
T ::=   a
      | TP
      | (T1, ..., Tn)
      | (TC T1 ... Tn)
      | (T1 -> T)
```

En el caso `(TC T1 ... Tn)` se entiende que `TC` es el nombre de un tipo construido con n parámetros, que se han sustituido por los tipos simples `Ti`. Los nombres de los tipos construidos se llaman también *constructoras de tipos* porque se pueden usar para construir tipos de esta manera. En el caso particular $n = 0$, el tipo `(TC T1 ... Tn)` se reduce a `TC`.

Dominio de un tipo construido

El dominio $\mathtt{D_{T\ \overline{a}}}$ de un tipo construido declarado como

$$\mathtt{data\ T}\ \overline{a}\ \ \mathtt{=}\ \ \mathtt{C}_0\ \overline{A}_0\ \mathtt{|}\ \cdots\ \mathtt{|}\ \mathtt{C}_{n-1}\ \overline{A}_{n-1}$$

se define de manera que incluya como elementos a $\bot$ y a todos los valores de la forma $(\mathtt{C}_i\ \overline{u_i})$, siendo $0 \leq i < n$ y siendo $\overline{u_i}$ valores de los dominios correspondientes a los tipos de los parámetros de la constructora $\mathtt{C}_i$. El orden de información $\sqsubseteq_{\mathtt{T\ \overline{a}}}$ se define como el menor orden parcial sobre $\mathtt{D_{T\ \overline{a}}}$ que verifique las condiciones siguientes, para todo $0 \leq i < n$:

- $\bot \sqsubseteq_{\mathtt{T\ \overline{a}}} (\mathtt{C}_i\ \overline{u_i})$, para cualquier valor $(\mathtt{C}_i\ \overline{u_i})$ construido con la constructora $\mathtt{C}_i$.
- $(\mathtt{C}_i\ \overline{u_i}) \sqsubseteq_{\mathtt{T\ \overline{a}}} (\mathtt{C}_i\ \overline{v_i})$, siempre que cada componente de $\overline{u_i}$ sea menor o igual que la componente homóloga de $\overline{v_i}$ con respecto al orden de información del dominio correspondiente.

Por ejemplo, en el caso del tipo `Either a b`, resulta:

$$\mathtt{D_{Either\ a\ b}} = \{\mathtt{Left}\ x \mid x \in \mathtt{D_a}\} \cup \{\mathtt{Right}\ y \mid y \in \mathtt{D_b}\} \cup \{\bot\}$$

con el orden de información $\sqsubseteq_{\mathtt{Either\ a\ b}}$ definido como el menor orden parcial sobre $\mathtt{D_{Either\ a\ b}}$ que verifique las condiciones siguientes:

- $\bot \sqsubseteq_{\mathtt{Either\ a\ b}} \mathtt{Left}\ u$, siempre que $u \in \mathtt{D_a}$.
- $\bot \sqsubseteq_{\mathtt{Either\ a\ b}} \mathtt{Right}\ v$, siempre que $v \in \mathtt{D_b}$.
- $\mathtt{Left}\ u \sqsubseteq_{\mathtt{Either\ a\ b}} \mathtt{Left}\ u'$, siempre que $u \sqsubseteq_{\mathtt{a}} u'$.
- $\mathtt{Right}\ v \sqsubseteq_{\mathtt{Either\ a\ b}} \mathtt{Right}\ v'$, siempre que $v \sqsubseteq_{\mathtt{b}} v'$.

En los dominios de tipos construidos recursivos pueden aparecer *elementos infinitos*. Consideraremos esta cuestión al estudiar el dominio del tipo recursivo de las listas, en el capítulo 3.

Programación con tipos construidos

Un ejemplo de programación con tipos construidos aparece en el programa del apéndice A.6, formado por un único módulo `Carta`, que define y exporta el tipo construido `Carta`, junto con dos tipos enumerados auxiliares y una serie de funciones.

Los tipos enumerados `Palo` y `Valor` representan los cuatro palos de la baraja española y los diez valores posibles de una carta, respectivamente. El tipo construido `Carta` representa las cartas de la baraja española con ayuda de cuatro constructoras que corresponden a los cuatro palos. Estas constructoras no se deben confundir con las constructoras constantes del tipo `Palo`, cuyos identificadores son diferentes. En general, en Haskell no se permite que un mismo identificador aparezca como constructora en dos tipos de datos diferentes.

Las funciones `palo` y `valor` calculan el palo y el valor, respectivamente, de una carta dada. Las funciones booleanas `esAs` y `esFigura` reconocen los ases y las figuras de cualquier palo, respectivamente. Finalmente, la función booleana `superior` compara dos cartas de un mismo palo según su valor. El resultado de la comparación es `False` siempre que las dos cartas sean de distinto palo.

Los lados izquierdos de algunas de estas definiciones utilizan *patrones* formados con variables y constructoras. Por ejemplo, `(Copa v)` es un patrón que representa una carta del palo de las copas, cuyo valor está representado por la variable `v`. Puesto que el tipo `Carta` tiene cuatro constructoras, un dato de este tipo puede tener cuatro formas posibles, correspondientes a los cuatro patrones `(Oro v)`, `(Copa v)`, `(Basto v)` y `(Espada v)`.

Una vez activado Hugs 98 y cargado el módulo `Carta`, la interacción con el intérprete permite, entre otros, los cómputos que se indican como comentarios al final del texto del módulo. En particular, los dos últimos cómputos muestran que la función `superior` no tiene el mismo comportamiento que el orden `(>)`. La razón es que la definición de `(>)` ha sido generada automáticamente por el sistema para incluir al tipo `Carta` en la clase `Ord`, con el criterio que hemos explicado más arriba. Como la constructora `Basto` aparece después que `Copa` en la definición del tipo, el sistema considera que `(Basto v) > (Copa w)` vale `True`, cualesquiera que sean los valores `v` y `w`.

Ejercicios

1. Los parroquianos de la taberna galáctica *Gloops* son terrícolas, venusianos y marcianos. Los terrícolas tienen nombre, sexo y edad. Los

marcianos tienen número de serie, color (con cinco valores posibles) y edad. Finalmente, los venusianos tienen nombre, sexo venusiano (con tres valores posibles) y número de cuernos (les crece un cuerno cada tres años venusianos). Define un tipo de datos construido `Ente` cuyos valores sirvan para representar a los parroquianos de *Gloops*. Te conviene definir algunos tipos enumerados auxiliares.

2. A partir del tipo `Ente` del ejercicio anterior, define funciones que sirvan para distinguir a las tres clases de parroquianos, y para calcular en cada caso sus atributos característicos. Una vez definidas tus funciones, explica como utilizarlas para reconocer si un ente es o no venusiano, y para calcular en caso afirmativo su número de cuernos.

3. Para representar temperaturas se conocen los sistemas *Centígrado*, *Fahrenheit* y *Celsius*, entre otros. Define un tipo de datos `Temp` construido con tres constructoras, cuyos valores sirvan para representar temperaturas en estos tres sistemas.

4. El intervalo de temperaturas [0..100] de la escala centígrada se corresponde con el intervalo [32..212] de la escala Fahrenheit y con el intervalo [273..373] de la escala Celsius. Sabiendo esto, ¿te parece que las operaciones de igualdad y orden que generaría Haskell con una directiva `deriving (Eq, Ord)` son adecuadas? En caso negativo, ¿cómo se podrían programar definiciones más adecuadas?.

5. Define una función `especial :: Either DiaSemana Mes -> Bool`, que calcule el resultado `True` si su argumento representa un día festivo o un mes primaveral, y el resultado `False` en otro caso. Te conviene utilizar las funciones `diaFestivo` y `estacionDe`, estudiadas en la sección 2.3.

6. Escribe declaraciones de ejemplar parametrizadas para incluir el tipo `Either a b` en las clases de tipos `Eq` y `Ord`. Estas declaraciones están predefinidas en Haskell.

7. Demuestra la validez de las ecuaciones siguientes, que expresan propiedades de la función `plus`:

```
plus(f,g) (Left x)  = Left (f x)
plus(f,g) (Right y) = Right (g y).
```

8. Usando cálculo simbólico, demuestra que `join` y `plus` verifican las siguientes leyes, duales de las leyes de `pair` y `cross`:

```
join(f,g) . Left       =  f
join(f,g) . Right      =  g
h . join(f,g)          =  join(h . f, h . g)
join(f,g) . plus(h,k)  =  join(f . h, g . k)
join(f,g) . plus(h,k)  =  join(f . h, g . k)
```

Nota: Las tres primeras leyes se pueden demostrar usando extensionalidad, y las dos últimas se pueden demostrar a partir de las anteriores, razonando con ecuaciones entre funciones.

9. Define una función recursiva `natToInt :: Nat -> Integer` que convierta números naturales de Peano en valores del tipo predefinido `Integer`. ¿Es posible definir una función inversa de la anterior, con tipo `intToNat :: Integer -> Nat`?

10. Define una función `(+) :: Nat -> Nat -> Nat` que calcule la suma de números naturales en notación de Peano, usando recursión sobre el segundo argumento. Demuestra por inducción que las tres propiedades siguientes se cumplen para todo `x, y :: Nat`.

    (a) `Cero + y = y`.
    (b) `Suc x + y = Suc (x + y)`.
    (c) `x + y = y + x`.

11. Define un tipo de datos recursivo `Ent` cuyos valores se puedan utilizar como representaciones simbólicas de los números enteros. Escribe declaraciones que conviertan a este tipo en ejemplar de las clases `Eq` y `Ord`. ¿Sería adecuado derivar estas declaraciones automáticamente?

12. Recuerda el tipo de datos `Rusa` definido en esta sección para representar muñecas rusas con forma de gobernantes. Define una función recursiva `regresion :: Rusa -> Rusa` tal que `(regresion m)` sea la muñeca obtenida a partir de `m` eliminando todas las apariciones de `Putin` y reemplazando `Gorbachov` y `Yeltsin` por `Lenin` y `Stalin`, respectivamente.

13. Escribe las declaraciones de ejemplar para las clases `Eq` y `Ord` que se generan automáticamente a partir de la cláusula `deriving` incluida en la declaración del tipo recursivo `Rusa`.

14. Define un tipo recursivo `Prop` cuyos valores representen fórmulas de la lógica de proposiciones. Programa una función `fnn :: Prop -> Prop` tal que `(fnn p)` calcule una fórmula lógicamente equivalente a `p`, en la cual el signo de negación solamente se aplique a fórmulas atómicas.

2.6 Tipos sinónimos

Declaración de tipos sinónimos

Como hemos visto en la sección anterior, las reglas sintácticas de formación de tipos en Haskell permiten construcciones complejas. Para conveniencia del programador, está permitido introducir un identificador como

nombre abreviado de un tipo. Esto se hace mediante las definiciones de tipo sinónimo, cuya sintaxis es la siguiente:

```
type T a = TD
```

Aquí, `T` es un identificador elegido por el programador como nombre abreviado del tipo simple `TD`, que debe estar construido según las reglas sintácticas explicadas en la sección 2.5. Las variables $\overline{a}$, en caso de estar presentes, representan *parámetros* que pueden aparecer también en `TD`. Por ejemplo, la definición

```
type Pareja a = (a,a)
```

significa que se define `Pareja a` como alias del tipo producto `(a,a)`. Los sinónimos que dependen de parámetros se mantienen al reemplazar éstos por tipos concretos. Por ejemplo, `Pareja Int` es sinónimo de `(Int,Int)`.

Al compilar un programa Haskell, los tipos sinónimos se eliminan, reemplazándolos por los lados derechos de sus definiciones. Puede suceder que el lado derecho de la definición de un tipo sinónimo contenga otros sinónimos. Por ello, el proceso de eliminación se reitera hasta que todos los identificadores de tipos sinónimos desaparecen. Para garantizar que este proceso termine, se prohibe que las definiciones de tipos sinónimos sean recursivas.

Programación con tipos sinónimos

La definición de tipos sinónimos bien elegidos contribuye a la legibilidad de los programas. En el apéndice A.7 aparece un programa formado por un único módulo `Persona`. Este módulo define y exporta un tipo sinónimo para representar personas, así como una serie de funciones para operar con personas. Haskell no impone ninguna limitación al orden de presentación de las diferentes declaraciones. Aplicando la regla de eliminación de sinónimos, observamos que `Persona` es sinónimo de `(String, String, Int)`. Utilizar el identificador `Persona` es más ilustrativo, pero Haskell trabaja internamente con el tipo `(String, String, Int)` a todos los efectos. En particular, una terna `(xs, ys, n) :: Persona` con `n` *negativo* no causará un error de tipo, a pesar de que el programador seguramente espera que la edad de cualquier persona sea no negativa.

Una vez cargado este módulo, se podrían efectuar los cómputos indicados como comentarios al final del texto del mismo. Nótese que las funciones de comparación `(>)` y `mayor` no tienen el mismo comportamiento, por razones similares a las que hemos explicado ya al comentar el programa del apéndice A.6 en la sección 2.5. Obsérvese también la función `printPersona`, cuya definición utiliza el caracter de control `'\t'` (tabulador) y la función primitiva de entrada-salida `putStr` (explicada en la sección 2.2).

Declaraciones `newtype`

Dado que los tipos sinónimos se tratan como abreviaturas de otros tipos, no está permitido declararlos como ejemplares de clases de tipos. En algunas ocasiones, esto puede resultar inconveniente. Por ejemplo, si declaramos un tipo `Angulo` como sinónimo de `Float`, las comparaciones de igualdad y orden entre ángulos serán las comparaciones aritméticas entre números de tipo `Float`, que no reflejan el hecho de que dos ángulos que se diferencien en un múltiplo de 2π se deben identificar.

Para remediar este problema se puede utilizar una declaración de la forma

```
newtype Angulo = Ang Float
```

que define a `Angulo` como un *tipo nuevo* con valores de la forma `Ang x`, `x :: Float`. En el texto de un programa que contenga la declaración anterior, `Ang` se debe utilizar como una constructora, y `Angulo` no se trata como sinónimo de `Float`. Es posible así construir declaraciones de ejemplar que definan métodos adecuados para la comparación de ángulos: [1]

```
normaliza    :: Float -> Float
normaliza x
  | x < 0      = normaliza (x+rot)
  | x >= rot   = normaliza (x-rot)
  | otherwise  = x
               where rot = 2*pi
                     pi  = 3.1416

instance Eq Angulo where
  Ang x == Ang y = normaliza x == normaliza y

instance Ord Angulo where
  Ang x <= Ang y = normaliza x <= normaliza y
```

En general, una declaración de estilo `newtype` sigue la sintaxis

```
newtype T ā  =  C TD
```

donde `C` debe ser una constructora, y `TD` un tipo simple que no debe depender recursivamente del nuevo tipo `T` que se está declarando, y que puede utilizar los parámetros $\overline{a}$.

A primera vista, una declaración `newtype` parece un caso particular de declaración de tipo construido con una sola constructora. Sin embargo, hay una diferencia significativa: la constructora `C` introducida por una

[1] Véase el módulo Hugs 98 del apéndice A.8.

declaración `newtype` se implementa como una operación *estricta*, de manera que `C` $\perp = \perp$. Por ejemplo, para el tipo nuevo `Angulo` se tiene que `Ang` $\perp = \perp$. Esto no es así para las constructoras de los tipos declarados como `data`, las cuales se implementan como operaciones *no estrictas*. Por ejemplo, para el tipo construido `Either a b` se cumple que `Left` $\perp \neq \perp$ y `Right` $\perp \neq \perp$.

Como consecuencia de esto, un tipo nuevo que se haya declarado como `newtype` se puede implementar algo más eficientemente que si se tratase de un tipo construido, evitando la representación de la constructora `C` en tiempo de ejecución.

Ejercicios

1. Los siguientes tipos sinónimos se pueden utilizar para representar puntos y rectas del plano:

   ```
   type Punto = (Float, Float)
   type Recta = (Punto, Punto)
   ```

   Define tres funciones de tipo `Recta -> Bool` que reconozcan si una recta dada como pareja de puntos es *degenerada*, *horizontal* o *vertical*. Entendiendo que la pareja de puntos `(p,q)` representa una recta degenerada cuando los dos puntos son el mismo.

2. Define una función `pertenece :: Punto -> Recta -> Bool` que calcule si un punto dado se encuentra o no sobre una recta dada. Ten cuidado con los casos especiales (recta horizontal, vertical o degenerada).

3. Utilizando los tipos `Anno` y `Dia` como sinónimos de `Int` y el tipo enumerado `Mes` estudiado en la sección 2.3, define un tipo sinónimo `Fecha` cuyos valores representen fechas.

4. Define tres funciones que calculen el día, el mes y el año de una fecha dada como argumento.

5. Razona si las operaciones de igualdad y orden utilizadas automáticamente por Haskell para el tipo `Fecha` comparan o no fechas de la manera correcta. En caso negativo, define otras funciones de comparación de fechas, que operen correctamente.

6. Define una función `showFecha :: Fecha -> String`, que devuelva una cadena de caracteres adecuada para ser mostrada como representación de la fecha dada como parámetro.

7. En el calendario Gregoriano vigente en la actualidad, un año representado como un número entero `a` es bisiesto si y solo si `a` es múltiplo de `4`, y además, si `a` es múltiplo de 100, entonces `a` también es múltiplo de `400`. Define una función `bisisesto :: Anno -> Bool` que calcule si un año dado es o no bisiesto.

8. Define ahora una función `nrDias :: Mes -> Anno -> Int` tal que `(nrDias m a)` calcule el número de días del mes `m` en el año `a`. Usa el ejercicio anterior.

9. En este ejercicio y el siguiente, suponemos un tipo enumerado `DiaSemana` con siete valores correspondientes a los días de la semana, enumerados en el orden `Domingo`, `Lunes`, $\cdots$, `Sabado`. Teniendo en cuenta que `365 ‘mod‘ 7 = 1` y que el `1` de Enero del año `0` del calendario Gregoriano fué Domingo, el día de la semana en el que cae el 1 de Enero del año número `a` se puede calcular evaluando la expresión

   ```
   (a + (a-1) ‘div‘ 4 - (a-1) ‘div‘ 100 +
   (a-1) ‘div‘ 400) ‘mod‘ 7
   ```

   Usando esta fórmula, define `diaAnnoNuevo :: Anno -> DiaSemana` que calcule el día de la semana en que cae el día de año nuevo de un año dado.

10. Define una función `primerDiaMes :: Mes -> Anno -> DiaSemana`, tal que `(primerDiaMes m a)` calcule el día de la semana en que cae el día `1` del mes `m` del año `a`. Aprovecha los ejercicios anteriores.

Capítulo 3

Programación funcional con listas

3.1 El tipo de datos de las listas

Las listas como tipo de datos recursivo

Muchos problemas de programación requieren trabajar con datos complejos formados como un agrupamiento de datos más sencillos. A estos agrupamientos se les llama también *estructuras de datos*. Los agrupamientos construidos en forma de sucesión se llaman *estructuras de datos lineales*. En lenguajes funcionales del estilo de Haskell, las estructuras de datos lineales más populares son las listas. Para representar listas se dispone de un tipo de datos recursivo, cuya declaración predefinida es

```
infixr 5 :
data [a] = [] | a : [a]
           deriving (Eq, Ord, Read, Show)
```

Según esta declaración, `[a]` es el tipo de las listas con elementos de tipo `a`, dado como parámetro, y se dispone de dos constructoras de listas:

- `[] :: [a]`, que representa la *lista vacía*.
- `(:) :: a -> [a] -> [a]`, que construye una lista `(x:xs) :: [a]` añadiendo un nuevo elemento `x :: a` a otra lista `xs :: [a]` previamente construida. Se dice que `x` es la *cabeza* y que `xs` es el *resto* de la nueva lista `(x:xs)`.

La constructora `(:)` está declarada como operador infijo. En general, Haskell permite utilizar operadores infijos cuyo nombre comience por el símbolo ':' como constructoras de tipos de datos declarados por el usuario.

Construcción de listas

Una lista de longitud n siempre se puede construir comenzando con la lista vacía `[]` y aplicando n veces la constructora de listas `(:)`. Por ejemplo, `0:(1:(2:[]))` representa una lista de longitud 3 cuyos elementos son los tres primeros números naturales. Puesto que `(:)` asocia por la derecha, la expresión anterior se puede abreviar como `0:1:2:[]`. Por convenio, una lista de longitud n de la forma $\texttt{x}_0\texttt{:}\ \cdots\ \texttt{:x}_{n-1}\texttt{:[]}$ se puede escribir también utilizando la notación $\texttt{[x}_0\texttt{,}\ \cdots\texttt{,}\ \texttt{x}_{n-1}\texttt{]}$. Por ejemplo, `[0,1,2]` es equivalente a `0:1:2:[]`.

En una lista $\texttt{x}_0\texttt{:}\ \cdots\ \texttt{:x}_{n-1}\texttt{:[]}$ es esencial tener en cuenta que todos los valores $\texttt{x}_i$, llamados *elementos* de la lista, deben tener un tipo común `a`, de manera que la lista pueda tener tipo `[a]`. Además, el número de elementos de la lista, el orden en el que aparezcan y sus posibles repeticiones, son relevantes. Por ejemplo, las expresiones que siguen representan listas correctamente formadas de tipo `[Int]`, y todas ellas son diferentes entre sí, ya que difieren o bien en el número de elementos, o bien en el orden de los elementos que las forman, o bien en el número de repeticiones.

```
[1,2]  [2,1]  [1,2,3]  [3,2,1]  [1,1,2]  [1,2,2]  [2,2,1]
```

En cambio, una "lista" tal como `[False, 2, True]` se rechaza como *mal tipada*, ya que no existe un tipo común a todos sus elementos. Para representar una información equivalente a la de la lista anterior, se podría utilizar `[Left False, Right 2, Left True] :: [Either Bool Int]`, que sí está bien tipada.

La lista vacía `[]` no tiene elementos y tiene tipo `[a]` para cualquier elección `a` del tipo de los elementos. Para cualquier valor `x :: a`, la lista `[x] :: [a]` (que abrevia `x:[]`) se llama *lista unitaria* formada por el único elemento `x`. Por ejemplo, `[5] :: [Int]` es una lista unitaria de enteros, y difiere de otras listas no unitarias tales como `[5,5]` y `[5,5,5]`.

Las cadenas son listas de caracteres

El tipo `String` que hemos estudiado en la Sección 2.2 se trata en Haskell como sinónimo del tipo `[Char]` de las cadenas de caracteres. Así, la cadena vacía de caracteres `""` equivale a la lista vacía `[]`, y las cadenas no vacías de caracteres se consideran como construidas con la constructora de listas. Por ejemplo, `"Pedro"` es equivalente a `['P', 'e', 'd', 'r', 'o']` y a `'P':'e':'d':'r':'o':[]`. Todas las funciones y técnicas de programación que vamos a estudiar en este capítulo se pueden aplicar automáticamente a cadenas de caracteres.

Listas anidadas

Nada impide que los elementos de una lista sean a su vez listas, con tal de que sean todas del mismo tipo. El tipo de las listas anidadas cuyos elementos son listas de tipo `[a]` se escribe `[[a]]`. Por ejemplo, las siguientes expresiones representan listas anidadas de tipo `[[Int]]`:

- `[[1,2,3],[],[4],[3,2,1]]`, una lista de listas de enteros.
- `[[3,2,1]]` una lista unitaria de listas de enteros, cuyo único elemento es la lista `[3,2,1]`.
- `[[3]]` una lista unitaria de listas de enteros, cuyo único elemento es la lista unitaria `[3]`.
- `[[]]` una lista unitaria de listas de enteros, cuyo único elemento es la lista vacía.

En particular, el ejemplo anterior muestra que `[[]]` *no es la lista vacía*, como suelen pensar equivocadamente muchos principiantes. La lista unitaria `[[]]` tiene tipo `[[a]]` para cualquier elección del tipo parámetro `a`.

Dominio del tipo de las listas

La definición del dominio asociado a un tipo construido cualquiera se ha explicado en la sección 2.5. En el caso de las listas, el dominio $\mathtt{D}_{[\mathtt{a}]}$ incluye tres clases de listas:

(a) *Listas finitas y terminadas*, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_{n-1} : []$ donde $n \geq 0$ y $\mathtt{x}_i \in \mathtt{D}_\mathtt{a}$, para todo $0 \leq i < n$. Este caso incluye la *lista vacía* `[]` cuando $n = 0$, así como las *listas unitarias* cuando $n = 1$.

(b) *Listas finitas y no terminadas*, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_{n-1} : \bot$ donde $n \geq 0$ y $\mathtt{x}_i \in \mathtt{D}_\mathtt{a}$, para todo $0 \leq i < n$. Este caso incluye la *lista indefinida* $\bot$ cuando $n = 0$.

(c) *Listas infinitas*, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_i : \cdots$ donde $\mathtt{x}_i \in \mathtt{D}_\mathtt{a}$ está en la posición i de la lista para cualquier $i \geq 0$.

Una lista se llama *finita* si cae dentro de alguna de las dos categorías (a) o (b); y se llama *inconclusa* si cae dentro de alguna de las dos categorías (b) o (c). En lo sucesivo, para indicar a qué categoría pertenece una lista `xs :: [a]` usaremos las notaciones siguientes:

- `finTer xs` para indicar que `xs` es finita y terminada.
- `fin xs` para indicar que `xs` es finita (terminada o no).

- `inf xs` para indicar que `xs` es infinita.

En el apéndice A.9 se encuentran las definiciones de tres funciones con las cuales se puede experimentar el cálculo de listas de las tres clases posibles. Las listas finitas y terminadas son las únicas que se pueden calcular en tiempo finito y sin causar errores de ejecución. Las listas finitas y no terminadas normalmente no deben aparecer en programas correctamente diseñados. Por el contrario, la programación de funciones que generen listas infinitas sí puede ser útil en problemas prácticos, ya que una lista infinita puede ser procesada por funciones que extraigan una parte finita de ella sin evaluar la lista por completo. Más adelante encontraremos ejemplos de esta técnica de programación.

Para definir el orden de información sobre el dominio de las listas, debemos especificar cuando se cumple $\mathtt{xs} \sqsubseteq_{\mathtt{[a]}} \mathtt{xs'}$ para listas cualesquiera $\mathtt{xs}$, $\mathtt{xs'} \in \mathtt{D}_{\mathtt{[a]}}$. Lo hacemos distinguiendo los tres casos posibles para `xs`:

(a) `xs` es finita y terminada, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_{n-1} : \mathtt{[]}$.
Entonces $\mathtt{xs} \sqsubseteq_{\mathtt{[a]}} \mathtt{xs'}$ se cumple si y solamente si `xs'` es de la forma $\mathtt{x'}_0 : \cdots : \mathtt{x'}_{n-1} : \mathtt{[]}$, siendo $\mathtt{x}_i \sqsubseteq_{\mathtt{a}} \mathtt{x'}_i$ para todo $0 \leq i < n$.

(b) `xs` es finita y no terminada, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_{n-1} : \bot$.
Entonces $\mathtt{xs} \sqsubseteq_{\mathtt{[a]}} \mathtt{xs'}$ se cumple si y solamente si `xs'` es de la forma $\mathtt{x'}_0 : \cdots : \mathtt{x'}_{n-1} : \mathtt{ys}$, siendo $\mathtt{x}_i \sqsubseteq_{\mathtt{a}} \mathtt{x'}_i$ para todo $0 \leq i < n$, e $\mathtt{ys} \in \mathtt{D}_{\mathtt{[a]}}$ una lista cualquiera.

(c) `xs` es infinita, de la forma $\mathtt{x}_0 : \cdots : \mathtt{x}_i : \cdots$
Entonces $\mathtt{xs} \sqsubseteq_{\mathtt{[a]}} \mathtt{xs'}$ se cumple si y solamente si `xs'` es de la forma $\mathtt{x'}_0 : \cdots : \mathtt{x'}_i : \cdots$, siendo $\mathtt{x}_i \sqsubseteq_{\mathtt{a}} \mathtt{x'}_i$ para todo $i \geq 0$.

Funciones de procesamiento de listas

En los lenguajes de programación imperativos, las listas se suelen representar como estructuras formadas por registros encadenados por punteros, de modo que cada elemento de la lista apunta al siguiente, excepto el último, que no apunta a ninguna parte. Las implementaciones de los lenguajes funcionales también utilizan a bajo nivel una representación de este estilo, pero el programador puede ignorarla. Las funciones que procesan listas se definen muy cómodamente, utilizando patrones y recursión. En los casos más sencillos, la recursión es innecesaria. Por ejemplo, para reconocer la lista vacía y extraer la cabeza y el resto de una lista no vacía se emplean tres funciones básicas, definidas en el preludio de Haskell del siguiente modo:

```
null             :: [a] -> Bool
null []           = True
null (_:_)        = False
```

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

tail             :: [a] -> [a]
tail (_:xs)       = xs
```

Obérvese que la cabeza y el resto de la lista vacía están indefinidos. Las definiciones anteriores emplean los dos patrones de tipo `[a]` utilizados con más frecuencia, a saber:

- `[]`, patrón al que solo se ajusta la *lista vacía*.
- `x:xs` (siendo `x :: a` y `xs :: [a]` variables), patrón al que se ajusta cualquier *lista no vacía* con *cabeza* `x` y *resto* `xs`.

La definición de funciones más complicadas requiere recursión. Esto ocurre por ejemplo en el caso de las funciones predefinidas `last` e `init`:

```
last           :: [a] -> a
last (x:xs)
  | null xs     =  x
  | otherwise   = last xs

init           :: [a] -> a
init (x:xs)
  | null xs     =  []
  | otherwise   = x : init xs
```

Dada una lista no vacía `xs`, `last xs` calcula su último elemento, mientras que `init xs` calcula la lista resultante de eliminar su último elemento. Tanto `last` como `init` están indefinidas para la lista vacía.

Otro ejemplo típico es la definición recursiva de la función de concatenación de listas, también incluida en el preludio de Haskell:

```
infixr 5  ++
(++)          :: [a] -> [a] -> [a]
[]     ++ ys   = ys
(x:xs) ++ ys   = x : (xs ++ ys)
```

Igualdad y orden para listas

Otros ejemplos interesantes de recursión se encuentra en la definición de los métodos `(==)` y `(<=)` para listas, que aparecen en las siguientes declaraciones parametrizadas de ejemplar para las clases de tipos `Eq` y `Ord` (predefinidas en Haskell):

```
instance Eq a => Eq [a] where
    []     == []     =  True
    []     == (_:_)  =  False
    (_:_)  == []     =  False
    (x:xs) == (y:ys) =  x == y && xs == ys

instance Ord a => Ord [a] where
    []     <=   []     = True
    []     <=   (_:_)  = True
    (_:_)  <=   []     = False
    (x:xs) <=   (y:ys) = x < y || (x == y && xs <= ys)
```

Puesto que el tipo `String` es sinónimo de `Char`, que es ejemplar de `Eq` y `Ord`, son las definiciones anteriores quienes gobiernan la comparación de cadenas de caracteres en Haskell.

Esquemas de recursión para listas

Como ya hemos visto, las definiciones recursivas de funciones que operan con listas pueden usar los patrones `[]` y `(x:xs)` para distinguir el caso directo y el caso recursivo. Esta idea conduce a dos esquemas de recursión para listas:

```
-- Recursion natural para listas.
-- Presupone e :: b
--           f :: a -> b -> b

   h        :: [a] -> b
   h []      = e
   h (x:xs)  = f x (h xs)

-- Recursion final con acumulador para listas.
-- Presupone e :: b
--           f :: b -> a -> b

   h           :: b -> [a] -> b
   h  e []      = e
   h  e (x:xs)  = h (f e x) xs
```

Obsérvese la analogía entre estos dos esquemas y los esquemas con el mismo nombre estudiados en la sección 1.3 para el caso de funciones que operen con números naturales. Ambos esquemas se pueden extender fácilmente a funciones con más parámetros, como ya hemos visto en el caso de `(++)`. Muchas definiciones recursivas de funciones de procesamiento de listas se ajustan a ellos, aunque no todas. Por ejemplo, las definiciones recursivas de las funciones `last` e `init` toman como caso directo las listas unitarias en lugar de la lista vacía.

Principio de inducción estructural para listas generales

Muchas funciones de procesamiento de listas verifican leyes que se pueden formular matemáticamente, frecuentemente con ayuda de ecuaciones. Por ejemplo, la concatenación de listas satisface las dos leyes siguientes (donde `xs, ys, zs :: [a]` representan listas cualesquiera:

(N) `[]` es elemento neutro para la concatenación de listas:
`[] ++ xs = xs` y `xs ++ [] = xs`

(A) La concatenación de listas es asociativa:
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`

En los dos casos anteriores, la ley se puede entender como una propiedad `P xs` de la lista `xs`, la cual afirma que la ecuación se cumple cualesquiera que sean las listas `ys` y `zs`. En general, para demostrar que cualquier lista `xs :: [a]` verifica una propiedad `P` previamente definida, se puede utilizar el *principio de inducción estructural para listas generales*, formulado como sigue:

Supongamos una propiedad `P` para listas `xs :: [a]` que esté **definida por medio de una ecuación**. Si se demuestran las *bases* ($BS_{[]}$) y ($BS_{\perp}$) y el *paso inductivo* ($PI_{(:)}$), entonces se puede afirmar la *conclusión* (CL), siendo:

($BS_{[]}$) `P []`
La lista vacía cumple `P`.

($BS_{\perp}$) `P` $\perp$
La lista indefinida cumple `P`.

($PI_{(:)}$) `P xs` $\Rightarrow$ `P (x:xs)`
Si la lista `xs :: [a]` cumple `P` (*hipótesis de inducción* HI)
entonces al añadir cualquier cabeza `x :: a`,
la lista `(x:xs)` también cumple `P`.

(CL) $\forall$`xs :: [a] (P xs)`
`P xs` se cumple para cualquier lista `xs :: [a]`
(incluso para listas infinitas)

Como ejemplos de uso de este principio de inducción, presentamos a continuación la demostración de las dos leyes (N) y (A) de la concatenación, enunciadas más arriba. En los razonamientos usaremos la técnica de cálculo simbólico explicada en la sección 1.2, y escribiremos `(++.1)` y `(++.2)` para hacer referencia a las dos ecuaciones que forman la definición de `(++)`. En general, en adelante usaremos la notación `(f.i)` para referirnos a la ecuación número `i` de la definición de una función `f`.

Demostración de la propiedad (N)

La ecuación `[] ++ xs = xs` se cumple por definición de la función `(++)`. La ecuación `xs ++ [] = xs` se puede demostrar por inducción sobre `xs`, distinguiendo los tres casos previstos por el principio de inducción:

($BS_{[]}$) `[] ++ [] = []`.
Razonamiento: el lado izquierdo de esta ecuación se reducen fácilmente a `[]`, que coincide con el lado derecho.

($BS_{\perp}$) $\perp$ `++ [] =` $\perp$.
Razonamiento: el valor del lado izquierdo es $\perp$ porque $\perp$ `++ []` no se ajusta al lado izquierdo de ninguna de las dos ecuaciones de `(++)`.

($PI_{(:)}$) `(x:xs) ++ [] = x:xs`, suponiendo (HI) `xs ++ [] = xs`.
Razonamiento: el lado derecho de la ecuación está en forma normal. Usando cálculo simbólico y (HI), el lado izquierdo se puede reducir al lado derecho. En efecto:

$$\begin{array}{ll} & \underline{\texttt{(x:xs) ++ []}} \\ \longrightarrow_{(++).2} & \\ & \texttt{x:(}\underline{\texttt{xs++[]}}\texttt{)} \\ =_{(HI)} & \\ & \texttt{x:xs} \end{array}$$

Demostración de la propiedad (A)

Razonamos por inducción sobre `xs`, distinguiendo los tres casos previstos por el principio de inducción:

($BS_{[]}$) `[] ++ (ys ++ zs) = ([] ++ ys) ++ zs`.
Razonamiento: calculando con las ecuaciones de `(++)`, los dos lados de esta ecuación se reducen fácilmente a la expresión común `ys ++ zs`.

($BS_{\perp}$) $\perp$ `++ (ys ++ zs) =` ($\perp$ `++ ys) ++ zs`.
Razonamiento: el valor de ambos lados de la ecuación es $\perp$ porque las ecuaciones de `(++)` no permiten realizar ninguna reducción.

($PI_{(:)}$) `(x:xs) ++ (ys ++ zs) = ((x:xs) ++ ys) ++ zs`, suponiendo (HI) `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`.
Razonamiento: usando cálculo simbólico y (HI), ambos lados de la ecuación se pueden reducir a una expresión común. En efecto:

Cálculo para el lado izquierdo:

```
                (x:xs) ++ (ys ++ zs)
⟶(++).2
                x : (xs ++ (ys ++ zs)
=(HI)
                x : ((xs ++ ys) ++ zs)
```

Cálculo para el lado derecho:

```
                ((x:xs) ++ ys) ++ zs
⟶(++).2
                (x : (xs ++ ys)) ++ zs)
⟶(++).2
                x : ((xs ++ ys) ++ zs)
```

Principio de inducción estructural para listas finitas y terminadas

El principio de inducción para listas arbitrarias exige que la propiedad `P` esté definida por medio de una ecuación y no es válido en otro caso. Incluso si `P` está definida por medio de una ecuación, hay ocasiones en las que la base ($BS_\perp$) para la lista indefinida no se puede demostrar. Para esto casos es útil un *principio de inducción estructural para listas finitas y terminadas*, formulado como sigue:

Supongamos cualquier propiedad `P` para listas `xs :: [a]`. Si se demuestran la *base* ($BS_{[]}$) y el *paso inductivo* ($PI_{(:)}$), entonces se puede afirmar la *conclusión* (CL), siendo:

($BS_{[]}$) `P []`
La lista vacía cumple `P`.

($PI_{(:)}$) `P xs` $\Rightarrow$ `P (x:xs)`
Si la lista `xs :: [a]` cumple `P` (*hipótesis de inducción* HI)
entonces al añadir cualquier cabeza `x :: a`,
la lista `(x:xs)` también cumple `P`.

(CL) $\forall$`xs :: [a] (finTer xs` $\Rightarrow$ `P xs)`
`P xs` se cumple para cualquier lista finita y terminada `xs :: [a]`.

Más adelante encontraremos ejemplos interesantes de propiedades de listas finitas y terminadas que se pueden demostrar usando este principio, y que no son válidas para listas arbitrarias.

Inducción sobre la longitud de una lista

El razonamiento por inducción estructural es adecuado en la mayoría de los casos, pero presenta la limitación de que en el paso inductivo la hipótesis

de inducción siempre se refiere al *resto* de una lista. Para algunos problemas en los cuales la inducción estructural no basta, es necesario razonar por inducción sobre la longitud de las listas. Esta forma de inducción se formula como sigue:

Supongamos cualquier propiedad `P` para listas `xs :: [a]`. Si se demuestran la *base* (BS_0) y el *paso inductivo* ($PI_{>0}$), entonces se puede afirmar la *conclusión* (CL), siendo:

(BS_0) `P []`
La lista vacía (de longitud 0) cumple `P`.

($PI_{>0}$) $\forall$`xs' :: [a](length xs' < length xs` $\Rightarrow$ `P xs')`
$\Rightarrow$ `P xs`
Si la lista `xs :: [a]` tiene longitud $n > 0$
y cualquier lista `xs' :: [a]` de longitud menor que n cumple `P`
(*hipótesis de inducción* HI)
entonces la propia `xs` también cumple `P`.

(CL) $\forall$`xs :: [a] (finTer xs` $\Rightarrow$ `P xs)`
`P xs` se cumple para cualquier lista finita y terminada `xs :: [a]`.

Recursión e inducción en otros tipos de datos recursivos

Los esquemas de definición recursiva de funciones y los principios de razonamiento por inducción tienen sentido en cualquier tipo de datos recursivo, sirviendo para la programación de funciones y la demostración de propiedades de programas, respectivamente. Estas técnicas son particularmente útiles en el caso de los diferentes tipos de *árboles*, que estudiaremos en el capítulo 4.

Ejercicios

1. ¿Cuáles de las expresiones que siguen admiten tipo, y en caso afirmativo, cuál?

```
[7,9]   7:9   [(7,9)]  [[7],[9]]  ([7],[9])
7:[9]  [7]:9  (7,[9])   [7,[9]]    [7:[9]]
```

2. Escribe una expresión que contenga dos apariciones de la lista vacía, siendo la primera de ellas de tipo `[Int]` y la segunda de tipo `[Char]`.

3. ¿Cuáles de las ecuaciones siguientes se verifican para cualquier lista `xs`, y cuáles no?

```
[]:xs  =  xs    []:xs  =  [[],xs]  xs:[]  =  xs
xs:[]  =  [xs]  xs:xs  =  [xs,xs]
```

4. ¿Cuáles de las ecuaciones siguientes se verifican para cualquier lista `xs`, y cuáles no?

```
[[]] ++ xs  =  xs        [[]] ++ xs    =  [xs]
[[]] ++ xs  =  [[],xs]   [[]] ++ [xs]  =  [[],xs]
[xs] ++ []  =  [xs]      [xs] ++ [xs]  =  [xs,xs]
```

5. ¿Cuáles de las listas que siguen se ajustan al patrón `x:y:zs`, y cual es en cada caso la sustitución que produce el ajuste para las variables `x`, `y` y `zs`?

   ```
   [1]  [1,2]  [1,2,3]  [1,2,3,4]
   ```

6. Repite el ejercicio anterior, considerando el mismo patrón de antes y las siguientes cadenas de caracteres:

   ```
   "M"  "Ma"  "Mad"  "Madr"  "Madri"  "Madrid"
   ```

7. ¿Qué valor denota la expresión `[head xs] ++ tail xs` cuando `xs = []`?

8. Demuestra que el tipo principal de `(++)` es $\forall$`a [a] -> [a] -> [a]`. Usa las reglas de inferencia de tipos de Milner, eligiendo un contexto que incluya supuestos adecuados para los tipos de las funciones en las que se basa la definición recursiva de `(++)`.

9. El tipo de datos de las listas también se podría definir utilizando una constructora `Snoc` que que añade un nuevo elemento *al final* de una lista:

   ```
   data Liste a = Nil | Snoc (Liste a) a
   ```

   (a) Define las funciones análogas a `null`, `head`, `tail`, `last` e `init` para el tipo `Liste a`.

   (b) Programa una función `lista :: Liste a -> [a]` que efectúe la conversión natural entre los dos tipos de datos.

10. La siguiente declaración de tipo está pensada para representar listas utilizando la operación de concatenación como constructora:

    ```
    data CList a = Nil | Unit a | Conc (CList a) (CList a)
    ```

    La idea es que `Nil` representa `[]`, `Unit x` representa `[x]`, y `Conc xs ys` representa `xs ++ ys`.

    (a) Programa una función `lista :: CList a -> [a]` que efectúe la conversión natural entre los dos tipos de datos.

    (b) Declara el tipo `CList a` como ejemplar de las clases `Eq` y `Ord`. ¿Sería adecuada una declaración automática de ejemplar?

    (c) Investiga como se programarían algunas funciones básicas de procesamiento de listas, usando en lugar de `[a]` el tipo `CList a`.

3.2 Funciones que operan con listas

En esta sección vamos a estudiar una colección de funciones de procesamiento de listas que se encuentran predefinidas, bien en el preludio o en el módulo de biblioteca `List.hs` de Haskell.[1] Se trata de funciones genéricas y en muchos casos de orden superior, lo que las hace reutilizables en contextos de uso muy variados.

Presentaremos las funciones agrupadas por afinidad. Dentro de cada grupo, expondremos las definiciones de las funciones, propiedades y leyes de interés, y en ocasiones ejemplos de uso. La mayoría de las leyes se demuestran mediante razonamientos por inducción, que dejaremos como ejercicio para el lector.

Entre las propiedades de interés incluiremos una indicación de la eficiencia, dando una cota superior del tiempo de ejecución expresada con ayuda de la notación $\mathcal{O}$. Nos limitaremos a enunciar y explicar informalmente estas estimaciones de complejidad, sin entrar en su demostración formal. En el capítulo 3 del texto [9] y en el capítulo 7 de [1] se explican algunas técnicas básicas para analizar la eficiencia de programas funcionales, siguiendo el criterio de que el tiempo de ejecución se mida en proporción al número de pasos de reducción necesarios para el cómputo.

Descomposición de de una lista

Las tareas más elementales de procesamiento de listas consisten simplemente en reconocer la lista vacía, y en descomponer una lista no vacía separando un elemento. Las funciones que ejecutan estas tareas son `null`, `head`, `tail`, `last` e `init`, ya estudiadas en la sección anterior.

Acerca de la *eficiencia* de estas funciones puede afirmarse que, dada una lista finita y terminada `xs` de longitud n:

- El tiempo de ejecución de `null xs`, `head xs` y `tail xs` es $\mathcal{O}(1)$.
- El tiempo de ejecución de `last xs` e `init xs` es $\mathcal{O}(n)$.

Longitud y elementos de una lista

Para calcular la longitud de una lista dada, y para averigüar si un elemento dado es o no miembro de una lista dada, Haskell ofrece las siguientes funciones:

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

[1] Al iniciar una sesión Hugs 98 en la que se desee probar funciones de proceso de listas, se recomienda cargar el módulo `List.hs`.

```
elem         :: Eq a => a -> [a] -> Bool
elem y []     = False
elem y (x:xs) = y == x || elem y xs

notElem         :: Eq a => a -> [a] -> Bool
notElem y []     = True
notElem y (x:xs) = y /= x && notElem y xs
```

Suponiendo que la lista `xs` dada como parámetros sea finita y terminada de longitud n, los tiempos de ejecución de estas tres funciones son $\mathcal{O}(n)$ en el caso peor. De hecho, la evaluación de `length xs` causa un recorrido completo de la lista `xs`, y solo termina sin error en el caso de que dicha lista sea finita y terminada.

Por el contrario, la evaluación de espresiones de la forma `elem y xs` o `notElem y xs` a veces puede terminar sin recorrer por completo la lista `xs`, incluso en el caso de que esta sea infinita. Para comprender esto, consideremos la función `from` definida como sigue:[2]

```
from   :: Int -> [Int]
from n  = n : from (n+1)
```

Las siguientes pruebas de ejecución muestran que `elem` y `notElem` pueden completar con éxito algunos cálculos con *listas infinitas* generadas por `from`. Ello es posible gracias a la estrategia de evaluación perezosa.

```
> elem 3 (from 0)
True
> notElem 3 (from 0)
False
```

Haskell también ofrece una función `(!!)` definida como operador infijo, tal que `xs !! i` calcula el elemento situado en la posición `i` de la lista `xs`, contando a partir de la posición 0:

```
(!!)               :: [a] -> Int -> a
(x:_)  !! 0         = x
(_:xs) !! n | n > 0 = xs !! (n-1)
(_:_)  !! _         = error "Prelude.!!: negative index"
[]     !! _         = error "Prelude.!!: index too large"
```

Como puede deducirse de la definición anterior, para una lista finita y terminada `xs` de longitud n, `xs !! i` se calcula en tiempo $\mathcal{O}(n)$ (en el caso peor) y solo está definida cuando el valor de `i` está comprendido entre 0

[2] `from` está predefinida en Haskell con el nombre `enumFrom`.

(posición del primer elemento) y $n-1$ (posición del último elemento). En el caso de que `xs` no sea finita y terminada, `xs !! i` puede calcularse con éxito siempre que `xs` incluya un elemento en la posición `i`. Por ejemplo, es posible calcular elementos de listas infinitas generadas por `from`:

```
> from 3 !! 0
3
> from 3 !! 1
4
> from 3 !! 2
5
```

Concatenación y aplanamiento de listas

La *concatenación* de dos listas del mismo tipo se calcula mediante la función `(++)` estudiada en la sección 3.1. La evaluación de `xs ++ ys` solo termina con éxito cuando `xs` es finita y terminada de una cierta longitud n, y requiere tiempo $\mathcal{O}(n)$.

A continuación se enuncian algunas leyes válidas, relacionadas con la operación de concatenación. Las dos primeras han sido demostradas en la sección 3.1. En todas las leyes, se supone: `x :: a`; `xs, ys, zs :: [a]`.

```
[] ++ xs            =  xs                      ([] es neutro por la izqda.)
xs ++ []            =  xs                      ([] es neutro por la dcha.)
xs ++ (ys ++ zs)    =  (xs ++ ys) ++ zs        ((++) es asociativa)
last (xs ++ [x])    =  x
init (xs ++ [x])    =  xs
init xs ++ last xs  =  xs                      si null xs = False
length (xs ++ ys)   =  length xs + length ys
xs ++ ys            =  xs                      si xs es inconclusa
```

Haskell ofrece además una función llamada `concat` que espera como parámetro una lista de listas xss :: [[a]] y devuelve el *aplanamiento* de `xss`, calculado como la concatenación reiterada de las diferentes listas xs :: [a] que forman `xss`:

```
concat         :: [[a]] -> [a]
concat []       =
concat (xs:xss) = xs ++ concat xss
```

Un ejemplo sencillo de uso de `concat` es el siguiente:

```
> concat [[0,1],[],[2,3,4],[],[5]]
[0,1,2,3,4,5]
```

En general, si `xss` es una lista finita y terminada formada por listas finitas y terminadas, `concat xss` se evalúa en tiempo $\mathcal{O}(n)$, siendo n la suma de las longitudes de las diferentes listas `xs` miembros de `xss`.

Es fácil demostrar la siguiente ley, que relaciona los comportamientos de `concat` y `(++)`:

```
concat [xs,ys]  =  xs ++ ys
```

Diferencia entre dos listas

La *diferencia* entre dos listas `xs, ys :: [a]` es la lista resultante de eliminar de `xs` cada uno de los elementos de `ys`. Más exactamente, para cada elemento `y` de `ys` que aparezca en `xs`, se desea eliminar la primera aparición. Por ejemplo:

```
> "aradios" \\ "as"
"radio"
```

La función `(\\)` se define junto con una función auxiliar `delete`:

```
delete          :: Eq a => a -> [a] -> [a]
delete y []      = []
delete y (x:xs)
  | y == x       = xs
  | otherwise    = x : delete y xs

(\\)         :: Eq a => [a] -> [a] -> [a]
xs \\ []      = []
xs \\ (y:ys)  = delete y xs \\ ys
```

Obsérvese que `delete y xs` devuelve el resultado de eliminar de `xs` la primera aparición de `y`, si es que hay alguna; y devuelve `xs` en otro caso.

Suponiendo que `xs` e `ys` sean ambas finitas y terminadas, de longitudes m y n respectivamente, el tiempo de ejecución de `xs \\ ys` es $\mathcal{O}(mn)$ en el caso peor (cuando ninguno de los elementos de `ys` aparece en `xs`). Se puede demostrar además que la función `(\\)` verifica las dos leyes siguientes:

```
(xs ++ ys) \\ xs  =  ys
xs \\ [head xs]   =  tail xs   si null xs = False
```

Inversa de una lista

La *inversa* de una lista `xs :: [a]` es la lista `ys :: [a]` formada por los elementos de `xs` tomados en orden inverso. Una definición ingenua de una función que calcula la inversa de una lista es como sigue:

```
inversa       :: [a] -> [a]
inversa []     = []
inversa (x:xs) = inversa xs ++ [x]
```

La definición anterior usa el esquema de recursión natural. Es fácil razonar que, suponiendo una lista `xs` finita y terminada de longitud n, el cálculo de `inversa xs` requiere tiempo $\mathcal{O}(n^2)$, lo cual es muy ineficiente.

La función de inversión de listas `reverse` incluida en el preludio de Haskell utiliza una definición diferente, basada en el esquema de recursión final con acumulador:

```
reverse       :: [a] -> [a]
reverse xs     = reverseOnto [] xs

reverseOnto          :: [a] -> [a] -> [a]
reverseOnto ys []     = ys
reverseonto ys (x:xs) = reverseOnto (x:ys) xs
```

Claramente, `reverse xs` también calcula correctamente la inversa de `xs`. Además, `reverse xs` se puede evaluar en tiempo $\mathcal{O}(n)$, siempre que `xs` sea una lista finita y terminada de longitud n.

Es posible incluso *derivar* la función `reverse` como optimización de `inversa`, utilizando la técnica de *plegado-desplegado* explicada al final de la sección 1.3. Para ello, hay que comenzar *especificando* `reverseOnto` como generalización de `inversa`, mediante la ecuación:

```
reverseOnto ys xs = inversa xs ++ ys
```

Usando esta especificación ineficiente de `reverseOnto` junto con las ecuaciones que definen a las funciones `inversa` y `(++)`, la transformación por plegado-desplegado produce las dos ecuaciones de la definición eficiente de `reverseOnto` mostradas más arriba. Se deja como *ejercicio* para el lector el razonar los detalles de esta transformación. Para ello es necesario utilizar la ley asociatividad de `(++)`, así como la ley que asegura que `[]` es elemento neutro de `(++)`.

Partición de una lista

Las funciones predefinidas `take` y `drop` sirven para partir una lista dada, separando los `n` primeros elementos de los restantes:

```
take                :: Int -> [a] -> [a]
take n _   | n <= 0  = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs

drop                :: Int -> [a] -> [a]
drop n xs | n <= 0  = xs
drop _ []           = []
drop n (_:xs)       = drop (n-1) xs
```

Analizando las definiciones de estas dos funciones, se observa que:

- Para `n` ≥ 0, `(take n xs)` se evalúa en tiempo $\mathcal{O}(n)$ y calcula la lista formada por los `n` primeros elementos de `xs`. El cómputo tiene éxito incluso si `xs` es infinita:

```
> take 5 (from 0)
[0,1,2,3,4]
```

- Para `n` ≥ 0, `(drop n xs)` se evalúa en tiempo $\mathcal{O}(n)$ y calcula la lista resultante de eliminar los `n` primeros elementos de `xs`. El cómputo solo acaba con éxito si la lista resultante es finita y terminada.

- Para `n` < 0, `(take n xs)` devuelve la lista vacía y `(drop n xs)` devuelve `xs` (que solo se puede visualizarse en tiempo finito si es finita y terminada). En ambos casos, el tiempo de cómputo es $\mathcal{O}(1)$.

Las funciones `take` y `drop` se pueden combinar. Son posibles, por ejemplo, los cómputos siguientes:

```
> take 5 (drop 10 (from 0))
[10,11,12,13,14]
> take 2 [10,11,12,13,14] ++ drop 2 [10,11,12,13,14]
[10,11,12,13,14]
```

Las funciones `take` y `drop` verifican varias leyes que se pueden demostrar por inducción:

```
take n xs ++ drop n xs  =  xs
drop 1 xs               =  tail xs              si null xs = False
take (length xs - 1)    =  init xs              si null xs = False
take m . take n         =  take (min m n)       si m, n ≥ 0
drop m . drop n         =  drop (m+n)           si m, n ≥ 0
take m . drop n         =  drop n . take (m+n)  si m, n ≥ 0
```

La primera de las leyes anteriores muestra que el efecto combinado de `take` y `drop` es dividir una lista en dos partes. Esta tarea la realiza la función `splitAt`, que se puede definir ingenuamente como sigue:

```
splitAt      :: Int -> [a] -> ([a], [a])
splitAt n xs  = (take n xs, drop n xs)
```

Esta definición es ineficiente, debido a que las llamadas a `take` y `drop` procesan *dos veces* la lista `xs` dada como parámetro. En el preludio de Haskell aparece una versión más eficiente de `splitAt`, que procesa *una sola vez* la lista dada como parámetro:

```
splitAt                :: Int -> [a] -> ([a], [a])
splitAt n xs | n <= 0   = ([],xs)
splitAt _ []            = ([],[])
splitAt n (x:xs)        = (x:xs',xs'')
                          where (xs',xs'') = splitAt (n-1) xs
```

Es posible derivar esta definición optimizada de `splitAt`, transformando la definición ingenua dada más arriba mediante la técnica de plegado-desplegado. En este caso, la etapa previa de generalización que a veces precede al plegado-desplegado no es necesaria. Se deja como ejercicio para el lector el completar los detalles de esta transformación.

Otro criterio de partición de una lista en dos partes consiste en utilizar un predicado (dado como función booleana) para decidir el punto de separación. Esta idea da lugar a las siguientes funciones de orden superior, predefinidas en el preludio de Haskell:

```
takeWhile             :: (a -> Bool) -> [a] -> [a]
takeWhile p []         = []
takeWhile p (x:xs)
  | p x                = x : takeWhile p xs
  | otherwise          = []

dropWhile            :: (a -> Bool) -> [a] -> [a]
dropWhile p []        = []
dropWhile p xs@(x:xs')
  | p x               = dropWhile p xs'
  | otherwise         = xs

span, break         :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
  | p x              = (x:ys, zs)
  | otherwise        = ([],xs)
                       where (ys,zs) = span p xs'
break p              = span (not . p)
```

El comportamiento de estas funciones puede describirse así:

- `(takeWhile p xs)` calcula un segmento inicial de `xs`, hasta el primer elemento que no cumpla el predicado `p`, exclusive. El tiempo de cómputo es proporcional a la longitud de la lista calculada.
- `(dropWhile p xs)` calcula la lista resultante de eliminar un segmento inicial de `xs`, hasta el primer elemento que no cumpla el predicado `p`, exclusive. El tiempo de cómputo es proporcional al número de elementos eliminados.

- `(span p xs)` calcula una pareja de listas equivalente a `(takeWhile p xs, dropWhile p xs)`, pero evitando procesar dos veces `xs`. Por tanto, se efectúa una partición de `xs` en dos listas, tomando como punto de división el primer elemento de `xs` que no cumpla `p`. Este queda como cabeza de la segunda lista de la partición.

- `(break p xs)` calcula una partición de `xs` de manera semejante a `span`, pero tomando como punto de división el primer elemento que cumpla `p`. Este queda como cabeza de la segunda lista de la partición.

los dos ejemplos que siguen ayudarán a entender el funcionamiento de `span` y `break`:

```
> span isDigit "10000 moscas acudieron al panal de rica miel."
("10000"," moscas acudieron al panal de rica miel.")
> break (== ' ') "Vini, vidi, vinci."
("Vini,"," vidi, vinci.")
```

Es fácil demostrar la siguiente ley, que relaciona los comportamientos de `takeWhile` y `dropWhile`:

`takeWhile p xs ++ dropWhile p xs = xs` si `p` es total sobre `xs`

La condición de que `p` sea *total* sobre `x` significa suponer que `(p x)` siempre calcula uno de los dos resultados `True` o `False` (y nunca $\perp$) para cualquier elemento `x` de `xs`.

Transformación de una lista

Una operación útil para resolver muchos problemas consiste en transformar una lista dada `xs` con elementos $\texttt{x}_i$ en una nueva lista `ys` con elementos `f` $\texttt{x}_i$, calculada aplicando una misma función `f` a cada uno de los elementos de la lista de partida. La función de orden superior predefinida `map` realiza este cómputo, tomando como parámetros a `f` y a `xs`:

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

Dada `f :: a -> b`, se dice que `map f :: [a] -> [b]` es la *transformación de listas* inducida por `f`. El tipo `b` de los elementos de la lista transformada puede ser diferente del tipo `a` de los elementos de la lista original. Sin embargo, los parámetros `a` y `b` se sustituyen por tipos concretos en cada uso particular de `map`, y no está prohibido sustituirlos por el mismo tipo. Por ejemplo, la siguiente expresión utiliza `map` para calcular la lista formada por los cuadrados de los quince primeros números enteros, de `2` en

adelante. En este caso, el parámetro `f` de `map` se sustituye por la función anónima `\x -> x*x :: Int -> Int`, con lo cual `a` y `b` se sustituyen ambos por `Int`:

```
> map (\x -> x*x) (take 15 (from 2))
[4,9,16,25,36,49,64,81,100,121,144,169,196,225,256]
```

Un ejemplo interesante de uso de `map` se encuentra en las defniciones predefinidas de las funciones `any` y `all`. Estas dependen de otras dos funciones `or` y `and`, también predefinidas:

```
or, and   :: [Bool] -> Bool
or []      = False
or (x:xs)  = x || or xs
and []     = True
and (x:xs) = x && and xs

any, all  :: (a -> Bool) -> [a] -> Bool
any p      = or  . map p
all p      = and . map p
```

Claramente, `or xs` (resp. `and xs`) calcula la *disyunción* (resp. *conjunción*) lógica de todos los valores booleanos que forman la lista `xs`. Por su parte, `any p` (resp. `all p`) se comporta como un predicado que comprueba si *algún elemento* (resp. *cualquier elemento*) de una lista dada verifica la propiedad `p`. Suponiendo `xs :: [a]` finita y terminada de longitud n, los tiempos de ejecución de `any p xs` y `all p xs` son $\mathcal{O}(n)$ en el caso peor. Sin embargo, en ciertos casos estos cómputos terminan también aunque `xs` sea infinita, como muestran los ejemplos siguientes.

```
> any isDigit "La plaza del 2 de Mayo"
True
> all isDigit "La plaza del 2 de Mayo"
False
> any odd (from 0)
True
> all odd (from 0)
False
```

La función `map` verifica muchas leyes que se pueden verificar razonando con ayuda de los principios de extensionalidad e inducción. Algunas de ellas se presentan a continuación.

```
map id            =  id
map f . map g     =  map (f . g)
map f (xs ++ ys)  =  map f xs ++ map f ys
map f . concat    =  concat . map (map f)
map f . reverse   =  reverse . map f
(map f)^-1        =  map f ^-1              si f ^-1 existe
f . head          =  head . map f           si f es estricta
map f . tail      =  tail . map f
```

Por cumplirse las dos primeras de las leyes anteriores, se dice que `map` es un *funtor*; y por cumplirse la tercera ley, se dice que `map f` es *distributivo* con respecto a `(++)`.

Finalizamos este apartado presentando una variante de `map` que será útil en la sección 3.3. Se trata de una función que *no está predefinida* en Haskell. Para probarla se recomienda utilizar el módulo Hugs 98 del apéndice A.10. Su definición es como sigue:

```
maybeMap         :: (a -> Maybe b) -> [a] -> [b]
maybeMap f []     = []
maybeMap f (x:xs) = case f x of
                      Just y  -> y : maybeMap f xs
                      Nothing -> maybeMap f xs
```

Obsérvese que la función que transforma cada elemento de la lista puede devolver resultados de la forma `Just y` o `Nothing`. La transformación recolecta todos los valores `y` correspondientes a los resultados `Just y`, e ignora los resultados `Nothing`.

En la definición de `maybeMap` hemos usado por primera vez la construcción `case`. En general, Haskell permite construir expresiones `case` de la forma

```
case e of
  p1 -> e1
  p2 -> e2

    ...

  pk -> ek
```

donde `e`, `e1`, ..., `ek` son expresiones, y `p1`, ..., `pk` son patrones lineales (como los que se usan en los lados izquierdos de las definiciones de funciones) del mismo tipo que `e`. Una expresión `case` se calcula evaluando `e` lo que sea necesario, hasta encontrar *el primer* patrón `pi` al cual `e` se ajuste. A continuación, se sigue calculando con `ei`, teniendo en cuenta la sustitución producida por el ajuste para aquellas variables de `pi` que aparezcan en `e`. En otras palabras, la evaluación de una expresión `case` como la anterior tiene el mismo comportamiento que tendría la evaluación de una expresión `fun e`, suponiendo que `fun` fuese una función definida por las ecuaciones

```
fun p1 = e1
fun p2 = e2
      ...

fun pk = ek
```

Filtrado de una lista

Otra operación de uso frecuente consiste en procesar una lista eliminando de ella todos los elementos que no cumplan una propiedad dada. Esta operación, llamada *filtrado* de la lista, se puede realizar en Haskell con ayuda de la función predefinida `filter`:

```
filter           :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x:xs)
   | p x          = x : filter p xs
   | otherwise    = filter p xs
```

Vemos que `filter` es una función de orden superior, cuyo primer parámetro corresponde a la propiedad `p` con respecto a la cual se hace el filtrado. Como ejemplo de uso de `filter`, es inmediato definir una función que extrae los *dígitos* de una cadena de caracteres dada; y a partir de esta otra función que calcula el *primer dígito* de una cadena de caracteres:

```
digitos :: String -> String
digitos  = filter isDigit

primerDigito    :: String -> Char
primerDigito xs  = case digitos xs of
                     []    -> '0'
                     (x:_) ->  x
```

En el apéndice A.11 se encuentra un pequeño módulo Hugs 98 que incluye las definiciones anteriores y dos pruebas sencillas. Es interesante observar que `primerDigito` se ha programado de tal manera que devuelva el dígito `'0'` en el caso de que la cadena de caracteres recibida como parámetro no contuviese ningún dígito. Para expresar este comportamiento, la definición de `primerDigito` efectúa una distinción de casos por medio de la construcción `case`, ya comentada más arriba.

Finalizamos este apartado enunciando algunas leyes de interés que relacionan el comportamiento de la función `filter` con otras funciones ya conocidas:

```
filter p (xs ++ ys)     =  filter p xs ++ filter p ys
filter p (filter q xs)  =  filter q (filter p xs)
                           si p, q son totales sobre xs
filter p . concat       =  concat . map (filter p)
filter p . map f        =  map f . filter (p . f)
```

Cremallera entre dos listas

Dadas dos listas `xs` e `ys` del mismo tipo, la función `zip` (*cremallera*) calcula una tercera lista `zs` que contiene la pareja `(xs !! i, ys !! i)` en cada posición `i`. En el caso de que las dos listas `xs` e `ys` no tengan la misma longitud, la longitud de `zs` será el *mínimo* de las longitudes de las dos listas dadas. La definición de `zip` en el preludio equivale a la siguiente:

```
zip               :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys)  = (x,y) : zip xs ys
zip   _      _     = []
```

El tiempo de ejecución de `zip xs ys` es $\mathcal{O}(n)$, siendo n el mínimo entre las longitudes de `xs` e `ys`. La evaluación solo puede terminar en tiempo finito si al menos una de las dos listas (la más corta) es finita y terminada; pero no importa si la otra es infinita. Por ejemplo:

```
> zip (take 5 (from 0)) (from 5)
[(0,5),(1,6),(2,7),(3,8),(4,9)]
```

En este curso utilizaremos también la versión no curryficada de `zip`, que *no está predefinida* en Haskell:

```
zipp  :: ([a],[b]) -> [(a,b)]
zipp   = uncurry zip
```

La función inversa de `zipp`, que convierte una lista de parejas en una pareja de listas, se puede definir de forma muy concisa utilizando la función de orden superior `pair` estudiada en la sección 2.4, combinada con la función `map`:

```
unzip  :: [(a,b)] -> ([a],[b])
unzip   = pair (map fst, map snd)
```

La definición anterior es ineficiente porque la evaluación de `unzip xys` causa un doble recorrido de la lista de parejas `xys`. Aplicando la transformación de plegado-desplegado se puede derivar una versión más eficiente, equivalente a la que aparece en el preludio de Haskell:

```
unzip  :: [(a,b)] -> ([a],[b])
unzip ([])          = ([],[])
unzip ((x,y) : xys)  = (x:xs, y:ys)
                       where (xs, ys) = unzip xys
```

Con esta definición, `unzip xys` se evalúa en tiempo $\mathcal{O}(n)$, suponiendo que `xys :: [(a,b)]` sea finita y terminada de longitud n.

Las funciones `zip` y `zipp` se pueden considerar como casos particulares de dos funciones de orden superior `zipWith` y `zippWith`, definidas como sigue:

```
zipWith                 :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys)  = f x y : zipWith f xs ys
zipWith _   _      _     = []

zippWith    :: (a -> b -> c) -> ([a], [b]) -> [c]
zippWith f   = uncurry (zipWith f)
```

De las dos funciones anteriores, tan solo la primera está predefinida en Haskell. Es fácil demostrar que:

```
zip   =  zipWith (\x -> \ y -> (x,y))
zipp  =  zippWith (\x -> \ y -> (x,y))
```

Otras dos leyes válidas relacionadas con las operaciones cremallera son:

```
cross (map f,map g) . unzip  =  unzip . map (cross (f,g))
zipp . cross (map f, map g)  =  map (cross (f,g)) . zipp
```

Aquí intervienen también las funciones de orden superior `pair` y `cross`, estudiadas en la sección 2.4.

Procesamiento de listas de números

Haskell ofrece también algunas funciones predefinidas útiles para cálculos en los que intervengan listas de números. Por ejemplo, las dos funciones definidas a continuación procesan en tiempo $\mathcal{O}(n)$ una lista de números de longitud n, devolviendo como resultado la *suma* o el *producto* de los miembros de la lista, respectivamente:

```
sum, product     :: Num a => [a] -> a
sum []            = 0
sum (x:xs)        = x + sum xs
product []        = 1
product (x:xs)    = x * product xs
```

Las definiciones de `sum` y `product` incluidas en el preludio no son idénticas a las anteriores, sino otras equivalentes que emplean un esquema de recursión final con acumulador. Otro tanto se puede decir a propósito de las funciones siguientes, que calculan el *máximo* (respectivamente, el mínimo) de una lista no vacía de elementos de un tipo de la clase `Ord`:

```
maximum, minimum :: Ord a => [a] -> a
maximum (x:xs)
   | null xs     = x
   | otherwise   = x `max` maximum xs
minimum (x:xs)
   | null xs     = x
   | otherwise   = x `min` minimum xs
```

Representación de funciones como listas

Concluimos esta sección presentando una aplicación de la programación con listas a un problema de cálculo numérico. En la sección 2.1 hemos mencionado ya el módulo Hugs 98 del apéndice A.1, que implementa el método de Newton para el cálculo aproximado de ceros de funciones. En la práctica, puede presentarse el problema de calcular un cero de una función para la cual no se conoce una fórmula explícita, pero sí una serie de valores tabulados. En estos casos, es posible representar la tabla de valores conocidos de una cierta función `f` como una lista de tipo `[(Float,Float)]` formada por parejas `(x,y)` tales que `f x = y`.

El módulo Hugs 98 del apéndice A.12 permite aplicar el método de Newton a funciones representadas de esta manera. Este módulo se ha diseñado importando el módulo `Newton` del apéndice A.1 y empleando recursos elementales de programación con listas, incluyendo un algoritmo de búsqueda para explorar la lista de parejas que representa los valores tabulados de la función cuyo cero se desea calcular.

Ejercicios

1. Demuestra que las propiedades siguientes se verifican para cualquier lista finita y teminada `xs`:

   ```
   init (xs ++ [x])       =  xs
   last (xs ++ [x])       =  x
   init xs ++ [last xs]   =  xs                          si not (null xs)
   init xs                =  take ((length xs)-1) xs     si not (null xs)
   ```

2. Demuestra la siguiente propiedad:

   ```
   (xs ++ ys) !!  i  =  if i < n then xs !!  i else ys !!  (i-n)
                        where n = lentgh xs
   ```

3. Demuestra que la ecuación `(xs ++ ys ) \\ xs = ys` es válida para cualquier lista finita y terminada `xs :: [a]` y cualquier lista `ys :: [a]`. ¿Vale esta propiedad si `xs` está inconclusa?.

4. Define una función `esPermut :: Eq a => [a] -> [a] -> Bool` que reconozca si las dos listas dadas como parámetros son una permutación

de la otra.
Pista: usa la función que calcula la diferencia entre dos listas.

5. Razonando por inducción, demuestra que cualquier lista *finita y terminada* `xs :: [a]` verifica la ecuación `inversa (inversa xs) = xs`. Usa la definición ingenua de la función `inversa`. ¿Se cumple esta propiedad para listas inconclusas?.

6. Demuestra las siguientes propiedades de las funciones `take` y `drop`:

```
take n xs ++ drop n xs  =  xs
take m . take n         =  take (min m n)       si m, n >= 0
drop m . drop n         =  drop (m+n)           si m, n >= 0
take m . drop n         =  drop n . take (m+n)  si m, n >= 0
```

7. Demuestra que la propiedad

```
takeWhile p xs ++ dropWhile p xs = xs
```

se cumple siempre que `p` sea *total sobre* `xs`. Esta condición quiere decir que para cualquier elemento `x` de `xs`, el valor de `p x` debe ser `True` o `False`, nunca $\perp$.

8. Define funciones

```
takeLast, dropLast :: Int -> [a] -> [a]
```

que tengan un comportamiento análogo al de `take` y `drop`, pero procesando la lista desde su extremo derecho. Explica el comportamiento de tus funciones en los "casos excepcionales" (argumento entero negativo o mayor que la longitud de la lista).

9. Define una función `separaLinea :: String -> (String, String)` que separe la *primera línea* de un texto representado como cadena de caracteres, del resto del texto. Más exactamente, `(separaLinea xs)` debe calcular una pareja `(us,vs)`, donde `us` sea la parte de `xs` anterior al primer salto de línea y `vs` sea la parte de `xs` posterior al primer salto de línea. Si `xs` no incluye ningún salto de línea, el resultado calculado debe ser `(xs,[])`.

10. ¿Cuál es el valor de `map f` $\perp$? ¿Y el valor de `map` $\perp$ `[]`?

11. ¿Cuál es el tipo principal de `map map`?

12. Demuestra las siguientes propiedades de `map`:

```
map id            =  id
map (f . g)       =  map f . map g          map es un funtor
map f (xs ++ ys)  =  map f xs ++ map f ys
map f . concat    =  concat . map (map f)
map f . reverse   =  reverse . map f
(map f)^-1        =  map f^-1               si f^-1 existe
```

13. Es posible definir `filter` usando `concat` y `map`. Completa la definición que sigue:

```
filter p  =  concat . map envasa
             where envasa = ···
```

14. Demuestra las siguientes propiedades de la función `filter`, indicando en cada caso si es necesaria alguna suposición especial acerca del predicado `p`.

```
filter p (xs ++ ys)     =  filter p xs ++ filter p ys
filter p (filter q xs)  =  filter q (filter p xs)
                           si p, q son totales sobre xs
filter p . concat       =  concat . map (filter p)
filter p . map f        =  map f . filter (p . f)
```

15. Recuerda el tipo sinónimo `Persona` estudiado en la Sección 2.6. Dada una lista de tipo `[Persona]`, se quieren resolver los problemas siguientes:

    (a) Calcular las edades de todas las personas de la lista.
    (b) Separar a todas las personas de la lista que sean adolescentes.
    (c) Calcular la media de las edades de los adolescentes de la lista.
    (d) Calcular los DNIs de todas las personas adultas de la lista.

    Define en cada caso una función adecuada para resolver el problema, declarando su tipo.

16. Demuestra que la siguiente propiedad es válida para listas finitas y terminadas:

```
length (zip xs ys) = min (length xs) (length ys)
```

17. Demuestra que la siguiente propiedad es válida para toda lista de parejas `xys :: [(a,b)]` que no contenga a $\perp$ como elemento.

```
zipp (unzip xys) = xys
```

18. Demuestra las propiedades siguientes usando inducción:

```
cross (map f,map g) . unzip  =  unzip . map (cross (f,g))
zipp . cross (map f, map g)  =  map (cross (f,g)) . zipp
```

19. La primera de las dos propiedades del ejercicio anterior se puede demostrar utilizando solamente razonamiento ecuacional y sin necesidad de inducción, siempre que se acepte la validez de las propiedades siguientes:

```
map (f . g)             =  map f . map g
pair(f,g) . h           =  pair(f . h, g . h)
cross(f,g) . pair(h,k)  =  pair(f . h, g . k)
fst . cross(f,g)        =  f . fst
snd . cross(f,g)        =  g . snd
```

Construye una demostración.

20. Demuestra que los tres items siguientes corresponden a definiciones equivalentes de una función que comprueba si una lista dada como parámetro está o no ordenada:

    (a) Definición recursiva directa:

    ```
    ordenada          :: Ord a => [a] -> Bool
    ordenada []        = True
    ordenada [x]       = True
    ordenada (x:y:zs)  = x <= y && ordenada (y:zs)
    ```

    (b) Definición basada en `and`, `map` y `zip`:

    ```
    ordenada    :: Ord a => [a] -> Bool
    ordenada xs  = and (map (uncurry (<=)) (zip xs (tail xs)))
    ```

    (c) Definición basada en `and` y `zippWith`, formulada en estilo combinatorio:

    ```
    ordenada  :: Ord a => [a] -> Bool
    ordenada   = and . zippWith (<=) . pair (id,tail)
    ```

21. Demuestra que los tres items siguientes corresponden a definiciones equivalentes de una función que calcula el producto escalar de una pareja de vectores representados como listas de números:

    (a) Definición recursiva directa:

    ```
    pe                   :: Num a => ([a],[a]) -> a
    pe ((x:xs), (y:ys))   = x*y + pe (xs, ys)
    pe (  _   ,   _   )   = 0
    ```

    (b) Definición basada en `sum`, `map` y `zip`:

    ```
    pe          :: Num a => ([a],[a]) -> a
    pe (xs, ys)  = sum (map (uncurry (*)) (zip xs ys))
    ```

    (c) Definición basada en `sum` y `zippWith`, formulada en estilo combinatorio:

    ```
    pe :: Num a => ([a],[a]) -> a
    pe  = sum . zippWith (*)
    ```

3.3 Sucesiones y listas intensionales

La mayoría de las funciones de procesamiento de listas estudiadas en la sección anterior se han programado empleando definiciones recursivas. Haskell y otros lenguajes funcionales disponen también de algunos recursos potentes para definir funciones que construyen listas sin utilizar recursión explícita. Se trata de las *sucesiones* y las *listas intensionales*, que vamos a estudiar en esta sección.

Sucesiones de enteros

Haskell permite utilizar algunas notaciones especiales para construir sucesiones de números enteros (bien sean de tipo `Int` o de tipo `Integer`). Se consideran cuatro tipos de sucesiones:

(a) Sucesiones `[n..]`.
Una sucesión de este tipo tiene como valor la lista infinita formada por todos los números enteros consecutivos a partir de `n`; es decir, la lista `[n, n+1, n+2, ···]`. A efectos de cálculo, la expresión `[n..]` se considera equivalente a `(iterate (+1) n)`, siendo `iterate` la función de orden superior definida como sigue:

```
iterate          :: (a -> a) -> a -> [a]
iterate f x       = x : iterate f (f x)
```

La definición de `iterate` está incluida en el preludio de Haskell. Como vemos, `(iterate f x)` calcula la lista infinita `[x, f x, f (f x), ···]`.

(b) Sucesiones `[n..l]`.
Una sucesión de este tipo tiene como elementos a los números menores o iguales que `l` de la sucesión infinita `[n..]`. Por ejemplo: `[5..8] = [5,6,7,8]`; `[5..5] = [5]`; `[5..2] = []`. A efectos de cálculo, la expresión `[n..l]` se considera equivalente a `takeWhile (<=l) [n..]`.

(c) Sucesiones `[n,m..]`.
Una sucesión de este tipo tiene como valor la lista infinita `[n, n+d, n+2d, ···]`, siendo `d = (m-n)`. Por lo tanto, a efectos de cálculo la expresión `[n,m..]` equivale a `(iterate (+(m-n)) n)`.

(d) Sucesiones `[n,m..l]`.
A efectos de cómputo, una sucesión de este tipo se considera equivalente a

```
takeWhile p [n,m..]
where p = if n <= m then (<=l) else (>=l)
```

Por lo tanto, `[n,m..l]` se evalúa de diferente modo según que se tenga `n <= m` o `n > m`. Cuando `n <= m`, la sucesión `[n,m..l]` está formada por los números menores o iguales que `l` de la sucesión `[n,m..]`. Cuando `n > m`, la sucesión `[n,m..l]` está formada por los números mayores o iguales que `l` de la sucesión `[n,m..]`. Por ejemplo: `[5,7..12] = [5,7,9,11]`; `[5,5..12] = [5,5..5] = [5, 5, 5, ···` (lista infinita); `[15,13..0] = [15,13,11,9,7,5,3,1]`; `[15,13..15] = [15]`; `[15,13..18] = []`.

De las explicaciones anteriores se deduce que las sucesiones de enteros satisfacen las leyes ecuacionales enumeradas a continuación, que son útiles para efectuar razonamientos basados en cálculo simbólico con sucesiones:

```
[n..]     =  n : [n+1..]
[n,m..]   =  n : [m,m+(m-n)..]
[n..l]    =  []                    si n > l
[n..l]    =  n : [n+1..l]          si n <= l
[n,m..l]  =  []                    si n <= m, n > l
[n,m..l]  =  n : [m,n+(m-n)..l]    si n <= m, n <= l
[n,m..l]  =  []                    si n > m, n < l
[n,m..l]  =  n : [m,n+(m-n)..l]    si n > m, n >= l
```

Sucesiones de valores de un tipo enumerado

Es posible también construir sucesiones de valores de cualquier tipo de la clase `Enum`. Recordemos la declaración predefinida de dicha clase:

```
class Enum a where
  succ, pred           :: a -> a
  toEnum               :: Int -> a
  fromEnum             :: a -> Int
  enumFrom             :: a -> [a]              -- [n..]
  enumFromThen         :: a -> a -> [a]         -- [n,m..]
  enumFromTo           :: a -> a -> [a]         -- [n..l]
  enumFromThenTo       :: a -> a -> a -> [a]    -- [n,m..l]

-- Conjunto suficiente de metodos: toEnum, fromEnum

  succ                  = toEnum . (1+)       . fromEnum
  pred                  = toEnum . subtract 1 . fromEnum
  enumFrom n            = map toEnum [fromEnum n ..]
  enumFromThen n m      = map toEnum [fromEnum n, fromEnum m, ..]
  enumFromTo n l        = map toEnum [fromEnum n .. fromEnum l]
  enumFromThenTo n m l  = map toEnum
                              [fromEnum n, fromEnum m .. fromEnum l]
```

Ahora estamos en condiciones de entender las definiciones genéricas de los cuatro último métodos de la clase, y vemos que su comportamiento co-

rresponde al cálculo de sucesiones. Haskell acepta las equivalencias siguientes, cualesquiera que sean los valores `n, m, l :: a` de cualquier tipo `a` de la clase `Enum`:

```
[n..]     =  enumFrom n
[n,m..]   =  enumFromThen n m
[n..l]    =  enumFromTo n l
[n,m..l]  =  enumFromThenTo n m l
```

En el caso de tipos de la clase `Enum` que tengan una cantidad finita de valores, todas las sucesiones resultan ser listas finitas y terminadas. Como ejemplo, mostramos el cálculo de algunas sucesiones de caracteres:

```
> ['0'..'9']
"0123456789"
> ['a','c'..'z']
"acegikmoqsuwy"
> ['\247'..]
"\247\248\249\250\251\252\253\254\255"
```

Listas intensionales

En el resto de esta sección vamos a estudiar una notación muy potente, que facilita mucho la programación con listas. Se trata de algo similar a la notación utilizada en matemáticas para definir conjuntos por *comprehensión* (esto es, especificando las propiedades que deben cumplir los elementos). Por ejemplo, la expresión matemática

$$\{(x,\ x\ div\ 2) \mid x \in \mathbb{N},\ x \text{ impar}\}$$

representa un conjunto infinito de parejas de números naturales, correspondientes a los números impares y a sus mitades aproximadas. La expresión anterior no sepuede utilizar en Haskell, donde no hay un tipo de datos para los conjuntos. Pero sí se puede usar la expresión siguiente, que representa una *lista infinita* de parejas de números, definida con el mismo criterio:

```
[(x, x `div` 2) | x <- [0..], odd x]
```

El cálculo de la expresión anterior produce la lista formada por todas las parejas deseadas y las va mostrando en pantalla. El cómputo no termina nunca, porque la lista es infinita. En cambio, el cálculo siguiente sí termina:

```
> take 9 [(x, x `div` 2) | x <- [0..], odd x]
[(1,0),(3,1),(5,2),(7,3),(9,4),(11,5),(13,6),(15,7),(17,8)]
```

En general, Haskell permite utilizar *listas intensionales*, que son expresiones de la forma `[e | c`$_1$`, ···, c`$_k$`]` construidas de acuerdo con los criterios siguientes:

- `e` es una expresión, llamada la *expresión principal.*
- Cada c_i puede ser un *generador* o una *restricción.* Obligatoriamente, c_1 debe ser un generador.
- Un *generador* es de la forma `p <- l`, donde `l` es una expresión de tipo lista y `p` es un patrón lineal del tipo de los elementos de `l`. Se dice que las variables de `p` son vinculadas por el generador `p <- l`, el cual opera considerando todos los valores posibles para las variables de `p` obtenidos de los diferentes elementos de `l` que se ajusten a `p` (ignorando los que no ajusten). La expresión `l` debe usar variables vinculadas por generadores anteriores, o definidas en el contexto en el que se esté evaluando la lista intensional.
- Una *restricción* es una expresión booleana. Debe usar variables vinculadas por generadores anteriores, o definidas en el contexto en el que se esté evaluando la lista intensional.
- La expresión principal debe usar variables vinculadas por los generadores, o definidas en el contexto en el que se esté evaluando la lista intensional.

Para calcular el valor de una lista intensional, Haskell considera todos los maneras posibles de asignar valores a las variables de la expresión principal, del modo especificado por los generadores, y de manera que se que cumplan las restricciones. En cada caso, la expresión principal toma un cierto valor. La lista de todos estos valores es el valor de la lista intensional.

Ejemplos de evaluación de listas intensionales

Examinemos algunos ejemplos de evaluación de listas intensionales, que se pueden experimentar usando el intérprete Hugs 98. En la mayoría de los ejemplos usaremos generadores sencillos de la forma `x <- l`, cuyo patrón se reduce a una variable `x`. Comenzamos con el cálculo de los cuadrados de los números pares de un intervalo, que combina un generador con una restricción:

```
> [x*x | x <- [1..10], even x]
[4,16,36,64,100]
```

A continuación calculamos el producto cartesiano de dos intervalos. Aquí podemos observar el orden en el que se calculan los elementos de la lista. Para cada valor fijo de `x` producido por el primer generador, `y` recorre todos los valores que puede producir el segundo generador.

```
> [(x,y) | x <- [1..3], y <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```

En el siguiente ejemplo, el segundo generador depende de la variable introducida por el primero. La lista resultante reúne aquellas parejas del producto cartesiano de un intervalo por sí mismo, cuya ordenada es menor o igual que su abcisa:

```
> [(x,y) | x <- [1..3], y <- [1..x]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

En el siguiente ejemplo, un poco más complicado, se calculan todas las ternas de números enteros acotados por 20 que pueden corresponder a las longitudes de los tres lados de un triángulo rectángulo. Las ternas con esta propiedad se llaman *Pitagóricas*.

```
> [(x,y,z) | z <- [2..20], y <- [1..(z-1)], x <- [1..y],
             x*x+y*y == z*z]
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]
```

En ocasiones, tiene sentido que la expresión principal ignore todas o algunas de las variables introducidas por los generadores. Esto se ilustra en el caso siguiente:

```
> ['a' | i <- [1..5]]
"aaaaa"
```

Hasta aquí, todos los ejemplos han usado generadores `p <- l` cuyo patrón `p` se reduce a una variable. La utilidad de generadores con patrones más complejos se ilustra en los dos cómputos sigientes:

```
> [x*x | Just x <- [Nothing, Just 3, Nothing, Just 5]]
[9,25]
> [x+y | ((x,y),True) <- [((1,2),True), ((2,3),False), ((3,4),True)]]
[3,7]
```

En general, un generador de la forma `p <- l` produce valores para las variables de `p` considerando todos los elementos de `l` que se ajusten a `p`, e ignorando los que no se ajusten a `p`.

Programación con listas intensionales

Como hemos visto en los ejemplos anteriores, las listas intensionales se pueden utilizar en las expresiones propuestas al intérprete para su evaluación. Sin embargo, su mayor utilidad consiste en emplearlas dentro de las definiciones de las funciones de un programa. A continuación vamos a ver algunos ejemplos típicos de programación con listas intensionales, definiendo varias funciones que no están predefinidas en Haskell.

Ejemplo: Producto escalar

Comenzamos programando una función que calcula el *producto escalar de dos vectores*. Para los propósitos de este ejemplo, es adecuado representar un vector como una lista. Podemos definir un tipo sinónimo para representar los vectores, y la función que calcula productos escalares, como sigue:

```
type Vector a = [a]
prodEsc     :: Num a => Vector a -> Vector a -> Vector a
prodEsc u v  = sum [x*y | (x,y) <- zip u v]
```

La definición de `prodEsc` espera que el tipo de los elementos de los vectores sea un tipo numérico. Además de las listas intensionales, la definición utiliza las funciones `sum` y `zip`. Como vimos en la sección 3.2, también es posible definir la función `prodEsc` mediante un diseño recursivo directo, o de otras maneras. La ventaja de la definición elegida aquí es su sencillez y claridad. En general, al diseñar programas funcionales interesa en muchos casos reutilizar funciones conocidas, en lugar de plantear nuevas definiciones recursivas.

Ejemplo: Producto matricial

A continuación, vamos a definir una función que calcula el *producto de dos matrices*. Elegimos representar una matriz como la lista formada por sus filas, que a su vez serán vectores. La definición del tipo sinónimo `Matriz` se debe entender en este sentido. Para calcular el producto de dos matrices `m` y `n`, tenemos en cuenta que cada fila de la matriz producto está formada por los productos escalares de una fila fija de `m` con todas las columnas de `n`. Con ayuda de una lista intensional, podemos expresar directamente esta idea:

```
type Matriz a = [Vector a]
prodMat     :: Num a => Matriz a -> Matriz a -> Matriz a
prodMat m n  = [[prodEsc fila columna | columna <- columnas n]
                | fila <- filas m]
```

La definición anterior presupone que el número de columnas de `m` es igual al número de filas de `n`, ya que en otro caso el producto escalar de filas de `m` por columnas de `n` no tendría sentido. Obsérvese también que en este caso la expresión principal de la lista intensional es otra lista intensional. En general, Haskell permite utilizar una lista intensional en cualquier lugar de un programa donde pueda aparecer una expresión de tipo lista.

Nos falta diseñar las funciones `filas` y `columnas`, que calculan respectivamente la lista de filas y la lista de columnas de una matriz. La primera de las dos es trivial. Como hemos elegido representar una matriz como la lista de sus filas, la función `filas` se reduce a la identidad:

```
filas  :: Matriz a -> [Vector a]
filas   = id
```

Para definir la función `columnas`, pensemos en una matriz `m` que tenga `q` columnas, numeradas con índices `j` comprendidos entre `0` y `q-1`. Para cada `j` fijo, los elementos que forman la columna de índice `j` son los que ocupan la posición `j` en las diferentes filas de `m`. Usando una lista intensional es fácil expresar todo ésto:

```
columnas   :: Matriz a -> [Vector a]
columnas n  = [[fila !! j | fila <- filas n] | j <- [0..(q-1)]]
              where q = numCol n
```

Ya solo nos falta definir la función que calcula el número `q` de columnas de una matriz `n`. Claramente, `q` es la longitud de cualquiera de las filas de `n`. Puesto que hemos elegido representar una matriz como la lista de sus filas, podemos definir:

```
numCol  :: Matriz a -> Int
numCol   = length . head
```

La ecuación de la definición anterior equivale a poner `numCol n = length (head n)`.

Ejemplo: Listas de asociación

Las listas de parejas, con tipo `[(a,b)]`, se utilizan muchas veces para representar un almacén de datos de tipo `a`, cada uno de los cuales tiene una información asociada de tipo `b`. En estos casos, se habla de una *lista de asociación*. Por ejemplo, la siguiente lista de asociación asocia a una serie de palabras castellanas sus traducciones al inglés:

```
[("animal", "animal"), ("carne", "meat"), ("dia", "day"),
 ("leche", "milk"), ("manzana", "apple"), ("mujer", "woman"),
 ("roto", "broken"), ("socorro", "help"), ("tomate", "tomato")]
```

En cada pareja `(x,y)` de una lista de asociación, el dato `x` se suele llamar *clave*, mientras que `y` es la *información asociada* a la clave `x`. La siguiente función sirve para consultar una lista de asociación y calcular la información asociada a una clave dada:

```
assoc       :: Eq a => a -> [(a,b)] -> b
assoc x xys  = head [y | (x', y) <- xys, x == x']
```

La ejecución de `(assoc x xys)` causa un error en el caso de que la lista de asociación `xys` no incluya *ninguna* pareja con clave `x`. En el caso de que `xys` incluya *varias* parejas con clave `x`, `assoc` se ha definido de tal manera

que tiene en cuenta la pareja más próxima a la cabeza de la lista.

Dos errores comunes

Las listas intensionales pueden dar lugar a comportamientos inesperados si el programador las utiliza incorrectamente. Una posible causa de mal comportamiento es el uso de *generadores infinitos*. Se dice que un generador `p <- l` es infinito cuando la evaluación de la expresión `l` da lugar a una lista infinita. Debe tenerse en cuenta:

Primera cautela: La evaluación de una lista intensional que contenga generadores infinitos no termina.

Para comprender ésto, podemos probar el siguiente cálculo en Hugs 98:

```
> [3*x | x <- [0..], 2*x < 5]
[0,3,6
```

Podría esperarse equivocadamente que el resultado del cálculo fuese `[0,3,6]`. Sin embargo, el cómputo no termina, salvo que el usuario fuerce su interrupción pulsando una tecla de control. La razón es que el generador `x <- [0..]` produce infinitas posibilidades para `x` y el intérprete intenta probarlas todas, sin darse cuenta de que solo hay una cantidad finita de ellas que cumplan la restricción `2*x < 5`.

Otra causa frecuente de errores resulta de ignorar la

Segunda cautela: Un generador `p <- l` vincula las variables del patrón `p` con prioridad sobre cualquier otra vinculación de variables con los mismos identificadores, que pueda existir en un contexto más externo.

Para entender bien lo que se quiere decir con ésto, consideremos una nueva definición para la función `assoc` que hemos estudiado más arriba:

```
wrongAssoc       :: Eq a => a -> [(a,b)] -> b
wrongAssoc x xys  = head [y | (x,y) <- xys]
```

A primera vista, esta definición parece correcta y más sencilla que la presentada anteriormente. Sin embargo, teniendo en cuenta la segunda cautela vemos que el generador `(x,y) <- xys` vincula `x`, de tal modo que *el valor pasado como parámetro se ignorará*. Así, `[y | (x,y) <- xys]` resultará ser la lista de las segundas componentes de *todas* las parejas de `xys`, y `(wrongAssoc x xys)` siempre calculará la información asociada a *la primera pareja* de la lista `xys`. Por ejemplo, se tendrá:

```
> wrongAssoc "gato" [("perro","dog"), ("gato","cat")]
"dog"
```

Reglas de eliminación de listas intensionales

Las explicaciones informales que hemos dado hasta ahora son suficientes en principio para comprender cómo se calcula el valor de una lista intensional. No obstante, el conocer una explicación más precisa puede ayudar a evitar errores como los que acabamos de comentar.

Las implementaciones de Haskell no evalúan directamente las listas intensionales, sino que las preprocesan previamente, traduciéndolas a expresiones en las cuales ya no se utiliza la sintaxis de lista intensional. Este proceso de *eliminación de listas intensionales* se puede realizar aplicando reiteradamente las reglas de transformación enunciadas a continuación. Al leerlas, recuérdese que cualquier lista intensional debe incluir por lo menos una condición, y que la primera condición (de izquierda a derecha) debe ser un generador.

(M_{var}) Esta regla se aplica para transformar una lista intensional con una sola condición `x <- l`, donde `x` es una variable.

```
[e | x <- l] = map f l
               where f x = e
```

Una forma equivalente de expresar esta regla usa una función anónima en lugar de una definición local:

```
[e | x <- l] = map (\x -> e) l
```

Hay además dos casos particulares interesantes en los cuales la transformación se puede simplificar:

```
[x | x <- l]   = l

[f x | x <- l] = map f l
```

(M_{pat}) Esta regla se aplica para transformar una lista intensional con una sola condición `p <- l`, donde el patrón lineal `p` no es una variable.

```
[e | p <- l] = maybeMap f l
               where f p = Just e
                     f _ = Nothing
```

Obsérvese que la definición de la función auxiliar `f`, y el uso de `maybeMap` en lugar de `map`, se encargan de que los elementos de la lista `l` que no se ajusten al patrón `p` sean ignorados.

(F_{var}) Esta regla se aplica para transformar una lista intensional cuyas dos primeras condiciones son `x <- l, b`, donde `x` es una variable y `b` es una restricción booleana. Si hay más de dos condiciones, todas ellas de la tercera en adelante se conservan en la lista intensional transformada.

```
[e | x <- l, b, ...] = [e | x <- filter t l, ...]
                       where t x = b
```

Esta transformación admite una forma equivalente que usa una función anónima:

```
[e | x <- l, b, ...] = [e | x <- filter (\x -> b) l, ...]
```

Además, la transformación se puede simplificar cuando cuando `b` ya es de la forma `t x`:

```
[e | x <- l, t x, ...] = [e | x <- filter t l, ...]
```

(F_{pat}) Esta regla se aplica para transformar una lista intensional cuyas dos primeras condiciones son `p <- l, b`, donde el patrón lineal `p` no es una variable y `b` es una restricción booleana. Las condiciones restantes (si las hay) se conservan.

```
[e | p <- l, b] = [e | p <- filter t l]
                  where t p = b
                        t _ = False
```

En este caso, la definición de la función booleana auxiliar `t` se encarga de que los elementos de la lista `l` que no e ajusten al patrón `p` sean ignorados.

(C) Estas regla se aplica para transformar una lista intensional cuyas dos primeras condiciones son dos generadores. Las condiciones restantes (si las hay) se conservan.

```
[e | p1 <- l1, p2 <- l2, ...]
= concat [[e |  p2 <- l2, ...] | p1 <- l1]
```

La idea de esta transformación es que, para cada vínculo de las variables de `p1` producido por el primer generador, se calculará el valor de la lista intensional `[e | p2 <- l2, ...]`; a continuación, todas estas listas se agruparán en una sola gracias a `concat`.

Las funciones `map`, `filter` y `concat` juegan un papel importante en la eliminación de listas intensionales. Como curiosidad, nótese que estas tres funciones se pueden definir muy fácilmente usando listas intensionales:

```
map      :: (a -> b) -> [a] -> [b]
map f xs  = [f x | x <- xs]

filter      :: (a -> Bool) -> [a] -> [a]
filter p xs  = [x | x <- xs, p x]

concat     :: [[a]] -> [a]
concat xss  = [x | xs <- xss, x <- xs]
```

En la práctica, las implementaciones de Haskell eliminan las listas intensionales aplicando versiones optimizadas de las reglas de transformación que hemos explicado aquí (o de otras equivalentes).

Uso de las reglas de eliminación de listas intensionales

A continuación, presentamos tres ejemplos de uso de las reglas de eliminación de listas intensionales. En cada caso, se tratará de eliminar listas intensionales utilizadas dentro de la definición de una cierta función.

Ejemplo 1: La siguiente función sirve para calcular los cuadrados de los números pares del intervalo `[0..n]`:

```
cuadrados   :: Int -> [Int]
cuadrados n  = [x*x | x <- [0..n], even x]
```

Aplicando sucesivamente las transformaciones (F_{var}) y (M_{var}) se elimina el uso de listas intensionales, y resulta la siguiente definición equivalente:

```
cuadrados   :: Int -> [Int]
cuadrados n  = map cuadrado (filter even [0..n])
               where cuadrado x = x*x
```

Ejemplo 2: Consideramos ahora una función que calcula el producto cartesiano de los intervalos `[0..n]` y `[0..m]`:

```
producto     :: Int -> Int -> [(Int,Int)]
producto n m  = [(x,y) | x <- [0..n], y <- [0..m]]
```

En este caso, aplicando primero la transformación (C) y a continuación la transformación (M_{var}) dos veces seguidas, resulta la definición equivalente:

```
producto     :: Int -> Int -> [(Int,Int)]
producto n m  = concat (map (\x -> map (\y -> (x,y)) [0..m]) [0..n])
```

Se deja como *ejercicio* para el lector el detallar los pasos de esta transformación.

Ejemplo 3: Finalmente, consideramos una función que recibe como parámetro una lista `l` de tipo `Either a b` y separa en dos listas diferentes los elementos de `l`, según que sean copias de elementos de tipo `a` o copias de elementos de tipo `b`:

```
separa   :: [Either a b] -> ([a], [b])
separa l  = ([x | Left x <- l], [y | Right y <- l])
```

En este caso, para eliminar la lista intensional hay que utilizar dos veces la transformación (M_{pat}). El resultado es:

```
separa   :: [Either a b] -> ([a], [b])
separa l  = (maybeMap f l, maybeMap g l)
            where f (Left x)   = Just x
                  f    _       = Nothing
                  g (Right y)  = Just y
                  g    _       = Nothing
```

A partir de los ejemplos, podemos concluir que el uso de listas intensionales permite en muchos casos escribir definiciones más sencillas y legibles. Como inconveniente, la eficiencia de tales definiciones no siempre es óptima. Por ejemplo, la función `separa` se ejecuta realizando dos recorridos de la lista dada como parámetro. Sería posible una definición recursiva equivalente y más eficiente, que realizase un solo recorrido.

Aplicación: Algoritmos de ordenación de listas

La *ordenación* de una lista es útil para muchas aplicaciones en las que se requiere ordenar una serie de datos residentes en la memoria principal. Como aplicación de las técnicas de programación funcional estudiadas hasta este momento, vamos a abordar el diseño funcional de tres de los algoritmos de ordenación más conocidos: la ordenación mediante *inserción*, la ordenación mediante *mezcla* y la ordenación *rápida* debida a Hoare. Todas las funciones de ordenación presentadas en este apartado se pueden utilizar para ordenar listas de elementos de cualquier tipo de la clase `Ord`.

Ordenación mediante el algoritmo de inserción

Este algoritmo se basa en una idea muy sencilla: suponiendo que una parte de la lista ya haya sido ordenada, se *inserta* un nuevo elemento, de manera que quede situado en el lugar que le corresponde. El proceso se repite hasta consumir todos los elementos de la lista original. En ese momento, se

tiene la lista totalmente ordenada. Se sabe que el tiempo de ejecución de este algoritmo es $\mathcal{O}(n^2)$, siendo n la longitud de la lista que se está ordenando.

En los lenguajes imperativos, la ordenación por inserción se suele programar mediante un bucle. La formulación del algoritmo en estilo funcional se obtiene fácilmente siguiendo el esquema de recursión natural:

```
ordIns      :: Ord a => [a] -> [a]
ordIns []     = []
ordIns (x:xs) = insOrd x (ordIns xs)

insOrd          :: Ord a => a -> [a] -> [a]
insOrd x    []   = [x]
insOrd x (y:ys)  = if x <= y
                      then x:y:ys
                      else y : insOrd x ys
```

La función `ordIns` representa el algoritmo de ordenación, mientras que `insOrd` es la función auxiliar encargada de insertar un nuevo elemento en una lista ya ordenada, de manera que la lista resultante vuelva a estar ordenada.

Ordenación mediante el algoritmo de mezcla

El algoritmo de ordenación por mezcla obedece a un esquema recursivo típico de la estrategia conocida como *divide y vencerás*:

- Una lista de longitud menor que dos ya está ordenada.
- Para ordenar una lista de longitud mayor o igual que dos, la lista se divide en dos mitades que se ordenan recursivamente, cada una por separado. Una vez ordenadas las dos mitades, se *mezclan* para obtener la ordenación de la lista total.

La formulación funcional de este algoritmo se expresa mediante recursión doble:

```
ordMezcla                :: Ord a => [a] -> [a]
ordMezcla    []           = []
ordMezcla    [x]          = [x]
ordMezcla    xs@(x:y:zs)  =  mezclaOrd (ordMezcla us) (ordMezcla vs)
                             where n       =  length xs `div` 2
                                   (us,vs) =  splitAt n xs

mezclaOrd              :: Ord a => [a] -> [a] -> [a]
mezclaOrd []     ys     = ys
mezclaOrd (x:xs) []     = x:xs
mezclaOrd (x:xs) (y:ys) = if x <= y
                             then x : mezclaOrd xs (y:ys)
                             else y : mezclaOrd (x:xs) ys
```

La función `ordMezcla`, encargada de la ordenación, usa los patrones de los lados izquierdos de sus tres ecuaciones para distinguir los caso directos (lista vacía o unitaria) del caso recursivo (lista con dos o más elementos). En el caso recursivo, la notación `xs@(x:y:zs)` usada en el lado izquierdo de la ecuación indica una lista llamada `xs` que se debe ajustar al patrón de la forma `(x:y:zs)`. En el lado lado derecho de la misma ecuación, la función `splitAt` estudiada en la sección 3.2 se usa para dividir `xs` en dos mitades con aproximadamente la misma longitud. Los resultados de las llamadas recursivas que ordenan las dos mitades se mezclan con ayuda de la función auxiliar `mezclaOrd`, que espera recibir como parámetros dos listas ordenadas y devuelve como resultado una única lista ordenada que reúne los elementos de las otras dos.

La función `ordMezcla` es aplicable a los mismos casos que la función `ordIns`, aunque más eficiente. Requiere tiempo de cómputo $\mathcal{O}(n\,log(n))$. Para que la diferencia de tiempo sea apreciable, es necesario probar ejecuciones de ambas funciones sobre una misma lista de tamaño suficientemente grande. Se puede usar como ejemplo una lista de enteros generada con ayuda de algún algoritmo pseudoaleatorio, como se hace en las pruebas sugeridas en el apéndice A.14.

Ordenación rápida mediante el algoritmo de Hoare

Uno de los mejores algoritmos de ordenación que se conocen es la ordenación *rápida* de Hoare, cuyo tiempo de ejecución (en promedio) es también $\mathcal{O}(n\,log(n))$. También en este caso el algoritmo sigue un esquema recursivo de tipo *divide y vencerás*:

- La lista vacía ya está ordenada.
- Una lista no vacía con cabeza `x` se ordena separando los elementos de su resto en dos sublistas: los *menores o iguales* que `x`, y los *mayores* que `x`. Estas dos sublistas se ordenan recursivamente. Por último, la ordenación de la lista total se obtiene concatenando ls dos sublistas ordenadas resultantes de las llamadas recursivas, con el elemento `x` intercalado entre las dos.

En Haskell, la descripción de este algoritmo queda muy breve y clara utilizando listas intensionales:

```
ordRapido             :: Ord a => [a] -> [a]
ordRapido    []        = []
ordRapido    (x:xs)    = ordRapido [ u | u <- xs, u <= x ]
                                     ++ [x] ++
                         ordRapido [ v | v <- xs, v > x ]
```

Algoritmos de ordenación con el orden dado como parámetro

Las tres funciones de ordenación expuestas más arriba son aplicables a listas de datos de cualquier tipo `a` que esté en la clase `Ord`, y siempre ordenan la lista con respecto al orden `(<=)` que se haya establecido al incluir al tipo `a` en dicha clase. Este planteamiento resulta demasiado rígido para aplicaciones en las que interese considerar órdenes alternativos para un mismo tipo de datos. Pensemos por ejemplo en el tipo sinónimo `Persona` definido en la sección 2.6. Nos puede interesar comparar personas con respecto al nombre, al DNI o a la edad, según los casos. A la hora de ordenar listas de personas, sería conveniente que pudiésemos elegir con respecto a qué orden entre personas nos interesa hacerlo (alguno de los tres anteriores, o algún otro).

En un lenguaje de orden superior como Haskell, es muy fácil resolver este problema. Basta considerar el orden entre elementos que deseemos utilizar como un *parámetro* de la función de ordenación. Observemos que un orden entre elementos de tipo `a` será una función con tipo `(a -> a -> Bool)`. Por lo tanto, el tipo de una función de ordenación que reciba al orden como parámetro será `(a -> a -> Bool) -> [a] -> [a]`. Ahora ya no hace falta la condición de que `a` pertenezca a la clase `Ord`, porque no vamos a usar el orden `(<=)` sino el que nos dan como parámetro.

A continuación presentamos las funciones que realizan la ordenación por inserción, por mezcla y por el método rápido, modificadas en los tres casos para que el orden de los elementos venga dado como un parámetro, nombrado por el identificador `mig`.

Ordenación por inserción con el orden dado como parámetro

```
ordInsCon           :: (a -> a -> Bool) -> [a] -> [a]
ordInsCon mig []     = []
ordInsCon mig (x:xs) = insOrdCon mig x (ordInsCon mig xs)

insOrdCon              :: (a -> a -> Bool) -> a -> [a] -> [a]
insOrdCon mig x    []   = [x]
insOrdCon mig x (y:ys)  = if mig x y
                             then x:y:ys
                             else y : insOrdCon mig x ys

--   Observacion: ordIns = ordInsCon (<=)
```

Ordenación por mezcla con el orden dado como parámetro

```
ordMezclaCon                    :: (a -> a -> Bool) -> [a] -> [a]
ordMezclaCon mig    []           = []
ordMezclaCon mig    [x]          = [x]
ordMezclaCon mig    xs@(x:y:zs)  =  mezclaOrdCon mig (ordMezclaCon mig us)
```

```
                                                  (ordMezclaCon mig vs)
                                      where n       =  length xs `div` 2
                                            (us,vs)  =  splitAt n xs

mezclaOrdCon                   :: (a -> a -> Bool) -> [a] -> [a] -> [a]
mezclaOrdCon mig []     ys      = ys
mezclaOrdCon mig (x:xs) []      = x:xs
mezclaOrdCon mig (x:xs) (y:ys) = if mig x y
                                    then x : mezclaOrdCon mig xs (y:ys)
                                    else y : mezclaOrdCon mig (x:xs) ys

--   Observacion: ordMezcla = ordMezclaCon (<=)
```

Ordenación rápida de Hoare con el orden dado como parámetro

```
ordRapidoCon                  :: (a -> a -> Bool) -> [a] -> [a]
ordRapidoCon mig    []         = []
ordRapidoCon mig    (x:xs)     = ordRapidoCon mig [ u | u <- xs, mig u x ]
                                                  ++ [x] ++
                                 ordRapidoCon mig [ v | v <- xs, not (mig v x) ]

--   Observacion: ordRapido = ordRapidoCon (<=)
```

Las funciones de ordenación del apartado anterior (basadas en el orden por defecto (`<=`)) se comportan como casos particulares de las funciones de ordenación del presente apartado (que reciben el orden como parámetro). Esto se puede observar en los comentarios incluidos en las líneas finales de los tres últimos algoritmos. Por ejemplo, la ecuación

```
ordRapido = ordRapidoCon (<=)
```

se puede utilizar como definición ejecutable de `ordRapido` a partir de `ordRapidoCon`.

En este punto, el lector debería utilizar el módulo Hugs 98 del apéndice A.14 para probar el funcionamiento práctico de todas las funciones de ordenación que acabamos de estudiar.

Aplicación: Factura de un supermercado

Finalizamos esta sección presentando otra aplicación sencilla de técnicas de programación con listas, inspirada por un ejemplo presentado en el texto de Simon Thompson [10]. Consideramos el siguiente problema: la lectora óptica de la caja de un supermercado produce una lista de códigos de barras, tal como `[1234,4719,3814,1112,1113,1234]`, donde cada código de barras se representa como un número entero con cuatro dígitos. A partir de la lista de códigos de barras, se desea imprimir una factura con un formato legible. Por ejemplo, a partir de la lista anterior se querría obtener una factura con el siguiente aspecto:

```
Tio Pepe, 1lt......................5.40
Fritos de maiz.....................1.21
Cacahuetes.........................0.56
Chupa Chups (bolsa gigante).........1.33
Item desconocido...................0.00
Tio Pepe, 1lt......................5.40

Total..............................13.90
```

La factura indica los precios *en euros*, con dos decimales. Para poder calcularla, se supone disponible una *base de datos* en la que se puedan consultar el *nombre* y el *precio* del producto asociado a cada código. Supondremos que la base de datos se representa como lista de ternas de la forma `(c,n,p)`, donde `c` es un código de barras, `n` es el nombre del correspondiente producto, y `p` es su precio *en céntimos de euro*. Supondremos también que esta lista está ordenada en orden creciente con respecto a los códigos. Cuando una lista de códigos contenga algún código erróneo que no apaerzca en la base de datos, la factura deberá hacerlo constar como un item desconocido de precio nulo.

El módulo Hugs 98 del apéndice A.15 se ha diseñado para resolver este problema. Se definen varios tipos sinónimos útiles, incluyendo `Compra` como sinónimo del tipo de las listas de códigos de barras, y `Factura` como sinónimo del tipo de las listas formadas por parejas de tipo `(Nombre, Precio)`. La función principal del módulo es `printFactura`, que convierte una compra dada como parámetro en el proceso de entrada/salida que imprime en el terminal la correspondiente factura. La ecuación:

```
printFactura  = putStrLn . showFactura . hazFactura
```

muestra como el problema se reduce a otros tres: calcular una factura a partir de una compra (función `hazFactura`), convertir la factura a una representación bien formateada como cadena de caracteres con saltos de línea (función `showFactura`), y finalmente imprimir la cadena anterior (función predefinida `putStrLn`). En general, en los programas funcionales la descomposición de un problema en varios problemas que se han de resolver consecutivamente, se suele expresar mediante la composición de varias funciones. En un lenguaje imperativo, la composición secuencial de órdenes aparecería en este lugar.

La programación de `hazFactura` se reduce a sucesivas consultas a la base de datos, realizadas por medio de la función `consulta`. La sucesión de consultas tiene lugar a lo largo de las llamadas recursivas de `hazFactura` a sí misma. Por otro lado, la definición de `showFactura` está estructurada con ayuda de otras funciones auxiliares, que utilizan básicamente la operación de concatenación de cadenas de caracteres y la conversión de números a cadenas de caracteres. El estudio de los detalles y la ejecución de pruebas

se dejan como ejercicio.

Ejercicios

1. Demuestra las dos propiedades siguientes:

   ```
   map (k+) [m..n]      =  [k+m..k+n]   si m ≤ n+1
   [m..p] ++ [p+1..n]   =  [m..n]       si m ≤ p+1 ≤ n+1
   ```

2. Prueba el siguiente cálculo en el intérprete Hugs 98. Explica de palabra qué significa la lista de números obtenida como resultado.

   ```
   > [i | ('a', i) <- zip xs [0..]]
     where xs = "La vida es maravillosa"
   [1,6,12,14,21]
   ```

3. Usando listas intensionales, programa una función

   ```
   posicion ::  Eq a => [a] -> a -> Int
   ```

   tal que `posicion xs x` devuelva el menor entero `i` tal que `xs !! i = x`, si existe, o -1 en caso contrario.

4. Programa la función `(!!)` usando listas intensionales.

5. Traduce las listas intensionales siguientes a expresiones equivalentes que no usen la notación de lista intensional.

   (a) `[(x,y) | x <- [1..5], y <- [1..x]]`.

   (b) `[(x,y) | x <- [1..5], odd x, y <- [1..x]]`.

6. Sin usar listas intensionales, construye una definición recursiva de la función `separa` estudiada en el último ejemplo de esta sección. Hazlo de manera que la evaluación de `(separa l)` se ejecute con un único recorrido de la lista `l` dada como parámetro.

7. Elimina las listas intensionales de la definición de la función `assoc` que hemos definido para hacer consultas en listas de asociación. Considerando la definición resultante como una especificación de `assoc`, usa la técnica de plegado-desplegado para deducir otra definición recursiva más eficiente en la que no intervengan funciones de orden superior.

8. La función `assoc` produce un error en el caso de que la lista no contenga ninguna pareja cuya primera componente sea la clave buscada. Define otra función de consulta

   ```
   lookup :: Eq a => a -> [(a,b)] -> Maybe b
   ```

que devuelva el resultado `Just y` cuando la lista `xys` contenga una pareja `(x,y)`, y el resultado `Nothing` en caso contrario. Esta nueva función siempre termina su cómputo con éxito. Recuerda que `Maybe` es el tipo construido predefinido que hemos estudiado en la sección 2.5.

9. Define variantes más optimizadas de las funciones `assoc` y `lookup`, bajo el supuesto de que el tipo de las claves sea de la clase `Ord` y la lista de asociación dada esté ordenada en orden creciente con respecto a las claves.
   Pista: Para una lista de asociación de longitud n, los tiempos de ejecución de `assoc` y `lookup` en el caso peor son $\mathcal{O}(n)$. Las versiones optimizadas que se piden no podrán mejorar esta cota en el caso peor, pero sí en ciertos casos. Explica en qué casos.

10. Utilizando una lista intensional, define una función

    ```
    resultados :: [Maybe a] -> [a]
    ```

    de tal manera que `(resultados xs)` calcule la lista de todos los `x :: a` tales que `(Just x)` aparece en `xs`. Los elementos `Nothing` que puedan aparecer en `xs` deben ser ignorados. Escribe una definición alternativa de la misma función sin usar listas intensionales.

11. Programa definiciones de las dos funciones siguientes utilizando listas intensionales. A continuación, deduce definiciones equivalentes que no usen listas intensionales.

    (a) `inits :: [a] -> [[a]]` tal que `inits xs` devuelva la lista formada por todos los segmentos iniciales de `xs`, en orden de longitud creciente. Por ejemplo, `inits [0,1,2]` debe devolver el resultado `[[],[0],[0,1],[0,1,2]]`.

    (b) `perms :: [a] -> [[a]]` tal que `perms xs` devuelva una lista formada por todas las permutaciones posibles de la lista `xs`, en un cierto orden y sin repeticiones.

12. Define una función de ordenación que ordene una lista dada mediante el algoritmo de *selección reiterada del elemento mínimo*, que opera como sigue: si la lista es vacía, ya está ordenada; en otro caso, se separa el elemento mínimo, se ordena recursivamente la lista formada por los demás elementos, y se añade en cabeza el elemento mínimo para obtener así la lista totalmente ordenada.

    Indicación: Programa la función pedida de dos maneras, según que el orden entre los elementos de la lista venga o no dado como parámetro.

13. Modifica el módulo `Supermercado` de manera que la factura se imprima ordenada alfabéticamente con respecto a los nombres de los productos.

14. Añade otra modificación a `Supermercado` de manera que se realice un descuento de un euro por cada dos botellas de *Tio Pepe* adquiridas. Por ejemplo, para una compra representada por la lista de códigos

```
[1234,4719,3814,1112,1113,1234]
```

se deberá imprimir la siguiente factura:

```
Cacahuetes.........................0.56
Chupa Chups (bolsa gigante).....1.33
Fritos de maiz.....................1.21
Item desconocido...................0.00
Tio Pepe, 1lt......................5.40
Tio Pepe, 1lt......................5.40

Descuento..........................1.00

Total.............................13.90
```

Sugerencia: Define algunas funciones nuevas, elegidas de manera que sea fácil combinarlas con el programa antigüo para obtener el nuevo.

15. Rediseña `Supermercado` de manera que los códigos de barras desconocidos no produzcan ninguna información visible en la factura final. Hay al menos dos maneras de abordar este problema:

    - Mantener la función `hazFactura` tal cual está y modificar la función `showFactura`.
    - Modificar la función `hazFactura` de modo que produzca una factura en la que no figuren items desconocidos.

16. Supón que la base de datos de `Supermercado` esté ordenada en orden creciente con respecto a los códigos de barras. Bajo este supuesto, diseña una definición más eficiente de la función `consulta`, que interrumpa la exploración de la base de datos tan pronto como aparezca un código de barras estrictamente mayor que el que se está buscando.

17. Añade a `Supermercado` una nueva función que sirva para actualizar una base de datos, añadiendo una nuevo código de barras, junto con el nombre y el precio del producto correspondiente. Si el código de barras ya existía anteriormente en la base de datos, la antigüa información asociada a él debe desaparecer.
    Indicación: Para resolver este ejercicio, supón que la base de datos que va a ser actualizada está ordenada en orden creciente de códigos de barras. La nueva base de datos deberá quedar ordenada del mismo modo.

3.4 Operadores de acumulación

En la sección 3.1 hemos descrito brevemente el esquema de *recursión natural* y el esquema de *recursión final con acumulador*, a los que se ajustan muchas definiciones recursivas de funciones que operan con listas. Los operadores de acumulación que vamos a estudiar en esta sección son funciones de orden superior que sirven para expresar de forma genérica el comportamiento de estos esquemas.

Los operadores `fold`

Recordemos los esquemas de recursión presentados en la sección 3.1:

```
-- Recursion natural para listas.
-- Presupone e :: b
--           f :: a -> b -> b

    h        :: [a] -> b
    h []      = e
    h (x:xs)  = f x (h xs)

-- Recursion final con acumulador para listas.
-- Presupone e :: b
--           f :: b -> a -> b

    h           :: b -> [a] -> b
    h  e []      = e
    h  e (x:xs)  = h (f e x) xs
```

Considerando a `f` y `e` como parámetros resultan las definiciones de dos funciones de orden superior que expresan de manera genérica estos esquemas:

```
    foldr            :: (a -> b -> b) -> b -> [a] -> b
    foldr f e []      = e
    foldr f e (x:xs)  = f x (foldr f e xs)

    foldl            :: (b -> a -> b) -> b -> [a] -> b
    foldl f e  []     = e
    foldl f e  (x:xs) = foldl f (f e x) xs
```

Las dos funciones anteriores están predefinidas en Haskell, y se llaman operadores de acumulación. Su uso permite escribir definiciones de la forma

`h = foldr f e` o bien `h = foldl f e`

(con una elección concreta de los parámetros `f` y `e`), que corresponden al comportamiento de la recursión natural y la recursión final con acumulador, respectivamente. A veces se utiliza un operador infijo ($\oplus$) como parámetro `f`. En este caso, suponiendo una lista finita y terminada

`xs = x`$_0$`: ··· :x`$_{n-1}$`:[]`, la evaluación de las llamadas a los operadores `fold` se comporta como sigue:

$$\texttt{foldr } (\oplus)\ \texttt{e xs} \longrightarrow^* \texttt{x}_0 \oplus (\texttt{x}_1 \oplus (\ \cdots (\texttt{x}_{n-1} \oplus \texttt{e}) \cdots))$$
$$\texttt{foldl } (\oplus)\ \texttt{e xs} \longrightarrow^* (\ \cdots ((\texttt{e} \oplus \texttt{x}_0) \oplus \texttt{x}_1) \cdots) \oplus \texttt{x}_{n-1}$$

El uso de las letras "`r`" y "`l`" en los nombres de los dos operadores hace alusión al comportamiento anterior.

Ejemplos de uso de los operadores `fold`

Muchas definiciones de funciones recursivas se expresan fácilmente por medio de los operadores `fold`, dejando la recursión implícita en las definiciones de estos. Los ejemplos que siguen comprenden varias funciones estudiadas en las secciones anteriores de este capítulo. Consideramos en primer lugar ejemplos para `foldr`:

(R1)
```
sum :: Num a => [a] -> a
sum  = foldr (+) 0
```

(R2)
```
product :: Num a => [a] -> a
product  = foldr (*) 1
```

(R3)
```
concat :: [[a]] -> a
concat  = foldr (++) []
```

(R4)
```
and :: [Bool] -> Bool
and  = foldr (&&) True
```

(R5)
```
or :: [Bool] -> Bool
or  = foldr (||) False
```

(R6)
```
length :: [a] -> Int
length  = foldr unoMas 0
          where unoMas x n = 1+n
```

(R7)
```
inversa :: [a] -> [a]
inversa  = foldr ponDetras []
           where ponDetras ys x = ys ++ [x]
```

(R8)
```
unzip :: [(a,b)] -> ([a],[b])
unzip  = foldr pon2Delante []
         where pon2Delante (x, y) (xs, ys) = (x:xs, y:ys)
```

(R9)
```
map   :: (a -> b) -> [a] -> [b]
map f  = foldr ((:) . f) []
```

(R10)
```
takeWhile   :: (a -> Bool) -> [a] -> [b]
takeWhile p  = foldr ponConOjo []
               where ponConOjo x ys
                       | p x       = x:ys
                       | otherwise = []
```

Estudiando estos ejemplos, podemos observar que en algunos casos la evaluación de una llamada a `foldr` de la forma `(foldr f e xs)` puede terminar sin necesidad de completar la evaluación de la lista `xs`, incluso aunque `xs` sea infinita. Esto ocurre en los casos siguientes:

```
> and [even n | n <- [0..]]
False
> or [even n | n <- [0..]]
True
> takeWhile (<10) [0..]
[0,1,2,3,4,5,6,7,8,9]
```

A continuación presentamos una colección de ejemplos de uso de `foldl`:

(L1)
```
sum :: Num a => [a] -> a
sum  = foldl (+) 0
```

(L2)
```
product :: Num a => [a] -> a
product  = foldl (*) 1
```

(L3)
```
concat :: [[a]] -> a
concat  = foldl (++) []
```

(L4)
```
and :: [Bool] -> Bool
and  = foldl (&&) True
```

(L5)
```
or :: [Bool] -> Bool
or  = foldl (||) False
```

(L6)
```
length :: [a] -> Int
length  = foldl MasUno 0
          where MasUno n x = n+1
```

(L7)
```
reverse :: [a] -> [a]
reverse  = foldl ponDelante []
           where ponDelante ys x = x : ys
```

(L8)
```
unzip :: [(a,b)] -> ([a],[b])
unzip  = foldl pon2Detras []
         where pon2Detras (xs, ys) (x, y) = (xs ++ [x], ys ++ [y])
```

(L9)
```
valDec :: [Int] -> Int
valDec  = foldl acumulaDigito 0
          where acumulaDigito n d = 10*n+d
```

Observamos que los ejemplos (L1)–(L5) se corresponden exactamente con (R1)–(R5), pues se trata de funciones que se pueden definir del mismo modo con ambos operadores `fold`. Los ejemplos (L6)–(L8) también se corresponden con (R6)–(R8), aunque en estos casos las definiciones obtenidas con ambos operadores difieren en la elección del parámetro `f`. El ejemplo (L9) define una función que calcula como resultado el valor numérico del entero cuyos dígitos en notación decimal vienen indicados por la lista `ds :: [Int]` dada como parámetro.

La serie de ejemplos (L1)–(L9) también sirve para ilustrar el hecho de que una llamada a a `foldl` de la forma `(foldr f e xs)` solo puede terminar con éxito cuando la lista `xs` es finita y terminada. Esto ocurre necesariamente siempre que se opta por definiciones basadas en recursión final con acumulador. En particular, si `and` y `or` se definiesen mediante `foldl` (cosa que *no se hace* en el preludio de Haskell), nunca podrían aplicarse a listas infinitas.

Leyes de dualidad de los operadores `foldl`

Como hemos vistos en las series de ejemplos (R1)–(R8) y (L1)–(L8), son frecuentes los casos en que una misma función admite definiciones alternativas basadas en `foldr` y `foldl`, respectivamente. Los dos teoremas siguientes explican este fenómeno.

Primer Teorema de Dualidad. Supongamos que `(⊕)` sea una operación asociativa y que `e` sea elemento neutro de `(⊕)` por ambos lados, es decir, que sean válidas las ecuaciones:

```
(A)  x ⊕ (y ⊕ z)  =  (x ⊕ y ) ⊕ z
(E)  x ⊕ e        =  x
     e ⊕ x        =  x
```

Entonces, para cualquier lista finita y terminada `xs` del tipo adecuado, se verifica:

```
foldr (⊕) e xs = foldl (⊕) e xs
```

Segundo Teorema de Dualidad. Supongamos que `(⊕)`, `(⊗)` y `e` sean tales que las siguientes ecuaciones sean válidas:

```
(A)  x ⊕ (y ⊗ z)  =  (x ⊕ y ) ⊗ z
(E)  x ⊕ e        =  e ⊗ x
```

Entonces, para cualquier lista finita y terminada `xs` del tipo adecuado, se verifica:

```
foldr (⊕) e xs = foldl (⊗) e xs
```

Claramente, el primer teorema de dualidad es un caso particular del segundo, el cual se demuestra razonando por inducción sobre la lista `xs`. La demostración, que omitimos aquí, se encuentra en la sección 4.6 del texto [1].

Como ejemplos de aplicación del primer teorema de dualidad valen las equivalencias entre `foldr` y `foldl` presentadas más arriba en los ejemplos (R1)–(R5), (L1)–(L5). Análogamente, los ejemplos (R6)–(R8), (L6)–(L8) ilustran el segundo teorema de dualidad.

Criterios de elección entre `foldlr` y `foldl`

Los ejemplos recién mencionados muestran también que las equivalencias afirmadas por los dos teoremas de dualidad pueden fallar cuando la lista `xs` no es finita y terminada. Como hemos visto para el caso de las funciones `and`, `or` y `takeWhile`, las llamadas de la forma `(foldr (⊕) e xs)` pueden terminar con éxito en ocasiones aunque `xs` sea infinita, siempre que `(⊕)` no sea estricta con respecto a su segundo parámetro. Incluso para el caso de que `xs` sea finita y terminada, si `(⊕)` no es estricta con respecto a su segundo parámetro la ejecución de `foldr` puede terminar sin recorrer por completo `xs`, mientras que `foldl` siempre la recorre por completo.

Para problemas que exijan un recorrido completo de la lista a procesar, `foldl` conduce frecuentemente a algoritmos más eficientes. Por ejemplo, para el problema del cálculo de la inversa de una lista, `foldr` conduce a la función ingenua `inversa` con tiempo de ejecución $\mathcal{O}(n^2)$, mientras que `foldl` conduce a la función eficiente `reverse` con tiempo de ejecución $\mathcal{O}(n)$.

Debido a estas consideraciones, un criterio pragmático útil para la programación es que `foldr` es preferible en general a `foldl` en todos aquellos casos en que la función que se desea definir no exije un recorrido completo de la lista dada como parámetro. Este criterio tiene como excepción algunas situaciones especiales en las que el tiempo de ejecución de `(⊕)` depende críticamente del tamaño de su primer parámetro. Por ejemplo, para definir la función `concat` el uso de `foldr` conduce a un algoritmo más eficiente que `foldl`, puesto que en este caso `(⊕)` es la operación de concatenación de listas `(++)`, cuya tiempo de ejecución es proporcional al tamaño de su primer parámetro.

Operadores `fold` para listas no vacías

Algunas funciones de procesamiento de listas no están definidas para la lista vacía, y no pueden ser definidas mediante `foldr` y `foldl`. Para tratar estos casos, Haskell ofrece los operadores `foldr1` y `foldl1`, definidos como sigue:

```
foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)   = f x (foldr1 f xs)

foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)   = foldl f x xs
```

Un uso típico de `foldl1` aparece en el preludio de Haskell, para definir funciones encargadas de calcular el máximo y el mínimo de una lista no vacía de elementos de un tipo con orden:

```
maximum, minimum :: Ord a => [a] -> a
maximum           = foldl1 max
minimum           = foldl1 min
```

Las dos definiciones anteriores también se podrían haber formulado mediante `foldr1`. No se ha hecho así debido a que la función definida requiere un recorrido completo de la lista dada, por lo cual es preferible optar por recursión final.

Aplicación de `foldr` a un problema de procesamiento de textos

Un ejemplo interesante de uso de los operadores `foldr` y `foldr1` se presenta al diseñar una solución para el problema de descomponer en líneas un texto dado. Para abordar este problema, supondremos definidos los tipos sinónimos

```
type Texto = String

type Linea = String
```

que usaremos bajo el supuesto práctico de que las cadenas de caracteres utilizadas con la intención de representar una línea no contengan el carácter de control `'\n'` (salto de línea). Para descomponer en líneas un texto dado, entenderemos que cada aparición de `'\n'` en el texto actúa como *separador* entre dos líneas, de modo que un texto que contenga n saltos de línea se descompondrá en $n+1$ líneas. Según esto, un texto vacío (que incluye 0 saltos de línea) se descompone dando lugar a una sola línea vacía.

La función `separaLineas` definida a continuación calcula la lista de líneas resultante de descomponer un texto dado con el criterio antedicho. Se define como caso particular de `separaCon`, una función más general que descompone una lista dada `xs :: [a]` utilizando como separador un elemento dado `s :: a`:

```
separaLineas :: Texto -> [Linea]
separaLineas  = separaCon '\n'
```

```
separaCon   :: Eq a => a -> [a] -> [[a]]
separaCon s  = foldr (saltaEn s) [[]]

saltaEn              :: Eq a => a -> a -> [[a]] -> [[a]]
saltaEn s x (xs:xss)
   | s == x           = []:xs:xss
   | s /= x           = (x:xs):xss
```

Para recomponer un texto a partir de la lista de sus líneas, se requiere una función que se comporte como la inversa de `separaLineas`. Dicha función, llamada `juntaLineas`, espera recibir como parámetro a una lista no vacía de líneas, y se puede definir como caso particular de una función más general llamada `juntaCon`:

```
juntaLineas :: [Linea] -> Texto
juntaLineas  = juntaCon '\n'

juntaCon   :: a -> [[a]] -> [a]
juntaCon s  = foldr1 (pegaCon s)

pegaCon         :: a -> [a] -> [a] -> [a]
pegaCon s xs ys  = xs ++ [s] ++ ys
```

Es posible demostrar dos leyes que expresan la relación entre las funciones `separaLineas` y `juntaLineas`. Suponiendo cualquier texto `txt` y cualquier lista de líneas `lln`, no vacía y formada por líneas que no contengan al caracter `'\n'`, se verifica:

```
juntaLineas (separaLineas txt) = txt

separaLineas (juntaLineas lln) = lln
```

En el apéndice A.16 se encuentra un módulo Hugs 98 que permite probar el comportamiento de las funciones `separaLineas` y `juntaLineas`. El haberlas definido de forma tan general permite modificar fácilmente sus definiciones para programar otras funciones que resuelvan similares, como por ejemplo la descomposición de un texto en palabras.

En el preludio de Haskell se encuentran predefinidas dos funciones similares a `separaLineas` y `juntaLineas`, denominadas `lines` y `unlines`. Su comportamiento difiere de las funciones que hemos presentado aquí para el caso de "textos anómalos" que solo contengan saltos de línea:

```
> lines ""
[]
> separaLineas ""
[""]
```

```
> lines "\n"
[""]
> separaLineas "\n"
["",""]
```

En particular, vemos que `lines` descompone un texto vacío dando una lista vacía de líneas. En cuanto a `unlines`, está programada para comportarse como inversa de `lines`. El estudio de las definiciones de `lines` y `unlines` en el preludio de Haskell se deja como ejercicio para el lector.

Operadores `foldl` con acumulador impaciente

Al emplear recursión final, tanto `foldl` como `foldl1` pueden ser implementadas sin necesidad de que el sistema consuma memoria para mantener una pila de llamadas. Sin embargo, la evaluación perezosa retrasa la evaluación del acumulador hasta el momento en que termina el recorrido de la lista. El consumo de memoria causado por este comportamiento se puede reducir empleando los operadores `foldl'` y `foldl1'`, definidas como sigue:

```
foldl'           :: (b -> a -> b) -> b -> [a] -> b
foldl' f e []     = e
foldl' f e (x:xs) = (foldl' f $! f e x) xs

foldl1'           :: (a -> a -> a) -> [a] -> a
foldl1' f (x:xs)   = foldl' f x xs
```

El preludio de Haskell incluye `foldl'` definida del modo anterior, aunque no `foldl1'`. En la definición es crucial el uso del operador `($!)`, que sirve para forzar que un parámetro de una función sea evaluado a forma normal antes de evaluar la llamada a la función. Está incluido en el preludio como primitiva dependiente de la implementación:

```
    primitive seq           :: a -> b -> b
--  (seq e1 e2) evalua en secuencia primero e1 y luego e2,
--  devolviendo el valor de e2.

    primitive ($!) "strict" :: (a -> b) -> a -> b
--  Evalua en secuencia primero x y luego f x
--  f $! x               = x `seq` f x
```

Un ejemplo típico de uso de `foldl'` es la definición de la función que calcula la longitud de una lista, incluida también en el preludio:

```
length           :: [a] -> Int
length            = foldl' (\n _ -> n + 1) 0
```

Operadores `scan`

La aplicación de una función definida mediante `foldl` a todos los *segmentos iniciales* de una lista dada es una operación útil. Para realizarla se utiliza el operador `scanl`, que se puede especificar como sigue:

```
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
scanl f e  = map (foldl f e) . inits

inits                    :: [a] -> [[a]]
inits []                  = [[]]
inits (x:xs)              = [[]] ++ map (x:) (inits xs)
```

Utilizando la especificación anterior como programa ejecutable, y suponiendo que `(f x)` se evalúe en tiempo $\mathcal{O}(1)$ para cualquier `x :: a`, el tiempo de ejecución de `(scan f e xs)` para una lista finita y terminada `xs :: [a]` de longitud n sería $\mathcal{O}(n^2)$. Transformando la especificación mediante la técnica de plegado-desplegado, se obtiene una versión optimizada de `scanl` con la cual el tiempo de ejecución se reduce a $\mathcal{O}(n)$:

```
scanl             :: (b -> a -> b) -> b -> [a] -> [b]
scanl f e     []   = [e]
scanl f e (x:xs)   = e : scanl f (f e x) xs
```

La definición de `scanl` incluida en el preludio de Haskell es equivalente a la anterior e igualmente eficiente, aunque no idéntica. Los detalles de la transformación de plegado-desplegado se pueden estudiar en la sección 4.5 del texto [1].

Usando `scanl` es muy fácil programar el efecto producido por la acumulación de una cierta operación `f` a lo largo de los segmentos iniciales de una lista. Por ejemplo, las dos funciones que siguen se encargan de acumular sumas y productos, respectivamente, a lo largo de los segmentos iniciales de una lista de números:

```
acuSum, acuProd  :: Num a => [a] -> [a]
acuSum            = scanl  (+) 0
acuProd           = scanl  (*) 1
```

Teniendo en cuenta que el cuadrado de n es igual a la suma acumulada de los n primeros números impares, y que el factorial de n es igual al producto acumulado de los n primeros números positivos, las funciones anteriores se pueden emplear a su vez para el cálculo de factoriales y cuadrados, definiendo:

```
cuadrado, factorial :: Int  -> Int
cuadrado   n         = acuSum  [1,3..] !! n
factorial  n         = acuProd [1..]   !! n
```

De manera dual a `scanl` se puede especificar el operador `scanr`, que se utiliza para calcular el resultado de aplicar una función definida mediante `foldr` a todos los *segmentos finales* de una lista dada:

```
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e  = map (foldr f e) . tails

tails                   :: [a] -> [[a]]
tails []                 = [[]]
tails xs@(_:xs')         = xs : tails xs'
```

Si la especificación anterior se aceptase como definición ejecutable de `scanr`, se tendría un problema de ineficiencia similar al descrito más arriba para el caso de `scanl`. La técnica de plegado-desplegado permite transformar la especificación anterior en la definión más eficiente que se encuentra en el preludio:

```
scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e     []   = [e]
scanr f e (x:xs)   = f x y : ys
                     where ys@(y:_) = scanr f e xs
```

Operadores `scan` para listas no vacías

Del mismo modo que los operadores `fold`, también los operadores `scan` admiten variantes diseñadas para procesar exclusivamente listas no vacías. A continuación se muestran sus definiciones optimizadas, las cuales se pueden derivar mediante plegado-desplegado a partir de las especificaciones ineficientes indicadas en cada caso:

```
scanl1           :: (a -> a -> a) -> [a] -> [a]

--  scanl1 f      = map (foldl1 f) . inits1
--  inits1 (x:xs) = map (x:) (inits xs)

scanl1 f (x:xs)   = scanl f x xs

scanr1           :: (a -> a -> a) -> [a] -> [a]

--  scanr1 f          = map (foldr1 f) . tails1
--  tails1 xs@(x:xs')
--    | null xs'      = [[x]]
--    | otherwise     = xs : tails1 xs'

scanr1 f xs@(x:xs')
  | null xs'         = [x]
  | otherwise        = f x y : ys
                       where ys@(y:_) = scanr1 f xs'
```

Para comprender las definiciones anteriores, téngase en cuenta que las funciones `inits1` y `tails1` calculan, respectivamente, los segmentos iniciales no vacíos y los segmentos finales no vacíos de una lista no vacía dada.

Operadores `scanl` con acumulador impaciente

Por las mismas razones ya explicadas para el caso de `foldl`, es útil disponer de una variante de los operadores `scanl` que fuerce la evaluación impaciente del acumulador. Este comportamiento se puede obtener empleando los operadores `scanl'` y `scanl1'`, cuyas definiciones (no incluidas en el preludio de Haskell) se indican a continuación:

```
scanl'             :: (b -> a -> b) -> b -> [a] -> [b]
scanl' f e     []   = [e]
scanl' f e (x:xs)   = e : (scanl' f $! f e x) xs

scanl1'           :: (a -> a -> a) -> [a] -> [a]
scanl1' f (x:xs)   = scanl' f x xs
```

En el apéndice A.17 se incluye un módulo Hugs 98 que ilustra el uso de los diversos operadores `scanl` para programar el cálculo de cuadrados y factoriales de números naturales.

Aplicación: Gráficos de tortuga

Finalizamos esta sección con une ejemplo tomado del texto [2]. Se trata de escribir un programa funcional que permita dibujar gráficos rudimentarios en la pantalla, por medio de una *tortuga*. Se entiende que la "tortuga" es un dispositivo provisto de una plumilla, que se puede desplazar sobre un plano discreto, y deja marcas en aquellos puntos de su trayectoria por los que haya pasado teniendo la plumilla bajada.

El módulo del apéndice A.18 contiene un programa Hugs 98 con el comportamiento deseado. La función principal

```
ejecuta :: Programa -> IO()
```

espera como parámetro un *programa* `p` para la tortuga, y ejecuta dicho programa mostrando en la pantalla el dibujo que la tortuga produce al obedecer las instrucciones del programa `p`. Los tipos que representan los programas de la tortuga y las instrucciones de los que estos se componen, se definen del siguiente modo:

```
type Programa     = [Instruccion]
type Instrucccion = Estado -> Estado
```

La idea es admitir como instrucción para la tortuga cualquier función que exprese un cambio de estado de la misma; donde el tipo que representa los posibles estados de la tortuga es como sigue:

```
   type Estado = (Direccion,Pluma,Punto)

   type Direccion = Int

-- 0: Direccion del semieje X -
-- 1: Direccion del semieje Y +
-- 2: Direccion del semieje X +
-- 3: Direccion del semieje Y -

   type Pluma = Bool

-- True:  Pluma bajada (dibuja)
-- False: Pluma alzada (no dibuja)

   type Punto = (Int,Int)
```

En A.18 se incluyen las definiciones de varias instrucciones básicas, que permiten a la tortuga avanzar, girar, subir y bajar la plumilla. La generalidad del tipo `Instruccion` permitiría fácilmente modificar el programa, definiendo otras instrucciones.

La definición de la función `ejecuta` es la parte más interesante de este ejemplo. Se apoya en varias funciones auxiliares. La idea es que un programa de tortuga `p :: Programa` se interpreta calulando sucesivamente la serie dde estados por los que pasa la tortuga obedeciendo las instrucciones del programa, la serie de puntos por los que ha pasado con la plumilla bajada, el dibujo resultante, y finalmente la cadena de caracteres que debe ser escrita en pantalla para mostrar dicho dibujo. Se supone que la tortuga siempre parte del mismo estado inicial, definido así:

```
-- Estado inicial de la tortuga:
-- En el origen, mirando hacia X -, con la pluma alzada.

   estadoInicial :: Estado
   estadoInicial = (0,False,(0,0))
```

Admás, para representar los dibujos se ha elegido los tipos sinónimos

```
   type Dibujo = [Fila]
   type Fila   = String
```

con la intención de identificar a un dibujo con la lista de las filas que aparecerán (desde arriba hacia abajo) cuando el dibujo sea visualizado en la pantalla.

Se recomienda al lector el estudio detallado del código de A.18 y la realización de pruebas de ejecución. El programa emplea muchos de los recursos de programación estudiados a lo largo de todo este capítulo, incluyendo listas intensionales y operadores `scan`. Se incluyen algunas definiciones de funciones que sirven para construir programas de tortuga. Por ejemplo, la función

```
cuadrado :: Int -> Programa
```

está definida de tal manera que `(cuadrado l)` da como resultado un programa de tortuga cuya ejecución dibuja un cuadrado de lado `l`, según muestra la prueba de ejecución siguiente:

```
Tortuga> ejecuta (cuadrado 3)

* * * *
*     *
*     *
* * * *
```

Ejercicios

1. Usando `foldr` y sin utilizar recursión explícita, construye definiciones de la función `filter`.

2. Generalizando la función `valDec` estudiada en uno de los ejemplos de esta sección, define una función `val :: Integer -> [Integer] -> Integer` tal que `(val b ds)` devuelva el valor entero cuya representación en base b corresponde a la lista de dígitos `ds`. Se supone que `b` es un número entero mayor que 1, y que los elementos de `ds` son enteros no negativos menores que `b`.

3. Usando `fold`, define una función

   ```
   remdups :: Eq a => [a] -> [a]
   ```

   tal que `(remdups xs)` devuelva la lista resultante de eliminar de `xs` apariciones consecutivas de elementos iguales. ¿Qué es preferible en este caso, `foldr` o `foldl`?.

4. Define una función

   ```
   insert :: Ord a => a -> [a] -> [a]
   ```

   tal que `(insert x xs)` inserte `x` en `xs` en una posición adecuada para que la lista resultante quede ordenada en orden no decreciente, si `xs` lo estaba. Usando `insert` y `foldr`, define una función

   ```
   insertSort :: Ord a => [a] -> [a]
   ```

   que ordene listas empleando el algoritmo de ordenación por inserción.

5. Define dos funciones llamadas `separaPalabras` y `juntaPalabras` que sirvan, respectivamente, para descomponer un texto en la lista de las palabras que lo forman y recomponerlo a partir de dicha lista.
   Indicación: considera que una palabra es una cadena no vacía de caracteres que no contenga ni blancos ni saltos de línea. Define las funciones pedidas reutilizando lo más posible las definiciones de las funciones `separaLineas` y `juntaLineas` estudiadas en esta sección.

6. Comprueba que el primer teorema de dualidad entre `foldr` y `foldl` se deduce como caso particular del primero. Comprueba también que las hipótesis (A) y (E) necesarias paar que sea aplicable el segundo teorema de dualidad, se cumplen en el caso de los ejemplos (R6)–(R8), (L6)–(L8) estudiados en esta sección.

7. Demuestra el *tercer teorema de dualidad* para `foldr` y `foldl`, que afirma:

   `foldr` $(\oplus)$ `e xs = foldl` $(\otimes)$ `e ys`

   siempre que `xs` sea finita y terminada, `ys` $=$ `reverse xs`, y $(\otimes) =$ `flip` $(\oplus)$.

8. Demuestra la siguiente propiedad, conocida como *ley de fusión* de `foldr` y `map`:

   `foldr` $(\oplus)$ `e . map f = foldr` $(\otimes)$ `e`
   siendo `x` $\otimes$ `y = f x` $\oplus$ `y`; o lo que es lo mismo, $(\otimes)$ `=` $(\oplus)$ `. f`

9. Considera la *definición ineficiente* de la función `scanl`

   ```
   scanl     :: (b -> a -> b) -> b -> [a] -> [b]
   scanl f e  = map (foldl f e) . inits
   ```

   como *especificación*, y emplea el método de plegado-desplegado para transformarla en la *definición eficiente* que sigue:

   ```
   scanl             :: (b -> a-> b) -> b -> [a] -> [b]
   scanl f e     []   = [e]
   scanl f e (x:xs)   = e : scanl f (f e x) xs
   ```

   Sugerencia: una solución de este problema se encuentra en la sección 4.5 del texto [1].

10. Estudia las funciones `cuadrado` y `cuadrados` utilizadas en el programa de los gráficos de tortuga para construir programas de tortuga cuya ejecución dibuja, respectivamente, un cuadrado o una serie de cuadrados. Define otras funciones que construyan programas de tortuga cuya ejecución realice otros dibujos de tu invención.

11. Este ejercicio y el siguiente también se refieren a la programación funcional de gráficos de tortuga. Define una función `depura :: Traza -> Traza`, tal que `depura ps` devuelva una traza con los mismos puntos que `ps`, pero ordenados en orden lexicográfico y sin repeticiones.

12. Suponiendo una traza depurada como en el ejercicio anterior, la función `bitmap :: Traza -> Bitmap` se puede optimizar. Estudia la optimización `fastbitmap` que aparece al final del tema 4.4 (pg. 96) del texto de Bird y Wadler [2].

3.5 Interacción con el exterior

Hasta ahora solamente hemos considerado ejemplos sencillos de programación en Haskell, donde la comunicación con el usuario se limitaba casi siempre a solicitar la evaluación de expresiones a través de la línea de órdenes del intérprete. En la práctica, la mayoría de las aplicaciones requieren formas más complejas de interacción con el exterior. En esta sección estudiamos la programación de operaciones de entrada/salida en Haskell, que será de utilidad en el resto del curso.

El tipo de los procesos de entrada/salida

Los procesos de entrada/salida siempre se representan como expresiones del tipo predefinido parametrizado `IO a`. Al evaluar un proceso `p :: IO a`, se obtienen:

- Un *valor* de tipo `IO a`, que representa la estructura interna del proceso, y no es visible para el usuario.
- Un *resultado* de tipo `a`, que hay que imaginar como el resultado que devuelve el proceso al terminar su ejecución. Tampoco es visible directamente para el usuario, pero juega un papel en la comunicación entre procesos, según veremos más abajo.
- Un *efecto* que modifica el entorno de cómputo, realizando acciones que pueden cambiar el estado de los diversos dispositivos de entrada/salida disponibles. Excepcionalmente, el efecto de algunos procesos muy simples es nulo (es decir, no modifica el entorno).

Según el tipo al que se vincule en cada caso el parámetro `a`, se tienen varios tipos concretos de procesos de entrada/salida. Por ejemplo, un proceso encargado de leer en un archivo de texto y devolver como reultado su contenido, tendrá el tipo `IO String`. Como caso especial se tienen los procesos de tipo `IO ()`, que devuelven como resultado la tupla vacía `()`. En este

caso, se entiende que el resultado devuelto por el proceso no tiene interés.

Combinadores de procesos

El tipo `IO a` se comporta como un tipo de datos abstracto, por lo cual no se puede observar la estructura interna de los procesos. Para progamar procesos de entrada/salida, se dispone de tres *combinadores de procesos*, y de una serie de funciones de entrada/salida más específicas. Los combinadores de procesos son las operaciones más básicas de que se dispone para formar procesos complejos a partir de otros más sencillos. Los indicamos a continuación.

```
   return  :: a -> IO a
-- Dado r :: a,
-- el proceso (return r) no causa efectos,
-- y termina devolviendo el resultado r.

   (>>=)   :: IO a -> (a ->IO b) -> (IO b)
-- Dados p :: IO a, c :: a -> IO b,
-- el proceso (p >>= c) comienza ejecutando p, que devuelve r;
-- continua ejecutando (c r),
-- y termina devolviendo el resultado que devuelva (c r).

   (>>)    :: IO a -> IO b -> IO b
   p >> q   = p >>= (\ _ -> q)
-- Dados p :: IO a, q :: IO b,
-- el proceso (p >> q) se comporta como la composicion secuencial
-- de p y q; ignora el resultado de p y devuelve el resultado de q.
```

Las implementaciones de `return` y `(>>=)` son dependientes del sistema, mientras que `(>>)` se define como caso particular de `(>>=)`. Además del tipo `IO a`, existen en Haskell otros tipos llamados *mónadas*, que disponen de operaciones similares a `return`, `(>>=)` y `(>>)`. De hecho, estas operaciones son métodos de una clase de tipos llamada `Monad`, que tiene otros ejemplares además del tipo `IO`. Por esta razón, se dice que Haskell utiliza *entrada/salida monádica*.

La notación `do`

Para componer procesos por medio de los combinadores `(>>=)` y `(>>)` está permitido usar una notación abreviada, basada en dos convenios: `do {p; q}` abrevia `p >> q`; y `do {r <- p; q}` abrevia `p >>= (\r -> q)`. Análogamente, es posible componer más de dos procesos. Las llaves y puntos y comas se pueden reemplazar por indentación, y los dos convenios se pueden combinar dentro de un mismo texto. Así, la notación `do` da lugar a expresiones del estilo

```
do r1 <- p1
   p2
   p3
   r4 <- p4
   p5
```

donde `p2`, `p3` y `p4` pueden utilizar `r1`, mientras que `p5` puede utilizar `r1` y `r4`. La notación `do` también se puede anidar, como veremos más adelante.

Funciones de entrada/salida usando teclado y terminal

Las funciones presentadas a continuación, junto con los combinadores de procesos (o, equivalentemente, la notación `do`), permiten programar operaciones de entrada/salida a través de los dos canales standard. Usualmente, el canal standard de entrada (CE) es el teclado, y el canal standard de salida (CS) es la pantalla. Cada función va acompañada de un comentario. En los casos donde no aparece ninguna definición, hay que entender que se trata de una operación primitiva, cuya impementación depende del sistema. Comenzamos por funciones encargadas de *operaciones de salida*:

```
   done :: IO ()
   done  = return ()

-- El efecto de done es nulo.
-- El resultado devuelto es ().

-- OJO: done NO esta predefinida en Haskell 98;
-- los programas que usen done deben incluir su definicion.

   putChar :: Char -> IO ()

-- El efecto de (putChar x) es imprimir el caracter x en el CS.
-- El resultado devuelto es ().

   putStr        :: String -> IO ()
   putStr []      = done
   putStr (x:xs)  = do putChar x
                       putStr xs

-- El efecto de (putStr xs) es imprimir la cadena xs en el CS.
-- El resultado devuelto es ().

   putStrLn    :: String -> IO ()
   putStrLn xs  = do putStr xs
                     putChar '\n'

-- El efecto de (putStr xs) es imprimir la cadena xs en el CS,
-- seguida de un salto de linea.
```

```
-- El resultado devuelto es ().

   print    :: Show a => a -> IO ()
   print x   = putStrLn (show x)

-- El efecto de (print x) es imprimir en el CS la
-- cadena de caracteres que representa a x.
-- El resultado devuelto es ().
```

Para *operaciones de entrada* se dispone de las funciones siguientes:

```
   getChar :: IO Char

-- El efecto de getChar es leer el caracter en curso del CE.
-- El resultado devuelto es el caracter leido.

   getLine :: IO String
   getLine  = do x <- getChar
                 if x == '\n'
                    then return []
                    else do xs <- getLine
                            return (x:xs)

-- El efecto de getLine es leer la linea en curso del CE.
-- El resultado devuelto es la linea leida.

   getContents :: IO String

-- El efecto de getContents es leer todo el contenido del CE
-- (desde el caracter en curso hasta el final).
-- El resultado devuelto es la cadena de caracteres leida.

   interact   :: (String -> String) -> IO ()
   interact f  = do xs <- getContents
                    putStr (f xs)

-- El efecto de interact es escribir (f xs) en el CS,
-- siendo xs el contenido leido del CE.
-- La lectura se produce linea a linea, segun lo demande f;
-- no es necesario que xs este tecleado por completo al principio.
-- El resultado devuelto es ().

   readIO :: Read a => String -> IO a
-- Definicion en el preludio standard.

-- El efecto de (readIO xs) es nulo.
-- El resultado devuelto es el valor x :: a
-- representado por xs :: String.

   readLn :: Read a => IO a
```

```
readLn  = do line  <- getLine
             value <- readIO line
             return value

-- El efecto de readLn es leer la linea en curso del CE,
-- interpretandola como un valor de tipo a.
-- El resultado devuelto es el valor de tipo a
-- representado por la linea leida.
```

Como ejemplo sencillo de uso de las funciones de entrada/salida presentadas hasta ahora, mostramos un programa incordiante que se empeña en que usuario adivine su nombre. Para probarlo se puede utilizar el módulo Hugs 98 del apéndice A.19.

```
incordia :: IO ()
incordia  = do putStrLn "Adivina como me llamo: "
               insiste

insiste :: IO ()
insiste  = do linea <- getLine
              if linea == "Incordion"
                 then do putStrLn "Acertaste. Abur."
                         done
                 else do putStrLn "Fallaste. Prueba otra vez."
                         insiste
```

Programación de procesos interactivos mediante `interact`

La función `interact` presentada más arriba se utiliza a veces para programar procesos interactivos sencillos que reaccionan al flujo de entrada introducido por el usuario a través del teclado, produciendo un flujo de salida en el terminal. Para programar interacciones de este estilo se puede seguir el siguiente esquema:

```
type Entrada     = String
type Salida      = String
type Interaccion = Entrada -> Salida

interaccion :: Interaccion
--  definicion adecuada a la interaccion que se  desee

proceso :: IO ()
prceso   = interact interaccion
```

Como ejemplo concreto, podemos utilizar esta técnica para programar el proceso interactivo `invierteLineas`, que va presentando en el canal de salida (terminal) las líneas introducidas por el usuario en el canal de entrada (teclado), pero invertidas. El proceso comienza advirtiendo al usuario que la

interacción terminará tan pronto como se encuentre en el canal de entrada una línea que contenga el texto "stop".

```
invLin         :: Interaccion
invLin entrada  = "Hola!\n" ++
                  "Invertire todas las lineas que teclees.\n" ++
                  "Me detendre cuando teclees stop\n\n" ++
                  salida ++
                  "\n\nAdios!"
                  where lineas = takeWhile (/= "stop") (lines entrada)
                        lineasInversas = [reverse linea | linea <- lineas]
                        salida = unlines lineasInversas

inversor :: IO ()
inversor  = interact invLin
```

Como se observa en este ejemplo, una interacción programada mediante esta técnica puede comenzar a producir información de salida antes de que el usuario haya tecleado ninguna información de entrada. Ello se debe a que la función de tipo `Interaccion` empleada para transformar el flujo de entrada en el flujo de salida (en este caso, `invLin`) se evalúa perezosamente.

El uso de la función `interact` no es el único recurso disponible en Haskell para programar proceso interactivos que reaccionen a la entrada del teclado. También es posible programar tales procesos empleando otras funciones de entrada/salida y la notación `do`, como ya hemos visto más arriba en el ejemplo del proceso `incordia`. El proceso `inversor'` definido más abajo tiene un comportamiento equivalente a `inversor` y está programado en un estilo similar al empleado para `incordia`:

```
inversor' :: IO ()
inversor'  = do putStrLn ("Hola!\n" ++
                          "Invertire todas las lineas que teclees.\n" ++
                          "Me detendre cuando teclees stop\n")
                invierteLineas

invierteLineas :: IO ()
invierteLineas  = do linea <- getLine
                     if linea == "stop"
                        then do putStrLn "\nAdios!"
                                done
                        else do putStrLn (reverse linea)
                                invierteLineas
```

El módulo Hugs 98 del apéndice A.20 permite ejecutar pruebas del funcionamiento de los procesos `inversor` e `inversor'`, que en efecto se comportan de la misma manera. La programación de `inversor'` resulta algo más natural que la de `inversor`. En general, el uso de la función `interact`

suele conducir a programas menos claros, excepto en el caso de interacciones muy simples.

Funciones de entrada/salida usando archivos

Para programar procesos de entrada/salida con lectura y/o escritura en archivos de texto, se pueden utilizar los combinadores de procesos junto con las funciones indicadas más abajo. Para leer y/o escribir en archivos valores que no sean cadenas de caracteres, es necesario convertirlos primero a cadenas de caracteres. Para ello se pueden usar las funciones `read` y `show` (para tipos de las clases `Read` y `Show`), u otras definidas por el programador.

```
   type FilePath = String

-- Tipo sinonimo.
-- Se usa para representar las sendas de acceso a archivos.

   writeFile :: FilePath -> String -> IO ()

-- El efecto de (writeFile s xs) es escribir xs como nuevo
-- contenido del archivo de texto localizado en la senda s,
-- borrando el anterior contenido.
-- El resultado devuelto es ().

   appendFile :: FilePath -> String -> IO ()

-- El efecto de (appendFile s xs) es concatenar xs al final
-- del contenido del archivo de texto localizado en la senda s,
-- sin borrar el anterior contenido.
-- El resultado devuelto es ().

   readFile :: FilePath -> IO String

-- El efecto de (readFile s) es leer el contenido
-- del archivo de texto localizado en la senda s.
-- El resultado devuelto es la cadena de caracteres leida.
```

Como ejemplo sencillo de uso de las funciones de lectura y escritura en archivos de texto, consideremos el problema de copiar un archivo de texto en otro, pero invirtiendo el orden de los caracteres de cada línea. Un programa Hugs 98 que resuelve ese problema se encuentra en el apéndice A.21. El proceso principal `copion` se encarga de la interaccion con el usuario, y llama a la función auxiliar `copiaInv` para efectuar la copia propiamente dicha:

```
copion :: IO ()
copion  = do putStr "Teclea la senda origen: "
             origen <- readLn
             putStr "Teclea la senda destino: "
```

```
                destino <- readLn
                copiaInv origen destino
                putStrLn ("Copia inversa del archivo "
                             ++ origen ++ "\n" ++
                          "realizada en el archivo "
                             ++ destino)

copiaInv                 :: FilePath -> FilePath -> IO ()
copiaInv origen destino  = do cont <- readFile origen
                              let contInv = unlines
                                            (map reverse (lines cont))
                                  in writeFile destino contInv
```

Obsérvese que la función `copiaInv` se comporta como un proceso de entrada/salida una vez que recibe como parámetros actuales las sendas de los archivos origen y destino de la copia. En general, al programar aplicaciones que requieran proceso de entrada/salida es frecuente el uso de funciones de tipo $\texttt{A}_1$ `->` $\cdots$ $\texttt{A}_n$ `->` `IO R`, que se comportan como un proceso de entrada/salida de tipo `IO R` una vez que reciben n parámetros de tipos $\texttt{A}_1$, $\cdots$, $\texttt{A}_n$. En lo sucesivo, llamaremos *procesos de entrada/salida parametrizados* a las funciones de este tipo.

Otras funciones de entrada/salida

Existen funciones para tratamiento de errores aparecidos en el curso de una operación de entrada/salida. También hay funciones que sirven para realizar entrada/salida en archivos, asociándoles "asideros" (*handlers*). Omitimos estas cuestiones, que se pueden consultar en [5, 7].

Programación con operaciones de entrada/salida

Para finalizar, presentamos dos ejemplos sencillos de programación que utilizan las operaciones de entrada/salida vistas en esta sección, junto con otras técnics de procesamiento de listas estudiadas en el resto de este capítulo.

Ordenación de listas de personas

El módulo Hugs 98 del apéndice A.22 utiliza las funciones de ordenación de listas del módulo A.14 para ordenar listas de personas con respecto a diferentes criterios de ordenación. Las operaciones de entrada/salida estudiadas en esta sección se han utlilizado para programar una función auxiliar llamada `printPersonas`, que sirve para escribir en la pantalla una lista de personas en un formato legible. El estudio detallado del código de este módulo y la ejecución de pruebas se dejan como ejercicio para el lector.

Procesamiento de textos

Aquí consideramos un problema típico de *procesamiento de textos* inspirado en un ejemplo del texto [10]. Dado un texto sin formatear, tal como

```
The heat bloomed      in December as the
    carnival  season
            kicked into  gear.
Nearly helpless with sun and glare, I avoided Rio's
brilliant sidewalks
  and glittering beaches,
panting in dark   corners
and waiting out the inverted southern summer.
```

se desea obtener un texto alineado a la izquierda, sin espaciados múltiples y con líneas de a lo sumo 35 caracteres, del estilo:

```
The heat bloomed in December as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering beaches, panting in
dark corners and waiting out the
inverted southern summer.
```

Para resolver este problema resulta natural proceder en dos etapas:

1. Transformar el texto dado en una lista de palabras.
2. Reagrupar las palabras de la lista obtenida en la etapa anterior en un nuevo texto con líneas de a lo sumo 35 caracteres, evitando blancos a comienzo de línea y dejando un solo blanco entre cada dos palabras. A su vez, esta etapa se puede descomponer en dos subetapas:

   (a) Dividir la lista de palabras del texto inicial en varias listas de palabras, cada una de las cuales corresponderá a una de las líneas del texto formateado.

   (b) Componer el texto formateado a partir del resultado de la subetapa anterior. Para ello basta concatenar todas las palabras de las listas correspondientes a las diferentes líneas, intercalando saltos de línea en las posiciones adecuadas.

Para implementar este método en Haskell, se pueden representar los textos, líneas y palabras como cadenas de caracteres, tal como se hace en el módulo presentado en el apéndice A.23. La función principal de entre las definidas en este módulo es `procTexto`, un proceso de entrada/salida

que interactúa con el usuario para leer el contenido de un archivo de texto, formatearlo, y escribir el resultado en otro archivo de texto. Además de las funciones de entrada/salida estudiadas en esta sección, es esencial aquí la función `formatea`, definida como la composición de otras tres funciones:

- `saltaBlancos`, que elimina los blancos iniciales del texto de partida.
- `separaPalabras`, que corresponde a la primera de las dos etapas explicadas más arriba.
- `componTexto`, que corresponde a la segunda de las etapas antedichas. Su definición está presentada como la composición de dos funciones, correspondientes a las subetapas (a) y (b) mencionadas antes.

El estudio detallado del módulo A.23 y la ejecución de pruebas se dejan como ejercicio para el lector.

Ejercicios

1. Utilizando la función predefinida `interact` y funciones de tipo `Interacion` definidas adecuadamente, programa procesos de tipo `IO ()` que reaccionen al flujo de entrada del teclado del modo indicado a continuación:

   (a) `duplica :: IO ()`
   Proceso que lee las líneas de la entrada y las va trasladando a la salida, pero cada una copiada dos veces. Se detiene al leer una línea en blanco.

   (b) `incrementa :: IO ()`
   Proceso que leyendo un número entero de cada línea del canal de entrada, y escribiendo cada vez el siguiente del número leído en una nueva línea del canal de salida. Se detiene al leer una línea que contenga `"-1"`.

   (c) `promedia :: IO ()`
   Proceso que va leyendo un número entero de cada línea del canal de entrada, y escribiendo cada vez el promedio de todos los números leídos hastas el momento en una nueva línea del canal de salida. Se detiene al leer una línea que contenga `"-1"`.

2. Programa los mismos procesos de entrada/salida del ejercicio anterior en un estilo alternativo, sin utilizar la función `interact`.

3. Recuerda el tipo `Persona` utilizado en la sección 2.6 para representar personas con nombre, DNI y edad conocidas. Programa un proceso de entrada/salida parametrizado

```
imprimePersonas :: [Persona] -> IO ()
```

de modo que (`imprimePersonas ps`) escriba en el canal de salida todas las personas de la lista `ps`, cada una en una línea diferente, y con los tres atributos de cada persona separados por tabuladores dentro de cada línea.

4. Programa dos funciones de conversión

```
showPersonas :: [Persona] -> String
readPersonas :: String -> [Persona]
```

de modo que se cumpla `readPersonas . showPersonas = id`

5. Recordando que el tipo sinónimo

```
type FilePath = String
```

sirve para representar la senda de acceso a un archivo, programa los proceso de entrada/salida parametrizados que se especifican a dos funciones especifican a continuación. Usa las dos funciones del ejercicio anterior, así como las funciones predefinidas suministradas por Haskell para realizar operaciones de entrada/salida en archivos de texto.

```
   guarda   :: [Persona] -> FilePath -> IO ()
-- (guarda ps fp) escribe en el archivo
-- indicado por la senda fp una cadena
-- de caracteres que representa la lista
-- de personas ps.

   recupera :: FilePath -> IO [Persona]
-- (recupera fp) lee la cadena de caracteres contenida
-- en el archivo indicado por la senda fp,
-- y devuelve la lista de personas representada
-- por dicha cadena.
```

6. Rediseña `ProcTexto` de modo que el texto formateado se imprima ajustando todas sus líneas a ambos márgenes. Para ello, todas las líneas (excepto la última de cada párrafo) deberán rellenarse con blancos, distribuidos del modo más uniforme que se pueda. Además, entre cada dos parráfos consecutivos deberá dejarse exactamente una línea en blanco. El texto utilizado como ejemplo anteriormente debería quedar formateado de forma similar a la siguiente:

```
The heat bloomed in December as the
carnival  season  kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and  glittering beaches, panting in
dark  corners  and  waiting out the
inverted southern summer.
```

7. Sea la declaración `type Dibujo = [String]`, entendiendo que `d` = `[xs`$_0$`, ···, xs`$_{n-1}$`]` representa un dibujo cuyas *filas*, vistas de arriba a abajo, son las cadenas de caracteres `xs`$_i$.

   (a) Define dos funciones `altura, anchura :: Dibujo -> Int` que calculen las dimensiones de un dibujo.

   (b) Define una función `casilla :: Int -> Int -> Char -> Dibujo` tal que `casilla v h c` represente un rectángulo de altura `v` (vertical) y anchura `h` (horizontal), relleno con el carácter `c`.

   (c) Define una función `hueco :: Int -> Int -> Dibujo` tal que `hueco v h` represente un rectángulo en blanco de altura `v` (vertical) y anchura `h` (horizontal).

   (d) Define una función `sobre :: Dibujo -> Dibujo -> Dibujo` tal que `d1 'sobre' d2` dé el dibujo resultante de colocar `d1` sobre `d2`, suponiendo que ambos tengan la misma anchura.

   (e) Define una función `juntoA :: Dibujo -> Dibujo -> Dibujo` tal que `d1 'juntoA' d2` dé el dibujo resultante de colocar `d1` junto a `d2` (con `d2` a la derecha), suponiendo que ambos tengan la misma altura.

   (f) Define `apilar, extender :: [Dibujo] -> Dibujo`, tales que `apilar ds` (resp. `extender ds`) devuelva el resultado de colocar juntos todos los dibujos de la lista `ds`, en *columna* (resp. *en fila*). ¿Qué condición deben cumplir los dibujos miembros de la lista `ds` para que estas dos funciones tengan sentido?

   (g) Aprovechando las funciones definidas en los apartados anteriores, define una función que dibuje un tablero de ajedrez, con casillas de un tamaño dado como parámetro.

8. Define una función `muestra :: Dibujo -> IO()` que sirva para mostrar un dibujo `d` = `[xs`$_0$`, ···, xs`$_{n-1}$`]` en pantalla, haciendo aparecer las diferentes cadenas de caracteres `xs`$_i$ en líneas consecutivas.

Capítulo 4

Programación funcional con otros tipos de datos

4.1 Tipos de datos abstractos

El concepto de tipo abstracto de datos (TAD)

Los tipos de datos que hemos estudiado hasta el momento se pueden clasificar en cuatro categorías:

- *Tipos de datos predefinidos.* Vienen dados por el sistema. No es necesario que los usuarios conozcan la representación interna utilizada para implementar estos tipos. En cambio, están disponibles una serie de funciones predefinidas que permiten operar con ellos.

- *Tipos de datos construidos* (que incluyen como caso particular a los *tipos enumerados*). Son definidos por los usuarios. Como consecuencia, los usuarios conocen las *constructoras* utilizadas para representar los datos, y pueden usarlas en patrones a la hora de definir funciones que operen con datos de este tipo. En lenguajes del estilo de Haskell, algunos tipos de datos predefinidos (tales como los booleanos y las listas) son equivalentes a tipos de datos construidos. En estos casos, los usuarios conocen las constructoras del tipo, cosa que no sucede en general para otros tipos predefinidos (como por ejemplo `Int` o `Float`).

- *Tipos sinónimos.* Como ya sabemos, solo sirven para simplificar o aclarar la notación, y no introducen ninguna otra novedad.

- *Tipos nuevos.* Se pueden considerar como un caso especial de tipo construido con una nueva constructora y una implementación optimizada. En cierto modo se parecen a los tipos sinónimos, pero con la importante diferencia de que permiten declaraciones de ejemplar.

A partir de ahora nos va a interesar tener en cuenta una quinta categoría: los *tipos abstractos de datos* (brevemente, TADs). En general, y sea cual sea el lenguaje de programación que se utilice, se entiende que un tipo de datos es *abstracto* si se cumplen los requisitos siguientes:

- Los usuarios no pueden manipular la *representación interna* de los datos de un TAD.
- Un TAD lleva asociadas una serie de funciones (o procedimientos, o métodos, según el lenguaje de programación de que se trate), llamadas *operaciones públicas*, que se dejan a disposición de los usarios.
- El comportamiento de las operaciones públicas de un TAD tiene una *especificación* que permite a los usuarios entender que tareas realizan. La *implementación* de las operaciones de un TAD no es competencia de los usuarios.

Las ventajas de los TADs son similares a las de los tipos predefinidos. Pueden ser utilizados por personas diferentes de quienes los han diseñado e implementado. Los usuarios de un TAD siempre están obligados a operar sobre sus datos utilizando exclusivamente las operaciones públicas. Esto conduce a programas más claros y seguros que los que se obtendrían manejando directamente una representación concreta de los datos. Para entender la última afirmación piénsese que un tipo predefinido tal como `Int` es análogo a un TAD. Los usuarios solo pueden operar con enteros a través de las operaciones públicas, que en este caso son las operaciones aritméticas habituales, y no pueden operar directamente con la representación de los enteros como palabras binarias. En un lenguaje de más bajo nivel (ensamblador, por ejemplo), `Int` no funcionaría como un TAD, y sería posible operar directamente con la representación binaria de los enteros. Aunque ésto puede ser necesario para algunas aplicaciones especiales, también es cierto que conduce a programas menos legibles y que aumenta la probabilidad de cometer errores de programación.

Especificación de TADs

A continuación, vamos a estudiar con más detalle una metodología de *especificación* e *implementación* de TADs, que está orientada a lenguajes funcionales del estilo de Haskell. Como hemos dicho más arriba, un TAD debe tener siempre una *especificación* conocida por los usuarios, que permita entender qué son los datos de ese tipo y cómo se comportan sus operaciones. Para cumplir estos propósitos, la especificación debe aportar las siguientes informaciones:

1. *Nombre* del TAD. Aquí hay que tener en cuenta que un TAD puede depender de *parámetros formales*. Lo mismo que en el caso de los tipos

construidos, los parámetros formales (si los hay) se pueden indicar por medio de variables que representan tipos. En ciertos casos, algunos de estos parámetros pueden estar cualificados por la condición de que sean ejemplares de determinadas clases de tipos.

2. *Nombre y tipo* de cada una de las operaciones públicas del TAD.

3. *Leyes de comportamiento* de las operaciones del TAD.

El punto más difícil de precisar suele ser la especificación de las leyes de comportamiento de las operaciones. Para formularlas se usan muchas veces ecuaciones, que no se deben confundir con las ecuaciones ejecutables usadas como definición de las funciones en los programas funcionales. También es posible especificar el comportamiento de las operaciones mediante comentarios verbales suficientemente precisos, que se pueden apoyar cuando sea conveniente en el comportamiento de otros tipos de datos ya conocidos, o en modelos matemáticos. Nosotros aceptaremos este método.

Ejemplo. Para servir de soporte a diferentes programas que trabajen con fechas del calendario, podemos pensar en un TAD llamado `Fecha`, equipado operaciones adecuadas. Suponemos ya conocidos los tipos `DiaSemana`, `Dia`, `Mes` y `Anno` (correspondientes a los días de la semana, los días del del mes, los meses y los años, respectivamente). Una posible especificación de las operaciones públicas de `Fecha` se indica a continuación:

```
fechaLegal ::  Dia -> Mes -> Anno -> Bool
– Reconoce si un día, mes y año dados
– constituyen una fecha legal.
creaFecha ::  Dia -> Mes -> Anno -> Fecha
– Crea una fecha a partir de un día, mes y año dados,
– si se trata de una fecha legal.
dia ::  Fecha -> Dia
– Consulta el día de una fecha dada.
mes ::  Fecha -> Mes
– Consulta el mes de una fecha dada.
anno ::  Fecha -> Anno
– Consulta el año de una fecha dada.
distancia ::  Fecha -> Fecha -> Int
– Calcula la distancia (medida como número de días)
– entre dos fechas dadas.
desplaza ::  Fecha -> Int -> Fecha
– Calcula el resultado de desplazar una fecha
– un cierto número de días.
diaSemana ::  Fecha -> DiaSemana
– Calcula el día de la semana de una una fecha dada.
```

Implementación de TADs

La especificación de un TAD se interpone como una *barrera de abstracción* entre los usuarios y los implementadores. Pueden existir diferentes implementaciones correctas de un mismo TAD, sin que los usuarios perciban ninguna diferencia de comportamiento al observar los cómputos de programas que utilicen el TAD. Entre otras ventajas, ésto permite que un cualquier momento se pueda sustituir una implementación de un cierto TAD por otra más eficiente, sin necesidad de alterar los programas usuarios del TAD. En cualquier caso, toda implementación de un TAD debe ser *correcta* con respecto a su especificación. Para diseñar metódicamente una implementación correcta de un TAD previamente especificado, los pasos a seguir son:

1. Definición del *tipo representante*.
   Se llama así al tipo de datos elegido para representar al TAD en una implementación particular. En general, se pueden encontrar varias maneras de definir un tipo representante. Por ejemplo, el caso del TAD `Fecha`, dos posibles elecciones del tipo representante son:

   (a) `(Dia, Mes, Anno)`. Este tipo representante corresponde a la decisión de representar una fecha como la terna formada por el día, el mes y el año correspondientes. La idea de representar fechas de este modo ya ha aparecido antes en los ejercicios del capítulo 2, donde aún no conocíamos el concepto de TAD.

   (b) `Int`. Este tipo representante corresponde a la decisión de representar cada fecha como el número de días transcurridos desde el primer día de nuestra era hasta la fecha en cuestión. Nótese que este número puede ser negativo, en el caso de fechas anteriores a nuestra era.

   Cada una de estas dos elecciones conduce a una implementación diferente del TAD `Fecha`.

2. Definición del *invariante de la representación*.
   En ocasiones, no todos los datos del tipo representante valen para representar un dato del tipo abstracto. Por ejemplo, si elegimos `(Dia, Mes, Anno)` como tipo representante de `Fecha`, una terna `(d, m, a)` solamente valdrá para representar una fecha si el valor de `d` se corresponde con un día existente en el mes `m` del año `a` de nuestro calendario. Por ejemplo, una terna `(40, m, a)` nunca será válida, porque no hay meses de cuarenta días. Tampoco será válida la terna `(29, Febrero, 1999)`, porque el año `1999` no es bisisesto. En general, el *invariante de la representación* es la propiedad, definida en cada caso por el implementador, que se verifica para aquellos datos del tipo representante

que se aceptan como representantes válidos de un dato abstracto. En los sucesivo, usaremos la notación `valido x` (leido: `x` *es un representante válido*) para indicar que un dato `x` del tipo representante verifica el invariante de la representación. En algunos casos, el invariante de la representación es la propiedad trivial que se cumple siempre. Esto ocurre, por ejemplo, al elegir `Int` como tipo representante para el TAD `Fecha`, debido a que cualquier número entero sirve como representación válida de una fecha.

3. Definición de la *equivalencia abstracta*.
   En muchas implementaciones de TADs ocurre que datos no idénticos del tipo representante representan el mismo dato abstracto. Se llama *equivalencia abstracta* a una relación de equivalencia, definida en cada caso por el implementador, que relaciona entre sí a todos aquellos datos del tipo representante que representen un mismo dato abstracto. De ahora en adelante, usaremos la notación `equiv x x'` (leido: `x`, `x'` *son dos representantes equivalentes*) para indicar que los dos datos `x` y `x'` (pertenecientes al tipo representante) verifican la relación de equivalencia abastracta. En algunos casos, la equivalencia abastracta es la identidad, y solo se cumple cuando `x` y `x'` son idénticos. Esto sucede por ejemplo en las dos implementaciones consideradas más arriba para el TAD `Fecha`. Más adelante estudiaremos otras implementaciones de TADs en las cuales la equivalencia abstracta no es la identidad.

4. *Implementación de las operaciones*.
   Cada operación pública del TAD se debe implementar definiendo una función `f` que se corresponda con ella. Dicha función debe tener el tipo previsto en la especificación. Su definición debe programarse utilizando el tipo representante, y debe ser correcta en el siguiente sentido:

   (a) Para cualquier parámetro de `f` cuyo tipo sea el tipo representante, se puede suponer que el dato asociado a dicho parámetro verifica el invariante de la representación.

   (b) Si el tipo del resultado de `f` es el tipo representante, se debe garantizar que el resultado cumpla el invariante de la representación.

   (c) Entre los parámetros de `f` y su resultado se debe cumplir la relación prevista en la especificación. Al verificar ésto, los representantes equivalentes de un mismo dato abtracto se deben tratar como indistinguibles.

   Por ejemplo, en una implementación del TAD `Fecha`, la operación `distancia` se debe implementar definiendo una función `distancia :: Fecha -> Fecha -> Int`, la cual deberá calcular el resultado correcto siempre que las dos fechas dadas como parámetros cumplan

el invariante de la reepresentación. Obsérvese que la implementación de `distancia` no puede ser la misma para las dos elecciones del tipo representante de `Fecha` comentadas más arriba. En general, las definiciones de las funciones que implementan las operaciones de un TAD siempre dependen de cómo se hayan elgido el tipo representante, el invariante de la representación, y la equivalencia abstracta.

5. *Encapsulamiento.*
   La implementación de un TAD siempre debe garantizar que los usuarios del TAD no puedan manipular directamente al tipo representante, y solo puedan utilizarlo a través de las operaciones públicas. Para expresar ésto, se dice que el tipo representante debe quedar *encapsulado* en la implementación. En muchos lenguajes de programación, incluyendo Haskell, el encapsulamiento del tipo representante se logra empaquetando toda la implementación del TAD en un *módulo*, que exporta el TAD y las funciones que implementan sus operaciones, pero no deja acceso a la estructura interna del tipo representante.

Módulos de implementación de TADs en Haskell

Para diseñar módulos de implementación de TADs, es necesario explotar los recursos de *importación* y *exportación* del sistema de módulos de Haskell. Como ya sabemos, un programa Haskell consta en general de varios módulos, cada uno de los cuales *exporta* ciertas entidades e *importa* otras entidades de otros módulos. La estructura esquemática de un módulo en Haskell es:

```
module Nombre [ ( ... entidades exportadas ...) ] where
                ...
import [qualified] Nombre' [ ( ... entidades importadas ...) ]
                ...
declaraciones y definiciones
                ...
```

El diseño de módulos en Haskell 98 está sujeto a muchos convenios cuyos detalles se deben estudiar en la documentación del lenguaje; véanse las referencias [5, 7, 6]. Para la mayoría de las aplicaciones sencillas, basta tener en cuenta lo que sigue:

- El *Nombre* de un módulo debe ser un identificador que comience por una letra mayúscula. No se prohibe que el nombre de un módulo coincida con el nombre de alguna de las entidades exportadas. En los módulos utilizados para implementar un TAD, nosotros seguiremos la práctica de que el nombre del módulo coincida con el nombre del tipo abstracto de que se trate, el cual será siempre una de las entidades exportadas.

- Las *entidades exportadas* por un módulo pueden comprender tipos, funciones y otras cosas (e.g. operadores, clases de tipos, etc.). Las entidades exportadas se escriben entre paréntesis y separadas por comas, justo después del nombre del módulo. Los corchetes `[ ...]` que aparecen en el esquema indican que esta información es opcional. *Si no se escriben entidades exportadas, se sobreentiende que el módulo exporta todas las entidades que define.*

- Cuando se exporta un tipo construido, es posible decidir si se exportan o no sus constructoras. Para exportar un tipo sin exportar sus constructoras, basta con incluir el nombre del tipo entre las entidades exportadas, sin incluir los nombres de las constructoras. *Siempre que un tipo de datos construido (o un tipo nuevo) se use como tipo representante de un TAD, hay que exportarlo sin exportar sus constructoras.* En caso contrario, el tipo representante no quedaría encapsulado dentro de la implementación.

- Un módulo puede incluir declaraciones de importación (ninguna, una o varias). Si las hay, cada una debe ir encabezada por la palabra reservada `import` (opcionalmente, `import qualified`), seguida de la lista de nombres de las entidades importadas, escritos entre paréntesis y separados por comas. Esta lista es opcional. *Si no aparece, se sobreentiende que se importan todas las entidades exportadas por el módulo exportador.*

- Las *declaraciones de importación cualificada* (que se reconocen por la sintaxis `import qualified`) obligan a que todos los usos de las entidades exportadas dentro del módulo importador se nombren con la sintaxis *Nexp.ide*, siendo *Nexp* el nombre del módulo exportador e *ide* el identificador de la entidad exportada dentro del módulo exportador. Este recurso sirve para evitar conflictos de identificadores en el caso de que un módulo haga varias importaciones de entidades que tengan el mismo nombre en los diferentes módulos que las exportan.

- Las declaraciones de importación y las declaraciones de operadores (si las hay) deben preceder al resto de las declaraciones y definiciones.

- Entre las declaraciones y definiciones de un módulo, pueden aparecer definiciones de entidades que no se hayan declarado como exportadas en la cabecera del módulo. Estas *entidades privadas* son útiles en muchos casos como recurso auxiliar para la definición de las entidades exportadas.

Ejemplo. A continuación, mostramos el esquema incompleto de un módulo de implementación para el TAD `Fecha`. Incluimos la especificación del TAD

en forma de comentario, y omitimos la implementación de algunas operaciones. En el resto de este curso, seguiremos un esquema similar para presentar otros TADs.

```
--                      Las fechas como TAD


   module Fecha (Fecha, fechaLegal, creaFecha, dia, mes, anno,
                 distancia, desplaza, diaSemana) where


--     Importacion de modulos que definen tipos auxiliares
--                  (no necesariamente TADs).

   import DiaSemana
   import Dia
   import Mes
   import Anno


-- TAD Fecha: sus datos representan fechas de nuestro calendario.

--                     Tipo representante:

   newtype Fecha = F (Dia, Mes, Anno)

--              Invariante de la representacion:

-- valida               :: Fecha -> Bool
-- valida (F (d, m, a))  = 1 <= d & d <= n
--                        where n = numero de dias del mes m del anno a

--                  Equivalencia abstracta:

-- equiv                                :: Fecha -> Fecha -> Bool
-- equiv (F (d, m, a)) (F (d', m', a'))  = (d, m, a) == (d', m', a')

--           Operaciones publicas del TAD Fecha:

   fechaLegal :: Dia -> Mes -> Anno -> Bool
-- Reconoce si un dia, mes y anno dados constituyen una fecha legal.
-- Falta completar la definicion.

   creaFecha      :: Dia -> Mes -> Anno -> Fecha
-- Crea una fecha a partir de un dia, mes y anno dados.
-- La operacion solo esta definida si se trata de una fecha legal.

   creaFecha d m a = if fechaLegal d m a
                     then F (d, m, a)
```

```
                        else error "Fecha ilegal"

   dia :: Fecha -> Dia
-- Consulta el dia de una fecha dada.

   dia (F (d, m, a)) = d

   mes :: Fecha -> Mes
-- Consulta el mes de una fecha dada.

   mes (F (d, m, a)) = m

   anno :: Fecha -> Anno
-- Consulta el anno de una fecha dada.

   anno (F (d, m, a)) = a

   distancia :: Fecha -> Fecha -> Int
-- Calcula la distancia (medida como numero de dias) entre dos fechas dadas.
-- Falta completar la definicion.

   desplaza :: Fecha -> Int -> Fecha
-- Calcula el resultado de desplazar una fecha un cierto numero de dias.
-- Falta completar la definicion.

   diaSemana :: Fecha -> DiaSemana
-- Calcula el dia de la semana en que cae una fecha dada.
-- Falta completar la definicion.
```

En este punto conviene hacer algunos comentarios para entender el diseño del módulo `Fecha`. En primer lugar, observemos que el del tipo representante se ha definido como `newtype`. Alternativamente, podríamos haber definido el tipo representante de `Fecha` como un tipo construido:

```
data Fecha = F (Dia, Mes, Anno)
```

La única diferencia entre las dos opciones es que en el caso de `newtype` el sistema descarta la constructora `F` en tiempo de ejecución, lo cual supone una ligera ganacia de eficiencia. Recuérdese lo estudiado acerca de `data` y `newtype` en la sección 2.6. En cualquier caso, es muy importnate observar que *el módulo* `Fecha` *no exporta la constructora* `F`. Gracias a ésto, el tipo representante queda encapsulado en la implementación. Los usuarios de `Fecha` no podrán usar patrones de la forma `F (d, m, a)` como representación de fechas, y estarán obligados a procesar datos de tipo `Fecha` basándose exclusivamente en las operaciones públicas del TAD, que corresponden precisamente a las funciones exportadas por el módulo.

También vale la pena observar que las definiciones del *invariante de la representación* y la *equivalencia abstracta* se han incluido como comen-

tarios dentro del módulo. Es aconsejable incluir esta información dentro del módulo de implementación de cualquier TAD. Aunque *no se pretende que* `valido` *y* `equiv` *sean en todos los casos funciones ejecutables*, sus definiciones se pueden presentar utilizando una notación parecida a la que se usa para las definiciones de funciones en los programas Haskell, como hemos hecho en el ejemplo de las fechas.

En el resto del curso tendremos ocasión de estudiar varios TADs típicos. Para lograr el encapsulamiento del tipo representante dentro de los correspondientes módulos de implementación, aplicaremos la misma técnica del ejemplo `Fecha`: definir el tipo representante como un `newtype`, y dejar sus constructoras sin exportar. *Nunca definiremos el tipo representante de un TAD como sinónimo de otro tipo*, ya que ésto no garantizaría el encapsulamiento. Por ejemplo, en el módulo de implementación del TAD `Fecha` nuestra idea es representar una fecha como una tupla. Sin embargo, sería inapropiado utilizar la definición

```
type Fecha = (Dia, Mes, Anno)
```

ya que de este modo `Fecha` sería sinónimo del tipo producto `(Dia, Mes, Anno)`, y la representación de las fechas como tuplas sería accesible a los usuarios del TAD. En general, cualquier programa usuario de un TAD deberá importar el módulo que exporte al TAD junto con sus operaciones. Al no exportarse constructoras, las funciones que utilicen un TAD se tendrán que definir sin utilizar patrones (excepto variables).

Ejercicios

1. Completa las definiciones que faltan en el módulo de implementación del TAD `Fecha` que sigue la idea de representar una fecha como una terna formada por un día, un mes y un año. Las soluciones de algunos de los ejercicios de la sección 2.6 te pueden ser útiles.

2. Plantea otro módulo de implementación del TAD `Fecha`, basado en la idea de representar una fecha como un número entero (el número de días transcurridos desde el comienzo de nuestra era hasta la fecha en cuestión). ¿Cómo se deben formular, en este caso, el invariante de la representación y la equivalencia abstracta? ¿Qué operaciones del TAD resultan más fáciles de implementar con esta representación, y por qué?

3. Especifica un TAD parametrizado `(Cjto a)` dotado de las siguientes operaciones públicas:

```
vacio    :: Cjto a
pon      :: Cjto a -> a -> Cjto a
```

```
quita    :: Cjto a -> a -> Cjto a
miembro  :: Cjto a -> a -> Bool
esVacio  :: Cjto a -> Bool
```

Estudia dos posibles implementaciones, usando como tipo representante `[a]` y `a -> Bool`, respectivamente. Especifica en cada caso el invariante de la representación y la equivalencia abstracta, y discute las ventajas e inconvenientes de cada representación para la implementación de las operaciones del TAD.

4. Considera un TAD `Listin`, tal que un dato de este tipo represente un listín que contiene los números de teléfono de una serie de personas. El TAD dispone de las siguientes operaciones: `creaListin`, que crea un listín vacío; `apuntaTelefono`, que añade a un listín el número de teléfono de una persona, dando un nuevo listín como resultado; y `consultaTelefono`, que busca en un listín el número de teléfono de una persona y devuelve dicho número como resultado. Diseña un módulo que implemente este TAD, basándote en listas de asociación para definir el tipo representante. Incluye como comentarios la especificación y las definiciones del invariante de la representación y la equivalencia abstracta. Puedes suponer disponibles otros dos módulos que exporten los tipos `Persona` y `Telefono`.

5. En este ejercicio se trata de programar algunos aspectos de la gestión de una biblioteca simplificada. Suponemos que la biblioteca dispone de uno o más ejemplares de cada libro. Un *libro* debe tener *autor* y *título*, mientras que un *ejemplar* debe tener además un *número de registro*. Los ejemplares se registran en la biblioteca al ser adquiridos, y posteriormente pueden ser prestados a *usuarios*. Por consiguiente, la *biblioteca* debe mantener la información de los ejemplares existentes y del *status* de cada uno, que puede consistir en estar disponible o prestado a un determinado usuario. Se pretende reflejar el comportamiento de la biblioteca en un TAD que exporte las siguientes operaciones:

   - `crea :: Biblioteca`
     Devuelve una biblioteca vacía.
   - `registra :: Biblioteca` $\rightarrow$ `Ejemplar` $\rightarrow$ `Biblioteca`
     Devuelve el nuevo estado de una biblioteca, tras incluir un nuevo ejemplar de un libro.
   - `presta :: Biblioteca` $\rightarrow$ `Libro` $\rightarrow$ `Usuario` $\rightarrow$ `Biblioteca`
     Devuelve un nuevo estado de una biblioteca, tras prestar un ejemplar disponible de un libro solicitado por un usuario.
   - `devuelve :: Biblioteca` $\rightarrow$ `Ejemplar` $\rightarrow$ `Biblioteca`
     Devuelve un nuevo estado de una biblioteca, tras la devolución de un ejemplar que estaba prestado.

- `autores :: Biblioteca → [Autor]`
  Devuelve la lista ordenada de todos los autores que aparecen en una biblioteca.
- `titulos :: Biblioteca → Autor → [Titulo]`
  Devuelve la lista ordenada de todos los títulos de un cierto autor que aparecen en una biblioteca.

Diseña una implementación del TAD `Biblioteca`, teniendo en cuenta como afecta la elección del tipo representante al espacio ocupado por la representación y al tiempo de ejecución de las funciones que implementen las operaciones. Previamente, construye definiciones adecuadas para los otros tipos de datos requeridos por el problema, tales como `Autor`, `Titulo`, `Libro`, `NrReg`, `Ejemplar`, `Usuario`, `Status`, etc.

4.2 Pilas, colas y listas ordenadas

Pilas genéricas

En el apéndice A.24 se presenta un módulo Hugs 98 que implementa el TAD de las pilas genéricas, representando las pilas por medio de listas. Desde el punto de vista de la eficiencia, el comportamiento de este TAD es razonable. Es fácil demostrar que la ejecución de cualquiera de las operaciones públicas del TAD `Pila` (excepto la que calcula el contenido de la pila) aplicada a una pila cualquiera, requiere tiempo $\mathcal{O}(1)$.

En el apéndice A.25 se presenta un módulo Hugs 98 que permite ejecutar pruebas interactivas con pilas genéricas de diferentes tipos.

Colas genéricas: una implementación ingenua

En el apéndice A.26 se presenta un módulo Hugs 98 que implementa el TAD de las colas genéricas empleando una representación sencilla, representando una cola como lista. Esta representación, aunque sencilla, es ineficiente. Por ejemplo, la operación que mete un nuevo elemento en una cola de tamaño n requiere tiempo $\mathcal{O}(n)$.

Colas genéricas: una implementación más eficiente

En el apéndice A.27 se presenta un módulo Hugs 98 que implementa el TAD de las colas genéricas empleando una representación más eficiente que la anterior. Se puede demostrar que la ejecución de una serie de n operaciones públicas del TAD `Cola`, comenzando con una cola vacía, requiere tiempo $\mathcal{O}(n)$ en promedio.

En el apéndice A.28 se presenta un módulo Hugs 98 que permite ejecutar pruebas interactivas con colas genéricas de diferentes tipos.

Listas ordenadas

En el apéndice A.29 se presenta un módulo Hugs 98 que implementa el TAD de las listas ordenadas con elementos de un tipo de la clase `Ord` dado como parámetro. Las operaciones públicas exportadas por este TAD solamente permiten a los clientes la construcción y consulta de listas ordenadas.

Para una lista ordenada de tamaño n, los tiempos de ejecución de las operaciones del TAD `Listord` encargadas de inserción y consulta son $\mathcal{O}(n)$ en el caso peor (cuando la lista se explora por completo). Sin embargo, el hecho de que la lista se mantenga ordenada permite que en muchos casos las operaciones se ejecuten más rápidamente, evitando un recorrido completo de la lista.

Listas de búsqueda

En el apéndice A.30 se presenta un módulo Hugs 98 que implementa el TAD de las listas de búsqueda. Estas son listas de asociación formadas por parejas de tipo `(c,v)`. El tipo `c` representa *claves* y debe ser de la clase `Ord`, mientras que el tipo `v` representa *valores* asociados a las claves, y puede ser arbitrario. Las operaciones públicas exportadas por este TAD solamente permiten a los clientes la construcción y consulta de listas de búsqueda que se mantienen ordenadas con respecto a las claves.

Para una lista de búsqueda de tamaño n, los tiempos de ejecución de las operaciones del TAD `Listbus` encargadas de inserción y consulta son $\mathcal{O}(n)$ en el caso peor (cuando la lista se explora por completo). Sin embargo, el hecho de que la lista se mantenga ordenada con respecto a las claves permite que en muchos casos las operaciones se ejecuten más rápidamente, evitando un recorrido completo de la lista.

Ejercicios

1. Desarrolla dos módulos Hugs 98 denominados `ActivaListord.hs` y `ActivaListbus.hs`, que permitan activar listas ordenadas y de búsqueda interactivas. Inspírate en los módulos `ActivaPila.hs` y `ActivaCola.hs` estudiados en esta sección.

2. Desarrolla una implementación del TAD `(ColaDoble a)` de las colas dobles de elementos del tipo `a` dado como parámetro, exportando operaciones adecuadas para el acceso a una cola doble por los dos extremos. Escribe también un módulo `ActivaColaDoble.hs` que permita activar colas dobles interactivas.

3. Desarrolla una implementación del TAD `(ColaPrio a)` de las colas de prioridad formadas por elementos del tipo `a` dado como parámetro, utilizando números enteros como prioridades. Las operaciones públicas de este TAD deben ser similares a las de las colas ordinarias, pero su comportamiento debe ser tal que los elementos de la cola se atiendan por orden de prioridad, y en caso de igual prioridad, por orden de llegada. Escribe también un módulo `ActivaColaPrio.hs` que permita activar colas de prioridad interactivas.

4. Desarrolla una implementación de un TAD `(Secuencia a)`, cuyos valores se comporten como listas con un punto de interés. Las operaciones públicas deben permitir construir una secuencia vacía, consultar o borrar el elemento situado en la posición del punto de interés, insertar un nuevo elemento en dicha posición, reconocer si el punto de interés se encuentra situado al final de la secuencia, y resituar el punto de interés al comienzo de la secuencia. Escribe también un módulo `ActivaSecuencia.hs` que permita activar colas de prioridad interactivas.

5. Reconsidera el TAD `(Cjto a)` planteado en los ejercicios de la sección 4.1. Estudia como implementarlo aprovechando el TAD de las listas ordenadas para la definición del tipo representante.

6. Especifica un TAD parametrizado `(MultiCjto a)` dotado de las siguientes operaciones públicas:

```
vacio          :: MultiCjto a
pon            :: MultiCjto a -> a -> MultiCjto a
quita          :: MultiCjto a -> a -> MultiCjto a
multiplicidad  :: MultiCjto a -> a -> Int
esVacio        :: MultiCjto a -> Bool
```

   Desarrolla una implementación de este TAD utilizando como tipo representante listas de búsqueda con claves de tipo `a` y valores asociados de tipo `Int`, que representarán las multiplicidades. Los tipos de las operaciones deberán restringirse con la cualificación `(Ord a)`.

7. Suponemos declarados los tipos `Coef` y `Exp` como sinónimos de `Integer`. Especifica un TAD `Poli` cuyos datos representan polinomios en una indeterminada con coeficientes enteros, dotado de las siguientes operaciones públicas:

```
nulo          :: Poli
monomio       :: Coef -> Exp -> Poli
suma          :: Poli -> Poli -> Poli
producto      :: Poli -> Poli -> Poli
```

```
esNulo        :: Poli -> Bool
esMonomio     :: Poli -> Bool
coeficiente   :: Poli -> Exp -> Coef
```

Desarrolla una implementación de este TAD utilizando como tipo representante listas de búsqueda con claves de tipo `Exp` y valores asociados de tipo `Coef`.

8. Reconsidera los TADs `Listin` y `Biblioteca` planteados en los ejercicios de la sección 4.1. Estudia como implementarlos aprovechando en cada caso el TAD de las listas de búsqueda para la definición del tipo representante.

4.3 Vectores

En esta sección se ofrece una presentación de los recursos disponibles para el manejo de vectores (*arrays*) en Haskell. Para más detalles, el lector puede consultar la sección 13 de [5] y las secciones 2.7 y 4.3 de [9].

Para el desarrollo práctico de programas que usen vectores, hay que tener en cuenta que Haskell trata a los vectores con índices de tipo `a` y valores almacenados de tipo `b` como un TAD denominado `(Array a b)`. El módulo de biblioteca `Array.hs` realiza este TAD empleando una representación eficiente dependiente de la implementación, y lo exporta junto con una serie de operaciones públicas descritas más abajo. Estas permiten a los clientes la construcción, modificación y consulta de vectores.

Indices: la clase de tipos `Ix`

`Ix` es una subclase de `Ord`. Sus ejemplares son aquellos tipos cuyos valores se pueden utilizar como índices de vectores. La declaración de la clase de tipos `Ix` en el preludio de Haskell es como sigue:

```
    class Ord a => Ix a where

    range               :: (a,a) -> [a]
-- (range l u) devuelve el rango de indices comprendidos
--  entre el extremo inferior l y el extremo superior u.

    index               :: (a,a) -> a -> Int
-- (index (l,u) i) asocia a un indice i perteneciente a
-- (range l u) un indice entero mayor o igual que 0.

    inRange             :: (a,a) -> a -> Bool
-- (inRange (l,u) i) decide si i pertenece a (range (l,u)).
```

```
    rangeSize            :: (a,a) -> Int
-- (rangeSize (l,u)) calcula la longitud de (range (l,u))

   rangeSize r@(l,u)
             | l > u      = 0
             | otherwise  = index r u + 1

-- Conjunto suficiente de metodos: range, index, inRange
```

Los ejemplares de la clase `Ix` incluyen los tipos `Int`, `Integer` y `Char`. Las correspondientes declaraciones de ejemplar incluidas en el preludio son como sigue:

```
instance Ix Int where
range (m,n)          = [m..n]
index b@(m,n) i
       | inRange b i = i - m
       | otherwise   = error "index: Index out of range"
inRange (m,n) i      = m <= i && i <= n

instance Ix Integer where
range (m,n)          = [m..n]
index b@(m,n) i
       | inRange b i = fromInteger (i - m)
       | otherwise   = error "index: Index out of range"
inRange (m,n) i      = m <= i && i <= n

instance Ix Char where
range (c,c')      = [c..c']
index b@(c,c') ci
   | inRange b ci = fromEnum ci - fromEnum c
   | otherwise    = error "Ix.index: Index out of range."
inRange (c,c') ci = fromEnum c <= i && i <= fromEnum c'
                    where i = fromEnum ci
```

Generalmente, todos los tipos enumerados se declaran como ejemplares de `Ix`. Además, Haskell entiende automáticamente que el producto `(a,b)` de dos tipos de la clase `Ix` es también ejemplar de `Ix`, lo cual permite utilizar parejas de índices como índices. Por ejemplo, la siguiente prueba en `Hugs 98` muestra el cálculo del rango de todos los índices de tipo `(Int,Int)` comprendidos entre el extremo inferior `(1,1)` y el extremo superior `(2,3)`:

```
Prelude> range ((1,1),(2,3))
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

El resultado anterior indica que los índices del rango comprendido entre `(1,1)` y `(2,3)` corresponden a los índices posibles para matrices de dimensión 2×3. Un comportamiento análogo se tiene para productos de cualquier

número de ejemplares de la clase `Ix`, que son siempre ejemplares de `Ix`.

Funciones de procesamiento de vectores

A continuación describimos las principales funciones de procesamiento de vectores exportadas por el módulo `Array.hs`, e ilustramos su uso mediante ejemplos sencillos. En el módulo Hugs 98 del apéndice A.31 se encuentran las definiciones de todas las funciones utilizadas en los ejemplos que siguen. Se recomienda utilizarlo para ejecutar pruebas durante el estudio de este apartado.

Creación de vectores

Para la creación de vectores, `Array.hs` exporta tres funciones:

```
array       :: Ix a => (a,a) -> [(a,b)] -> Array a b
listArray   :: Ix a => (a,a) -> [b] -> Array a b
accumArray  :: Ix a => (b -> c -> b) -> b -> (a,a) ->
                       [(a,c)] -> Array a b
```

De entre las tres operaciones anteriores, `array` es la más básica. La evaluación de `(array limites listAsoc)` devuelve un vector cuyo rango de índices es el indicado por `limites` y cuyo contenido viene descrito por la lista de asociación `listAsoc`. La función `cuadrados` definida a continuación sirve para crear un vector que contiene los cuadrados de todos los enteros comprendidos entre dos límites dados:

```
cuadrados       :: (Int,Int) -> Array Int Int
cuadrados (l,u)  = array (l,u) [(i,i*i) | i <- [l..u]]
```

La siguiente prueba en Hugs 98 ilustra el uso de `cuadrados`:

```
PruebaArray> cuadrados (2,8)
array (2,9) [(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]
```

Como vemos, un valor de tipo `Array` se representa indicando una llamada a la función `array` adecuada para crearlo. Esta sintaxis no revela ningún detalle de la representación interna empleada, la cual puede variar dependiendo de la implementación de Haskell utilizada.

La función `listArray` es una variante simple de `array`. Con ella se crea un vector a partir de una llamada de la forma `(listArray limites listElem)`. Ahora, el segundo parámetro `listElem` representa el contenido del vector como lista de elementos, no como lista de asociación entre índices y elementos. La función `cuadrados'` mostrada a continuación es equivalente a `cuadrados`, pero está definida usando `listArray` en lugar de `array`:

```
cuadrados'       :: (Int,Int) -> Array Int Int
cuadrados' (l,u)  = listArray (l,u) [i*i | i <- [l..u]]

PruebaArray> cuadrados' (2,8)
array (2,9) [(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]
```

El uso de las dos funciones `array` y `listArray` está sujeto a ciertas restricciones naturales:

- En una llamada `(array limites listAsoc)` se exige que `listAsoc` contenga *exactamente una pareja* `(i,x)` para cada índice `i` perteneciente al rango de índices `(range limites)`. En otro caso se produce un error, como muestran los siguientes ejemplos de ejecución:

  ```
  PruebaArray> array (2,4) [(2,4),(4,16)]
  array (2,4) [(2,4),(3,
  Program error: {_undefined_array_element}

  PruebaArray> array (2,4) [(2,4),(3,9),(3,27),(4,16)]
  array (2,4) [(2,4),(3,
  Program error: {_undefined_array_element}
  ```

- En una llamada `(listArray limites listElem)` se exige que `listElem` tenga *al menos tantos elementos* como `(range limites)`. Si el número de elementos es menor, se produce un error; y si es mayor, los elementos sobrantes se descartan, como muestran los siguientes ejemplos de ejecución:

  ```
  PruebaArray> listArray (2,4) [4,9]
  array (2,4) [(2,4),(3,9),(4,
  Program error: {_undefined_array_element}

  PruebaArray> listArray (2,4) [4,9,16,25,36]
  array (2,4) [(2,4),(3,9),(4,16)]
  ```

La función `accumArray` permite crear vectores de manera más flexible, evitando las restricciones anteriores. El comportamiento de `accumArray` es equivalente al expresado por la ecuación siguiente:

```
accumArray f e limites listAsoc
  = array limites listAsoc'
    where listAsoc' = [(i, foldl f e (assocs i)) |
                          i <- range limites        ]
          assocs i  = [x | (i,x) <- listAsoc]
```

La idea es que una llamada (`accumArray f e limites listAsoc`) crea un vector de tal manera que la serie de todos los valores asociados por `listAsoc` a cada índice `i` de (`range limites`) se acumula a un único valor, calculado con (`foldl f e`). Un ejemplo típico de `accumArray` es la función definida a continuación, que sirve para calcular histogramas:

```
histograma                :: (Ix a, Num b) =>
                             (a,a) -> [a] -> Array a b
histograma limites indices  = accumArray (+) 0
                                         limites
                                         [(i,1) | i <- indices,
                                                  inRange limites i]
```

Obsérvese que (`histograma limites indices`) representa una tabla en la cual cada índice perteneciente al rango de `limites` tiene asociado su número de apariciones como miembro de la lista `indices`. Por ejemplo:

```
PruebaArray> histograma (3,5) [i | i <- [1..7], j <- [3..i]]
array (3,5) [(3,1),(4,2),(5,3)]
```

Consulta de vectores

Para la consultade vectores, `Array.hs` exporta las funciones siguientes:

```
(!)      :: Ix a => Array a b -> a -> b
bounds   :: Ix a => Array a b -> (a,a)
indices  :: Ix a => Array a b -> [a]
elems    :: Ix a => Array a b -> [b]
assocs   :: Ix a => Array a b -> [(a,b)]
```

La primera de ellas, representada por el operador infijo (`!`), sirve para consultar el elemento almacenado en la posición de un índice dado de un vector dado. Por ejemplo:

```
PruebaArray> cuadrados (2,8) ! 5
25
```

Las otras cuatro fuciones sirven para realiazar consultas de carácter más global en un vector dado, obteniendo respectivamente los índices extremos, la lista de todos los índices, la lista de todos los elementos almacenados, y la lista de asociación entre índices y elementos. Volviendo al ejemplo del vector de cuadrados, se tiene:

```
PruebaArray> bounds (cuadrados (2,8))
(2,8)
PruebaArray> indices (cuadrados (2,8))
[2,3,4,5,6,7,8]
```

```
PruebaArray> elems (cuadrados (2,8))
[4,9,16,25,36,49,64]
PruebaArray> assocs (cuadrados (2,8))
[(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]
```

Modificación de vectores

Puesto que Haskell es un lenguaje funcional puro, no existen operaciones de modificación de vectores que produzcan el efecto de asignaciones destructivas. Sin embargo, sí existen funciones que pueden modificar un vector `v` dado como parámetro y devolver el vector modificado `v'` como resultado, *sin destruir el vector original* `v`. Concretamente, `Array.hs` exporta las siguientes funciones de modificación de vectores:

```
(//)  :: Ix a => Array a b -> [(a,b)] -> Array a b
accum :: Ix a => (b -> c -> b) -> Array a b -> [(a,c)] -> Array a b
ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c -> Array a c
```

Una llamada de la forma `(v // listAsoc)` calcula la modificación de `v` consistente en actualizar los elementos ascociados a algunos índices, según indique la lista de asociación `listAsoc`. Si `listAsoc` es vacía, no se produce ninguna modificación; si es unitaria, la modificación consiste en la actualización del elemento asociado a un único índice; y si en `listAsoc` aparecen varias asociaciones diferentes para un mismo índice, se produce un error de ejecución. Por ejemplo:

```
PruebaArray> cuadrados (2,8) // []
array (2,8) [(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]

PruebaArray> cuadrados (2,8) // [(5,27)]
array (2,8) [(2,4),(3,9),(4,16),(5,27),(6,36),(7,49),(8,64)]

PruebaArray> cuadrados (2,8) // [(5,27),(7,50)]
array (2,8) [(2,4),(3,9),(4,16),(5,27),(6,36),(7,50),(8,64)]

PruebaArray> cuadrados (2,8) // [(5,27),(5,28)]
array (2,8) [(2,4),(3,9),(4,16),(5,
Program error: {_undefined_array_element}
```

En ocasiones puede interesar actualizar un vector con respecto a una lista de asociación `listAsoc` que contenga varias asociaciones diferentes para un mismo índice. Esta modificación se puede realizar por medio de la función `accum`, cuyos parámetros `f` y `e` sirven para acumular todos las elementos asociados por `listAsoc` a un mismo índice `i`. Más exactamente, el comportamiento de `assoc` es equivalente al definido por la ecuación siguiente:

```
accum  f v listAsoc  =  v // listAsoc'
```

```
        where listAsoc' = [(i, foldl f inicial (assocs i)) |
                           i <- range limites         ]
              inicial   = v ! i
              assocs i  = [x | (i,x) <- listAsoc]
```

La diferencia de comportamiento con respecto a (`//`) se observa claramente al modificar el vector (`cuadrados (2,8)`) utilizando `accum`:

```
PruebaArray> accum (+) (cuadrados (2,8)) [(5,27),(5,28)]
array (2,8) [(2,4),(3,9),(4,16),(5,80),(6,36),(7,49),(8,64)]
```

Tanto (`//`) como `accum` requieren que todos los índices de la lista de asociación con respecto a la cual se desea actualizar el vector pertenezcan al rango de índices de este; en otro caso se producen errores de ejecución:

```
PruebaArray> cuadrados (2,8) // [(9,81)]
array
Program error: index: Index out of range

PruebaArray> accum (+) (cuadrados (2,8)) [(5,27),(5,28),(9,81)]
array
Program error: index: Index out of range
```

La función de modificación `ixmap` sirve para renombrar los índices de un vector. Más exactamente, (`ixmap nuevosLimites viejoInd v`) calcula un vector `v'` cuyos índices extremos corresponden a `nuevosLimites`, y cumpliendo que `v' ! i = v ! viejoInd i` para cada índice `i` perteneciente a (`range nuevosLimites`). La siguiente prueba de ecución muestra el resultado de modificar el vector (`cuadrados (2,8)`) trasladando sus índices a los nuevos límites (`0,6`):

```
PruebaArray> ixmap (0,6) (+2) (cuadrados (2,8))
array (0,6) [(0,4),(1,9),(2,16),(3,25),(4,36),(5,49),(6,64)]
```

Un ejemplo: producto de matrices

El módulo del apéndice A.31 incluye varias funciones que ilustran un ejemplo típico de programación con vectores en Haskell: el producto de matrices. Este ejemplo se ha tomado de la sección 13.5 de [5]. La función principal

```
prodMat    :: (Ix a, Ix b, Ix c, Num d) =>
              Array (a,b) d -> Array (b,c) d -> Array (a,c) d
```

es genérica con respecto a los tipos `a, b, c` de los índices (que pueden ser ejemplares cualesquiera de la clase `Ix`) y al tipo `d` de los elementos, que puede ser cualquier ejemplar de la clase `Num`. Se tienen dos restricciones naturales:

el tipo `d` de los elementos debe ser el mismo en las dos matrices, y el tipo `b` de los índices de columna de la primera matriz debe ser el mismo que el tipo de los índices de fila de la segunda matriz. Además, `prodMat` aborta el cómputo con un mensaje de error en el caso de que las dimensiones de las dos matrices no ajusten del modo debido para que sea posible multiplicarlas.

La definición de `prodMat` utiliza la primitiva `array` de creación de vectores, y una lista intensional para expresar la asociación entre cada índice `(i,j)` de la matriz producto y el correspondiente elemento, calculado como el producto escalar de la fila `i` de la primera matriz por la columna `j` de la segunda matriz.

En A.31 se define también otra función `prodMat'` equivalente a la anterior, pero utilizando la primitiva `accumArray` en lugar de `array` para crear la matriz producto. Esta definición alternativa se asemeja más a la codificación habitual del producto de matrices en los lenguajes de programación imperativos.

Finalmente, A.31 exporta una función llamada `prodMatGen` que generaliza a `prodMat` reemplazando las operaciones `sum` y `(*)` por dos funciones `sumList` y `prod` pasadas como parámetros. La función `prodMat` es verdaderamente un caso particular de `prodMatGen`, ya que se cumple:

```
prodMat = prodMatGen sum (*)
```

En A.31 se han incluido algunos comentarios que indican como realizar pruebas de ejecución con las diferentes funciones de producto de matrices.

Ejercicios

1. Define una función lo más genérica posible, que calcule el producto de un vector `v` por un elemento `x` del mismo tipo que los elementos de `v`. Se supone que `v` y `x` vendrán dados como parámetros de la función.

2. Define una función lo más genérica posible que calcule la suma de dos vectores `u` y `v` del mismo tipo y con el mismo rango de índices, dados como parámetros.

3. Recuerda que el *producto escalar* de dos vectores `u` y `v` con el mismo rango de índices y elementos de tipo numérico, es el número obtenido como suma de todos los productos `(u!i)*(v!i)`, con `i` variando sobre todos los índices posibles. Define una función lo más genérica posible que calcule el producto escalar de dos vectores del mismo tipo, dados como parámetros.

4. Define una función lo más genérica posible que reciba como parámetros una matriz `a` y dos índices `i, j`, y calcule la matriz resultante de intercambiar las filas `i` y `j` de `a`. Los parámetros `i, j` deberán ser

tales que la pareja `(i,j)` pertenezca al rango de índices de `a`; en caso contrario, la función quedará indefinida.

5. Define una función similar a la del ejercico anterior, que sirva para clacular la matriz resultante de intercambiar dos columnas de una matriz dada.

6. Recuerda que la *traspuesta* de una matriz cuadrada `a` es otra matriz cuadrada `b` con el mismo rango de índices, tal que `a(i,j) = b(j,i)` para todas las parejas de índices `(i,j)` adecuadas al rango de índices de `a`. Define una función lo más genérica posible que calcule la traspuesta de una matriz cuadrada arbitraria dada como parámetro.

4.4 Arboles binarios y generales

A lo largo de esta sección vamos a estudiar diferentes tipos de árboles, junto con funciones encargadas de realizar diferentes algoritmos de tratamiento de los mismos, y algunas aplicaciones.

Para cada tipo de árbol, presentaremos un módulo Hugs 98 encargado de definir y exportar dicho tipo, junto con algunas funciones básicas útiles para procesar árboles de ese tipo.

Los tipos estudiados en esta sección *no están diseñados como TADS*. Por consiguiente, los módulos que los implementan exportan también las constructoras.

Arboles binarios con información en los nodos internos

El módulo Hugs 98 del apéndice A.32 define y exporta el tipo genérico `(Arbin a)` de los árboles binarios que almacenan informaciones de tipo `a` en sus nodos internos, junto con algunas funciones que realizan operaciones básicas de procesamiento de árboles binarios. Para ejecutar algunas pruebas sencillas del comportamiento de dichas operaciones, se recomienda cargar el módulo del apéndice A.33.

Arboles binarios con información en las hojas

El módulo Hugs 98 del apéndice A.34 define y exporta el tipo genérico `(ArbinH a)` de los árboles binarios que almacenan informaciones de tipo `a` en sus hojas, y ninguna información en los nodos internos. El módulo exporta además algunas funciones que realizan operaciones básicas de procesamiento de estos árboles binarios. Para ejecutar algunas pruebas sencillas del comportamiento de dichas operaciones, se recomienda cargar el módulo del apéndice A.35.

Arboles binarios calibrados con información en las hojas

El módulo Hugs 98 del apéndice A.36 define y exporta el tipo genérico `(ArbinHC a)`, una variante de `(ArbinH a)` diseñada de manera que cada nodo del árbol (tanto los nodos internos como las hojas) almacena un *peso* de tipo `Int`. Además, las hojas siguen almacenando informaciones de tipo `a`. El módulo exporta también operaciones básicas para el procesamiento de este tipo de árboles. Para ejecutar algunas pruebas sencillas del comportamiento de dichas operaciones, se recomienda cargar el módulo del apéndice A.37.

Una aplicación: códigos de Huffman

Un ejemplo característico de uso de árboles binarios con datos almacenados en las hojas es el uso de árboles de tipo `(ArbinH Char)` (llamdos *árboles de codificación*) para representar códigos binarios de longitud variable con la propiedad de prefijo. Esta propiedad significa que los códigos de dos caracteres diferentes deben ser dos palabras binarias tales que ninguna de las dos sea prefijo de la otra. Así se garantiza que la decodificación es unívoca.

El interés práctico de esta clase de códigos consiste en que a partir de un texto dado `txt` es posible construir un *árbol de codificación óptimo*, en el sentido de que la longitud media de los códigos de los caracteres del texto (calculada teniendo en cuenta la frecuencia con la que cada caracter aparece en `txt`) es lo menor posible. De esta manera, el tamaño del código binario del texto también se minimiza.

El módulo Hugs 98 del apéndice A.38 define y exporta funciones que sirven para construir árboles de codificiación óptimos, siguieindo un algoritmo debido a *Huffman*. El módulo también exporta funciones que realizan las tareas de codificación y decodificación de textos, y algunos ejemplos para ejecutar pruebas.

Arboles generales y bosques

El módulo Hugs 98 del apéndice A.39 define y exporta el tipo genérico `(Arbol a)`, que representa árboles generales con informaciones de tipo `a` en sus nodos. En este tipo de árboles, cada nodo almacena un valor de tipo `a` y tiene un número arbitario $n \geq 0$ de hijos, que son árboles del mismo tipo. Los árboles más pequeños de este tipo se reducen a un solo nodo con una información de tipo `a` y sin ningún hijo; no existen árboles generales vacíos.

El módulo exporta también operaciones básicas para el procesamiento de árboles generales. Para ejecutar algunas pruebas sencillas del comportamiento de dichas operaciones, se recomienda cargar el módulo del apéndice A.37.

Ejercicios

1. Recuerda que

```
data ArbinH a = Hoja a | Nodo (ArbinH a) (ArbinH a)
```

   es el tipo de datos de los árboles binarios que almacenan informaciones de tipo `a` en sus hojas.

   (a) Define recursivamente una función `hojas :: ArbinH a -> Int` que calcule el número de hojas de un árbol.

   (b) Define recursivamente una función `nodos :: ArbinH a -> Int` que calcule el número de nodos internos de un árbol.

   (c) Demuestra por inducción que cualquier árbol finito y terminado `xa :: ArbinH a` verifica: `hojas xa = 1 + nodos xa`.

   (d) Define recursivamente una función `talla :: ArbinH a -> Int` que calcule la *talla* de un árbol, entendida como la distancia entre la raíz y la hoja más profunda.

   (e) Demuestra por inducción que cualquier árbol finito y terminado `xa :: ArbinH a` verifica: `talla xa` $<$ `hojas xa` $\leq 2^{\texttt{talla xa}}$.

2. Define recursivamente una función `sub :: ArbinH a -> [ArbinH a]`, que haga corresponder a un árbol binario la lista formada por todos sus subárboles. Enuncia y demuestra un resultado que relacione el número de subárboles de un árbol de este tipo con su número de hojas.

3. Define recursivamente una función `profs :: ArbinH a -> ArbinH Int`, de modo que `profs xa` sea el resultado de reemplazar la información de cada hoja de `xa` por su profundidad (distancia a la raíz).

4. Define una función `arbinH :: [a] -> ArbinH a` tal que `arbinH xs` sea un árbol `xa` de talla mínima con la propiedad de que `front xa = xs`.

5. Define dos funciones de orden superior

```
mapArbinH :: (a -> b) -> ArbinH a -> ArbinH b
foldArbinH :: (a -> b) -> (b -> b -> b) -> ArbinH a -> b
```

   cuyo comportamiento para árboles de tipo `(ArbinH a)` sea análogo al de `map` y `foldr` para listas de tipo `[a]`.

6. Utilizando `foldArbinH` y sin usar recursión explícita, construye definiciones de las funciones indicadas en los apartados siguientes:

(a) Las funciones que claculan el número de nodos, el número de hojas y la talla de un árbol de tipo `(ArbinH a)`.

(b) Una función que calcule el máximo de los valores almacenados en las hojas de un árbol de tipo `(ArbinH a)`, suponiendo `Ord a`.

(c) La función `mapArbinH`.

7. Demuestra que las funciones `mapArbinH` y `foldArbinH` verifican las siguientes propiedades:

```
mapArbinH id        =  id
mapArbinH (f . g)   =  mapArbinH f . mapArbinH g   (mapArbinH es funtor)
map f . front       =  front . mapArbinH f
foldArbinH id (⊕)   =  foldr1 (⊕) . front          si (⊕) es asociativa
foldArbinH id (⊕)   =  foldl1 (⊕) . front          si (⊕) es asociativa
```

8. Recuerda que

```
data Arbol a = Planta a [Arbol a]
```

es el tipo de datos de los árboles generales, que almacenan informaciones de tipo `a` en sus nodos. Define funciones recursivas que calculen el número de nodos y la talla de un árbol general.

9. Define dos funciones de orden superior

```
mapArbol :: (a -> b) -> Arbol a -> Arbol b
foldArbol :: (a -> [b] -> b) -> Arbol a -> b
```

cuyo comportamiento para árboles de tipo `(Arbol a)` sea análogo al de `map` y `foldr` para listas de tipo `[a]`.

10. Utilizando `foldArbol` y sin usar recursión explícita, construye definiciones de las funciones indicadas en los apartados siguientes:

(a) Las funciones que claculan el número de nodos y la talla de un árbol de tipo `(Arbol a)`.

(b) Una función que calcule el máximo de los valores almacenados en los nodos de un árbol de tipo `(Arbol a)`, suponiendo `Ord a`.

(c) La función `mapArbol`.

11. Demuestra que `mapArbol` verifica las dos propiedades siguientes:

```
mapArbol id       =  id
mapArbol (f . g)  =  mapArbol f . mapArbol g
```

12. Define dos funciones recursivas

```
bin :: Arbol a -> ArbinH a
gen :: ArbinH a -> Arbol a
```

que sean inversas una de otra. Demuestra que efectivamente son inversas.
OJO: este ejercicio no es trivial. Una solución se puede encontrar en la sección 6.4.1 del texto de Bird [1].

4.5 Arboles ordenados y de búsqueda

En esta sección vamos a estudiar los árboles ordenados y los árboles de búsqueda. Desde el punto de vista de las aplicaciones, su utilidad es comparable a la de las listas ordenadas y de búsqueda ya estudiadas en la sección 4.2, salvo que su eficiencia es mayor: las operaciones encargadas de inserción y consulta en un árbol ordenado o de búsqueda de tamaño n son ahora $\mathcal{O}(log\, n)$ bajo el supuesto de que el árbol esté equilibrado en talla.

Presentaremos estos dos tipos de árboles como TADs, y sus constructoras no serán exportadas. Las operaciones públicas disponibles los clientes solamente permitirán la construcción de árboles ordenados, lo cual no sería cierto si se exportasen las constructoras.

Las operaciones de inserción y borrado de los árboles ordenados y de búsqueda de esta sección, no garantizan que los árboles se mantengan equilibrados en talla. Esto se puede conseguir desarrollando un tipo más refinado de árboles ordenados y de búsqueda, conocidos como árboles AVL. La implementación de árboles AVL en estilo funcional se puede consultar en el texto de Rabhi y Lapalme [9].

Arboles ordenados

El módulo Hugs 98 del apéndice A.41 exporta el TAD `(Arbord a)` de los árboles binarios ordenados que almacenan informaciones de tipo `a`, junto con operaciones públicas que permiten la creación, modificación y consulta de este tipo de árboles. El tipo `a` de las informaciones almacenadas debe ser un ejemplar de la clase `Ord`.

Para ejecutar algunas pruebas sencillas con árboles ordenados se puede cargar el módulo A.42. Para realizar pruebas con mayor comodidad convendría desarrollar un módulo que permitiese el manejo interactivo de árboles ordenados, tal como se propone en uno de los ejercicios de esta sección.

Arboles de búsqueda

El módulo Hugs 98 del apéndice A.43 exporta el TAD `(Arbus c v)` de los árboles binarios de búsqueda que almacenan valores de tipo `v` asociados a

claves de tipo `c`, junto con operaciones públicas que permmiten la creación, modificación y consulta de este tipo de árboles. El tipo `c` de las claves almacenadas debe ser un ejemplar de la clase `Ord`.

Para ejecutar algunas prueas sencillas con árboles de búsqueda se puede cargar el módulo A.44. Para realizar pruebas con mayor comodidad convendría desarrollar un módulo que permitiese el manejo interactivo de árboles de búsqueda, tal como se propone en uno de los ejercicios de esta sección.

Una aplicación: diccionario interactivo

Como aplicación representativa de los árboles de búsqueda, el módulo Hugs 98 presentado en el apéndice A.45 ofrece a sus clientes un diccionario inglés-castellano que se puede usar interactivamente. La función principal exportada por este módulo es el proceso `diccionario :: IO ()`, que inicia una sesión interactiva de uso del diccionario. La interacción se basa en comandos tecleados por el usuario e interpretados por el proceso interactivo, cuyo efecto es efectuar consultas y/o modificaciones del diccionario.

Internamente, el diccionario se representa como un árbol de búsqueda de tipo `(Arbus String String)`, donde las claves se interpretan como textos en inglés y los valores asociados se interpretan como traducciones al castellano. Al comienzo de cada sesión interactiva, el diccionario se lee de un archivo de texto. Cada vez que termina una sesión, el diccionario (posiblemente modificado) se vuelve a escribir en el mismo archivo. Entre las funciones exportadas por el módulo A.45 se encuentran los procesos parametrizados `recuperaDic` y `guardaDic`, encargados de realizar estas tareas de lectura y escritura.

Se deja como ejercicio el estudio detallado del código de este módulo, así como la ejecución de las pruebas sugeridas como comentarios dentro del texto del mismo.

Ejercicios

1. Desarrolla dos módulos Hugs 98 denominados `ActivaArbord.hs` y `ActivaArbus.hs`, que permitan activar árboles ordenados y de búsqueda interactivos. Inspírate en los módulos `ActivaPila.hs` y `ActivaCola.hs` estudiados en la sección 4.2.

2. Programa una función `treeSort :: Ord a => [a] -> [a]` que devuelva la lista resultante de ordenar la lista dada como parámetro por el siguiente método: construir un árbol binario ordenado, mediante inserción reiterada de los elementos de la lista dada, a partir del árbol vacío; y seguidamente, recorrer dicho árbol en inorden.

3. Programa una función `frecuencias :: Ord a => [a] -> [(a,Int)]`, tal que `(frecuencias xs)` devuelva la lista formada por todas las

parejas `(x,n)` tales que `x` aparece exactamente `n` veces como miembro de `xs`, ordenada en orden creciente de valores de `x`. Utiliza un algoritmo que construya primero un árbol de búsqueda de tipo `Arbus a Int` a partir de `xs`, y luego calcule la lista pedida recorriendo el árbol.

4. Desarrolla una implementación del TAD `(Cjto a)` propuesto en el ejercicio 4.2.5, aprovechando el TAD de los árboles ordenados para el tipo representante.

5. Desarrolla una implementación del TAD `(MultiCjto a)` propuesto en el ejercicio 4.2.6, aprovechando el TAD de los árboles de búsqueda para el tipo representante.

6. Desarrolla una implementación del TAD `Poli` propuesto en el ejercicio 4.2.7, aprovechando el TAD de los árboles de búsqueda para el tipo representante.

7. Desarrolla implementaciones de los TADs `Listin` y `Biblioteca` propuestos en los ejercicios de la seccción 4.1, aprovechando el TAD de los árboles de búsqueda para el tipo representante.

8. Desarrolla una extensión del programa que gestiona un diccionario interactivo *inglés-español*, de manera que se añada la funcionalidad de un diccionario *español-inglés*. Deberás modificar convenientemente la estructura que representa el diccionario, así como el menú de órdenes disponibles, y la interacción con el usuario.

4.6 Otros tipos de datos recursivos

Además de los tipos de datos recursivos correspondientes a las estructuras de datos clásicas, los lenguajes funcionales estilo Haskell permiten definir otros tipos de datos recursivos, diseñados de acuerdo con las necesidades de aplicaciones específicas. A modo de ejemplo, en esta sección vamos a estudiar una introducción sencilla a la programación de *procesadores de lenguajes* en estilo funcional.

Un *procesador* para un cierto lenguaje de programación $\mathcal{L}$ es un programa capaz de procesar el texto de cualquier programa $\mathcal{P}$ escrito en el lenguaje $\mathcal{L}$, comprobar si es sintácticamente correcto, y en caso afirmativo, ejecutarlo. Se acostumbra a clasificar los procesadores de lenguajes en dos grupos:

- *Intérpretes*, que ejecutan el programa dado $\mathcal{P}$ interpretándolo directamente desde el lenguaje en el que esté escrito el intérprete.

- *Compiladores*, que ejecutan $\mathcal{P}$ traduciéndolo primero a código de algún otro lenguaje de programación de más bajo nivel, que posteriormente es ejecutado.

Tanto los intérpretes como los compiladores comienzan su tarea con una primera fase denominada *análisis sintáctico* del programa $\mathcal{P}$ dado. Al final de esta fase se ha reconocido si $\mathcal{P}$ es sintácticamente correcto, y en caso afirmativo se ha construido una representación de la *sintaxis abstracta* de $\mathcal{P}$, la cual refleja la estructura sintáctica de $\mathcal{P}$ sin ambigüedades. A partir de la sintaxis abstracta tiene lugar una segunda fase de *interpretación* o de *generación de código*, según se trate de un intérprete o de un compilador.

La programación de analizadores sintácticos en estilo funcional se realiza definiendo funciones de tipo

```
analiza :: String -> SA
```

que reciben como parámetro el texto del programa a analizar y devuelven como resultado su sintaxis abstracta, representada como valor de un tipo de datos recursivo `SA` convenientemente definido. Se conocen muchas técnicas útiles para la programación metódica de analizadores sintácticos en estilo funcional; véase por ejemplo el artículo de Fokker [4].

Nuestra tarea concreta en esta sección va a ser la programación de un intérprete de programas de un lenguaje imperativo sencillo, escrito en Haskell. Para simplificar, omitiremos el desarrollo del analizador sintáctico y nos centraremos en el diseño de los tipos de datos necesarios para representar la sintaxis abstracta, y en la programación de la segunda fase del intérprete. Supondremos un lenguaje fuente muy simple llamado `LOOP`, que solo puede manejar expresiones aritméticas de tipo entero, y cuyos programas se escriben utilizando las construcciones de asignación, distinción de casos estilo `if then else` y bucles estilo `while`.

Intérprete de expresiones aritméticas

El módulo Hugs 98 A.46 define un tipo de datos adecuado para representar la sintaxis abstracta de las expresiones `LOOP` y funciones encargadas de interpretarlas. Este módulo es cliente del módulo del apéndice A.47, el cual define y exporta un TAD adecuado para representar los estados de cómputo de programas `LOOP`, los cuales deben ser procesados por el intérprete. El núcleo del intérprete de expresiones es la función

```
eval :: Exp -> Estado -> Int
```

encargada de calcular el valor de una expresión `LOOP` dada en un estado dado.

Intérprete de programas imperativos

El intérprete se completa en el módulo del apéndice A.48, cliente del módulo A.46. En A.48 se definen el tipo de datos empleado para representar la sintaxis abstracta de programas `LOOP`, y las funciones encargadas de interpretarlas. El núcleo del intérprte de programas es la función

```
run :: Prog -> Estado -> Estado
```

encargada de calcular el estado final resultante de la ejecución de un programa `LOOP` dado en un estado inicial dado.

Variante: Intérprete paso a paso de programas imperativos

El intérprete `run` mencionado en el apartado anterior no suministra información sobre los estados intermedios del programa `LOOP` que se está interpretando. Para obtener un intérprete más flexible y capaz de comunicación interactiva con el usuario, es conveniente reemplazar `run` por otra función de *interpretación paso a paso*, que pueda calcular los estados intermedios por los que pasa la ejecución del programa interpretado. Este problema se resuelve en el módulo Hugs 98 del apéndice A.49. Este módulo podría tomarse como punto de partida para desarrollar un intérprete interactivo de un lenguaje imperativo más realista, diseñado como extensión de `LOOP`.

Ejercicios

1. El tipo de datos recursivo `Exp` definido a continuación está pensado para representar la sintaxis abstracta de expresiones aritméticas muy simplificadas, formadas únicamente a partir de constantes y operadores:

   ```
   data Exp = Cte Int | App Op Exp Exp
            deriving (Eq, Ord, Read, Show)

   data Op  = Add | Sub | Mul | Div | Mod
            deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
   ```

   Adapta el intérprete de expresiones estudiado en esta sección, definiendo dos funciones

   ```
   eval  :: Exp -> Int
   apply :: Op -> Int -> Int -> Int
   ```

   Observa que el tipo `Est` de los estados ya no es necesario, debido a la ausencia de variables.

2. El tipo de datos `Instr` sirve para representar instrucciones encargadas de evaluar expresiones en una máquina virtual que trabaja con una pila de enteros, representada con ayuda del tipo sinónimo `Stack`:

```
data Instr = Load Int | Apply Op
         deriving (Eq, Ord, Read, Show)

type Stack = [Int]
```

   Convenimos en que `xs :: Stack` representa una pila cuya cima es la cabeza de `xs`. Considera las siguientes especificaciones informales:

   (a) Al ejecutar `Load n`, se modifica la pila apilando el entero `n`.

   (b) Al ejecutar `Apply op`, se modifica la pila desapilando los dos enteros más próximos a la cima, aplicándoles el operador `op`, y apilando el resultado.

   Programa dos funciones

```
execute  :: Instr -> Stack -> Stack
run      :: [Instr] -> Stack -> Stack
```

   de modo que `execute` devuelva el nuevo estado de la pila después de ejecutar una instrucción, y `run` devuelva el nuevo estado de la pila después de ejecutar una secuencia de instrucciones.

3. Continuando el ejercicio anterior, programa una función

```
compile :: Expr -> [Instr]
```

   que genere la secuencia de instrucciones que deben ser ejeccutadas por la máquina virtual para evaluar una expresión dada. Razonando por inducción sobre `xs :: Stack`, demuestra que la ecuación

```
run (compile e) xs = eval e : xs
```

   es válida para cualquier `e :: Exp` y cualquier `xs :: Stack` finita y terminada. Particularizando al caso `xs = []`, resulta la *corrección del compilador* `compile` *con respecto al intérprete* `eval`.

   Atención: en el caso de que el razonamiento por inducción requiera el uso de algún *lema auxiliar*, enúncialo claramente y demuéstralo.

4. Reconsidera los dos ejercicios anteriores, utilizando en lugar del tipo sinónimo `Stack` el caso particular `(Pila Int)` del TAD genérico `(Pila a)` estudiado en la sección 4.2.

5. Retomando la idea del ejercicio 2.5.14, define un tipo de datos recursivo `Prop` cuyos valores representen fórmulas de la lógica de proposiciones, y programa las funciones que se especifican informalmente a continuación.

   (a) `fnc :: Prop -> Prop`
   `fnc p` devuelve una forma normal conjuntiva de `p`.

   (b) `fnd :: Prop -> Prop`
   `fnd p` devuelve una forma normal disyuntiva de `p`.

   (c) `sat :: Prop -> Bool`
   `sat p` devuelve `True` si `p` es satisfactible, y `False` en caso contrario.

   (d) `taut :: Prop -> Bool`
   `taut p` devuelve `True` si `p` es una tautología, y `False` en caso contrario.

6. El siguiente tipo sinónimo sirve para representar grafos dirigidos con informaciones de tipo `a` en sus nodos:

```
type Grafo a = a -> [Grafo a]
```

   La idea es que `g :: Grafo a` representa un grafo con nodos de tipo `a`, en el cual `(g x)` es la lista de sucesores inmediatos de cada nodo `x`. Programa una función

```
buscaProf :: Eq a => Grafo a -> (a -> Bool) -> a -> [a]
```

   de modo que `(buscaProf g p x)` busque en profundidad los nodos del grafo `g` que cumplan el predicado `p`, comenzando la búsqueda en el nodo `x`, evitando visitar más de una vez el mismo nodo, y devolviendo la lista de todos los nodos encontrados.

   Sugerencia: utiliza una función auxiliar más general con dos parámetros adicionales `vs, ps :: [a]`, que representen el *conjunto de nodos ya visitados* y la *pila de nodos pendientes de visitar*, respectivamente.

7. Programa ahora otra función

```
buscaNiv :: Eq a => Grafo a -> (a -> Bool) -> a -> [a]
```

   semejante a la del ejercicio anterior, pero que efectúe una búsqueda por niveles.

   Sugerencia: utiliza una función auxiliar más general con dos parámetros adicionales `vs, ps :: [a]`, que representen el *conjunto de nodos ya visitados* y la *cola de nodos pendientes de visitar*, respectivamente.

8. En las funciones auxiliares utilizadas para resolver los dos ejercicios anteriores, el parámetro auxiliar `vs :: [a]` se usaba como representación de un *conjunto*, mientras que el parámetro auxiliar `ps :: [a]` se usaba como representación de una *pila* o una *cola*, según el caso. Reconsidera los ejercicios utilizando los TADs `Cjto`, `Pila` y `Cola` estudiados en las secciones 4.1 y 4.2.

Bibliografía

[1] R. Bird, *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall Europe, 1998. (1st edition [BW88] by R. Bird and P. Wadler; hay traducción castellana publicada en el año 2000)

[2] R. Bird and Ph. Wadler, *Introduction to Functional Programming*, Prentice Hall Europe, 1988. (2nd edition [Bir98] by R. Bird, 1998)

[3] R.M. Burstall and J. Darlington, *A Transformation System for Developing Recursive Programs*, Journal of the ACM 24, pp. 44–67, 1977.

[4] J. Fokker, *Functional Parsers*, in J. Jeuring and E. Meijer (Eds.): *Advanced Functional Programming* (First International Spring School on Advanced Functional Programming Techniques), Springer LNCS 925, pp. 1–23, 1995.

[5] P. Hudak, J. Peterson and J.H. Fasel, *A Gentle Introduction to Haskell 98*, Technical Report, University of California, Los Alamos National Laboratory, October 1999.

[6] M.P. Jones and J.C. Peterson, *Hugs 98, A functional programming system based on Haskell 98*, User Manual, revised version, September 1999.

[7] S. Peyton Jones and J. Hughes (editors), *Report on the Programming Language Haskell 98*, February 1999.

[8] R. Plasmeijer and M. van Eekelen, *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[9] F. Rabhi and G. Lapalme, *Algorithms. A Functional Programming Approach*, 2nd edition, Addison Wesley, 1999.

[10] S. Thompson, *Haskell: The Craft of Functional Programming*, 2nd edition, Addison-Wesley, 1999.

[11] P. Wadler. *Why no one uses functional languages*. ACM SIGPLAN Notices 33(8), pp. 23–27, 1998.

Apéndice A

Programas en Hugs 98

A.1 Cálculo de ceros por el método de Newton

```
module Newton where

{-- Metodo de Newton:

    Si u es una aproximacion de un cero de una funcion derivable f,
    entonces u - (f  u / f' u) es una aproximacion mejor              --}


-- Calculo aproximado de la derivada de una funcion.
-- El segundo parametro "du" representa un "valor infinitesimal".

   deriv        :: (Float -> Float) -> Float ->  (Float -> Float)
   deriv f u du  = (f(u+du) - f u) / du


-- Funcion que implementa el metodo de Newton:
-- (newton f eps x) calcula una aproximacion de un cero de f,
-- usando x como aproximacion inicial, y eps como margen de error
-- y como "infinitesimo" para el calculo de derivadas:

   newton       :: (Float -> Float) -> Float -> Float -> Float
   newton f eps  = until done improve
                   where done u    = abs (f u) < eps
                         improve u = u - (f u / deriv f u eps)

-- Funcion predefinida en el preludio:
--
-- until          :: (a -> Bool) -> (a -> a) -> a -> a
-- until p f x     = if p x then x else until p f (f x)


-- Calculo de raices n-esimas por el metodo de Newton, usando eps = 0.0001
```

```
   raiz         :: Int -> Float -> Float
   raiz n x
     | n < 2     = error "n < 2!"
     | otherwise = newton f eps x
                   where f u = u^n - x
                         eps = 0.0001

-- Casos particulares: raices cuadradas y cubicas:

   raizCuad :: Float -> Float
   raizCuad  = raiz 2

   raizCubi :: Float -> Float
   raizCubi  = raiz 3

{-- Pruebas:

Newton> raizCuad 1023
31.9844
Newton> 31.9844^2
1023.0
Newton> raizCubi 1023
10.0761
Newton> 10.0761^3
1023.0

--}
```

A.2 Los días de la semana

```
module DiaSemana where

-- Tipo de datos enumerado, para representar los dias de la semana:

data DiaSemana = Lunes | Martes | Miercoles | Jueves |
                 Viernes | Sabado | Domingo
               deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

-- Funciones booleanas que reconocen los dias festivos y laborables:

diaLaborable   :: DiaSemana -> Bool
diaLaborable d  = Lunes <= d && d <= Viernes

diaFestivo     :: DiaSemana -> Bool
diaFestivo d    = d == Sabado || d == Domingo

{-- Pruebas:
```

```
DiaSemana> diaLaborable Martes
True
DiaSemana> diaFestivo Martes
False
DiaSemana> diaLaborable Sabado
False
DiaSemana> diaFestivo Sabado
True
DiaSemana> "El " ++ show Lunes ++ " no es festivo"
"El Lunes no es festivo"
DiaSemana> diaFestivo (read "Lunes")
False
DiaSemana> fromEnum Miercoles
2
DiaSemana> toEnum 2 :: DiaSemana
Miercoles
DiaSemana> diaLaborable (toEnum 2)
True

--}
```

A.3 Emparejamiento y producto de funciones

```
module Product where

-- pair (f,g) es el emparejamiento de las funciones f y g:

   pair         :: (a -> b, a -> c) -> (a -> (b,c))
   pair (f,g)  x = (f x, g x)


-- cross (f,g) es el producto de las funciones f y g:

   cross        :: (a -> b, c -> d) -> ((a,c) -> (b,d))
   cross (f, g)  = pair(f . fst, g . snd)

{-- Pruebas:

Product> pair ((+1),(*2)) 3
(4,6)
Product> cross (not, (*2)) (True,3)
(False,6)

--}
```

A.4 Cálculo de aproximaciones enteras por defecto

```
module Floor where

   import Product

{-- floor x = n <=>_def n <= x < n+1

    Algoritmo de busqueda secuencial (Sequential Search):

    Comenzar con cualquier entero n (e.g. 0);
    decrementar hasta obtener n <= x;
    incrementar hasta obtener n > x;
    devolver n-1 como resultado                          --}

   floorSS    :: Float -> Int
-- Vale incluso:
-- floorSS :: (Num a, Ord a) => a -> Int
   floorSS x   = searchFrom 0
                 where searchFrom = decrease . upper . lower
                       lower      = until (below x) decrease
                       upper      = until (above x) increase
                       below x n  = fromInt n <= x
                       above x n  = fromInt n  > x
                       decrease n = n-1
                       increase n = n+1

-- Funcion predefinida en el preludio:
--
-- until          :: (a -> Bool) -> (a -> a) -> a -> a
-- until p f x     = if p x then x else until p f (f x)

{-- floor x = n <=>_def n <= x < n+1

    Algoritmo de busqueda binaria (Binary Search):

    Comenzar con (m,n) = (-1,1);
    mientras no se cumpla m <= x < n:
        multiplicar m y n por 2;
    (ahora se tendran enteros m, n tales que m <= x < n)
    mientras no se cumpla m+1 = n:
       calcular p = (m+n) `div` 2;
       reemplazar (m,n) por (m,p) o (p,n),
       de modo que se mantenga m <= x < n;
    devolver n como resultado                            --}

   floorBS    :: Float -> Int
-- Vale incluso:
```

```
-- floorBS :: (Ord a, Num a) => a -> Int
   floorBS x  = searchFrom (-1,1)
                where searchFrom    = fst . reduce . find
                      find          = cross (lower, upper)
                      lower         = until (below x) double
                      upper         = until (above x) double
                      below x n     = fromInt n <= x
                      above x n     = fromInt n  > x
                      double n      = 2*n
                      reduce        = until done improve
                      done (m,n)    = m+1 == n
                      improve (m,n) = if fromInt p <= x
                                         then (p,n)
                                         else (m,p)
                                         where p = (m+n) `div` 2

{-- Pruebas:

Floor> :s +s
Floor> floorSS (-1025.03)
-1026
(18528 reductions, 19580 cells)
Floor> floorSS 1025.03
1025
(15435 reductions, 16485 cells)
Floor> floorBS (-1025.03)
-1026
(597 reductions, 712 cells)
Floor> floorBS 1025.03
1025
(564 reductions, 678 cells)

--}

{-- Ejercicio: programar dos versiones de la funcion

    ceiling x = n <=>_def n-1 < x <= n

    usando un algoritmo de busqueda secuencial y un algoritmo
    de busqueda binaria, respectivamente.                        --}
```

A.5 Fusión y suma de funciones

```
module Sum where

-- join (f,g) es la fusion de las funciones f y g:

   join                 :: (a -> c, b -> c) -> (Either a b -> c)
   join (f,g) (Left x)   = f x
```

```
   join (f,g) (Right y)  = g y

-- plus (f,g) es la suma de las funciones f y g:

   plus        :: (a -> c, b -> d) -> (Either a b -> Either c d)
   plus (f,g)  = join (Left . f, Right . g)

-- either :: (a -> c) -> (b -> c) -> (Either a b -> c)
-- esta predefinida en el preludio.
-- Se cumple: either f g = join (f,g)
-- Es decir: either = curry join

{-- Pruebas:

Sum> either not isDigit (Left True)
False
Sum> either not isDigit (Right 'm')
False
Sum> join (not,isDigit) (Left True)
False
Sum> join (not,isDigit) (Right 'm')
False
Sum> plus (not,ord) (Left True)
Left False
Sum> plus (not,ord) (Right 'm')
Right 109

--}
```

A.6 Las cartas de la baraja

```
module Carta where

-- Tipo de datos construido, que representa las cartas de la baraja:

data Carta = Oro Valor | Copa Valor | Basto Valor | Espada Valor
           deriving (Eq, Ord, Read, Show)

-- Tipo de datos enumerado, que representa los valores de las cartas:

data Valor = As   | Dos   | Tres | Cuatro  | Cinco |
             Seis | Siete | Sota | Caballo | Rey
           deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

-- Tipo de datos enumerado, que representa los palos de las cartas:

data Palo = Oros | Copas | Bastos | Espadas
          deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

```
-- Funciones que seleccionan el palo y el valor de una carta dada:

palo             :: Carta -> Palo
palo (Oro _)      = Oros
palo (Copa _)     = Copas
palo (Basto _)    = Bastos
palo (Espada _)   = Espadas

valor             :: Carta -> Valor
valor (Oro v)      = v
valor (Copa v)     = v
valor (Basto v)    = v
valor (Espada v)   = v

-- Funciones booleanas que reconocen los ases y las figuras:

esAs   :: Carta -> Bool
esAs c  = valor c == As

esFigura   :: Carta -> Bool
esFigura c  = valor c == Sota || valor c == Caballo || valor c == Rey

-- Comparacion entre cartas, con distinto criterio que > :

superior     :: Carta -> Carta -> Bool
superior c d  = palo c == palo d && valor c > valor d

{-- Pruebas:

Carta> palo (Oro Seis)
Oros
Carta> valor (Oro Seis)
Seis
Carta> esAs (Oro Seis)
False
Carta> esFigura (Oro Seis)
False
Carta> esAs (Basto As)
True
Carta> esFigura (Copa Sota)
True
Carta> Basto Sota > Oro Tres
True
Carta> superior (Basto Sota) (Oro Tres)
False
Carta> Basto Sota > Basto Tres
True
Carta> superior (Basto Sota) (Basto Tres)
True
```

```
--}
```

A.7 Personas con atributos

```
module Persona where

-- Tipo sinonimo para representar personas:

type Persona = (Nombre, DNI, Edad)

-- Otros tipos sinonimos auxiliares:

type Nombre  = String

type DNI     = String

type Edad    = Int

-- Tres personas, para usar en pruebas:

rosa, pedro, herminia :: Persona

rosa      = ("Rosa Lopez Guerra", "", 12)
pedro     = ("Pedro Jimenez Ruiz", "23450876-M", 16)
herminia  = ("Herminia Garcia Simancas", "13087542-L", 43)

-- Funciones que seleccionan las componentes de una terna:

primero         :: (a,b,c) -> a
primero (x,y,z)  = x

segundo         :: (a,b,c) -> b
segundo (x,y,z)  = y

tercero         :: (a,b,c) -> c
tercero (x,y,z)  = z

-- Funciones que seleccionan los atributos de una persona:

nombre :: Persona -> Nombre
nombre  = primero

dni :: Persona -> DNI
dni  = segundo

edad :: Persona -> Edad
edad  = tercero
```

```
-- Funciones booleanas que reconocen a los ninos,
-- los adultos y los adolescentes, respectivamente:

nino   :: Persona -> Bool
nino p  = edad p < 14

adulto   :: Persona -> Bool
adulto p  = edad p >= 18

adolescente   :: Persona -> Bool
adolescente p  = not (nino p) && not (adulto p)

-- Comparacion entre personas segun su edad.
-- OJO: difiere del orden >

mayor     :: Persona -> Persona -> Bool
mayor p q  = edad p > edad q

-- Conversion de personas a cadenas de caracteres:

showPersona   :: Persona -> String
showPersona p  = showNombre p ++ "\t" ++
                 showDNI p    ++ "\t" ++
                 showEdad p

showNombre :: Persona -> String
showNombre  = nombre

showDNI :: Persona -> String
showDNI  = dni

showEdad :: Persona -> String
showEdad  = show . edad

-- Escritura de una persona en pantalla:

printPersona :: Persona -> IO ()
printPersona  = putStr . showPersona

{-- Pruebas:

Persona> adolescente pedro
True
Persona> nino herminia
False
Persona> mayor pedro rosa
True
Persona> pedro > rosa
False
```

```
Persona> showPersona herminia
"Herminia Garcia Simancas\t13087542-L\t43"
Persona> printPersona herminia
Herminia Garcia Simancas        13087542-L      43

--}
```

A.8 Un newtype para la representación de ángulos

```
module Angulo where

  newtype Angulo = Ang Float
                 deriving (Read,Show)

  normaliza    :: Float -> Float
  normaliza x
   | x < 0      = normaliza (x+rot)
   | x >= rot   = normaliza (x-rot)
   | otherwise  = x
                where rot = 2*pi
                      pi  = 3.1416

  instance Eq Angulo where

    Ang x == Ang y = normaliza x == normaliza y


  instance Ord Angulo where

    Ang x <= Ang y = normaliza x <= normaliza y


{-- Pruebas:

Angulo> normaliza 7.2832
1.0
Angulo> Ang 7.2832 <= Ang 2.0
True

--}
```

A.9 Las tres clases de listas

```
module Listas where

-- (bajando n) calcula una lista finita y terminada:
--
--             (n-1) : (n-2) : ... : 0 : []
```

```
--
-- la evaluacion termina sin error!

   bajando   :: Int -> [Int]
   bajando n
     | n >  0 = (n-1) : bajando (n-1)
     | n <= 0 = []

-- (hundiendo n) calcula una lista finita y no terminada:
--               (n-1) : (n-2) : ... : 0 : \bot
--
-- la evaluacion termina con error!

   hundiendo   :: Int -> [Int]
   hundiendo n
     | n >  0   = (n-1) : hundiendo (n-1)
     | n <= 0   = error "lista no terminada!"

-- (subiendo n) calcula una lista infinita:
--              n : (n+1) : (n+2) : ...
--
-- la evaluacion no termina!

   subiendo   :: Int -> [Int]
   subiendo n  = n : subiendo (n+1)

{-- Pruebas:

Listas> bajando 3
[2,1,0]
Listas> hundiendo 3
[2,1,0
Program error: lista no terminada!

Listas> take 3 (subiendo 0)
[0,1,2]

--}
```

A.10 La transformación `maybeMap`

```
module MaybeMap where

maybeMap         :: (a -> Maybe b) -> [a] -> [b]
maybeMap f []     = []
maybeMap f (x:xs) = case f x of
                      Just y  -> y : maybeMap f xs
                      Nothing -> maybeMap f xs
```

```
maybeHead        :: [a] -> Maybe a
maybeHead (x:_)   = Just x
maybeHead   _     = Nothing

{-- Prueba:

MaybeMap> maybeMap maybeHead [[0..], [1..0], [2..], [3..2]]
[0,2]

--}
```

A.11 Los dígitos de una cadena

```
module Digitos where

digitos :: String -> String
digitos  = filter isDigit

primerDigito    :: String -> Char
primerDigito xs  = case digitos xs of
                     []    -> '0'
                     (x:_) ->  x

{-- Pruebas:

Digitos> digitos "El dia 2 de Mayo de 2002"
"22002"
Digitos> primerDigito "El dia 2 de Mayo de 2002"
'2'
Digitos> digitos "Los tres mosqueperros"
""
Digitos> primerDigito "Los tres mosqueperros"
'0'

--}
```

A.12 Representación de una función como lista

```
module Fun where

-- Importacion del modulo Newton:

    import Newton

-- Tipo pensado para representar una funcion de tipo (a -> a)
-- por medio de su grafo, lista de tipo [(a,a)]:

    type Fun a = [(a,a)]
```

```
-- Aplicacion de una funcion f representada como grafo a un argumento x.
-- (apply f x) espera que f sea una lista de parejas ordenada
-- con respecto a las primeras componentes. Busca dos parejas
-- consecutivas (x1,y1), (x2,y2) tales que x1 <= x < x2, y
-- devuelve un valor calculado mediante interpolacion lineal.

    apply    :: (Ord a, Fractional a) => Fun a -> (a -> a)
    apply f x  = case (search f x) of
                 []                 -> error "argument not found!"
                 [(x1,y1)]          -> y1
                 [(x1,y1),(x2,y2)]  -> y1+(x-x1)*((y2-y1)/(x2-x1))

    search                  :: Ord a => [(a,b)] -> a -> [(a,b)]
    search []                     x  = []
    search ((x1,y1):[])           x  = [(x1,y1)]
    search ((x1,y1):(x2,y2):xys) x
      | x < x1                    = [(x1,y1)]               -- x < x1
      | x < x2                    = [(x1,y1),(x2,y2)]       -- x1 <= x < x2
      | otherwise                 = search ((x2,y2):xys) x  -- x2 <= x

-- Funcion de tipo (Float -> Float) representada como grafo, para pruebas.

    fun :: Fun Float
    fun  = [(0.0,1.21),(0.2,0.81),(0.4,0.49),
            (0.6,0.25),(0.8,0.09),(1.0,0.01),
            (1.2,0.01),(1.4,0.09),(1.6,0.25),
            (1.8,0.49),(2.0,0.81),(2.2,1.21)]

-- Notese: fun representa algunos puntos del grafo de la funcion \x -> (x-1.1)^2),
-- que tiene un cero en  x = 1.1


{-- Pruebas:

Calculo por interpolacion de valores de fun que no aparecen
explicitamente en el grafo:

Fun> apply fun 0.1
1.01
Fun> apply fun 0.3
0.65
Fun> apply fun 0.5
0.37
Fun> apply fun 0.7
0.17
Fun> apply fun 0.9
0.05
Fun> apply fun 1.1
```

```
0.01
Fun> apply fun 1.3
0.05
Fun> apply fun 1.5
0.17
Fun> apply fun 1.7
0.37
Fun> apply fun 1.9
0.65
Fun> apply fun 2.1
1.01

Calculo aproximado de un cero de la funcion representada por fun,
usando el metodo de Newton con 0.5 como aparoximacion inicial del cero,
y 0.02 como margen de error:

Fun> newton (apply fun) 0.02 0.5
1.025
Fun> apply fun 1.025
0.01

Calculo aproximado de un cero de la funcion representada por fun,
usando el metodo de Newton con 1.7 como aparoximacion inicial del cero,
y 0.02 como margen de error:

Fun> newton (apply fun) 0.02 1.7
1.175
Fun> apply fun 1.175
0.01

Notese: intentar el calculo anterior con un margen de error 0.01
no tendria sentido, pues ya hemos visto que apply fun 1.1 = 0.01

--}
```

A.13 Generación de listas pseudoaleatorias de números enteros

```
--         Generacion pseudoaleatoria de listas de enteros

module Azar where

-- (random a) genera una sucesion pseudoaleatoria de enteros no negativos
-- a partir de la semilla entera a >= 0

random  :: Int -> [Int]
random a = iterate nextRandom (nextRandom a)
```

```
nextRandom  :: Int -> Int
nextRandom a = ((mult a b) + 1)`mod`m
               where b = 31415821
                     m = 100000000

mult    :: Int -> Int -> Int
mult p q = (((p0*q1)+(p1*q0)`mod`m1)*m1+p0*q0)`mod`m
           where m  =  100000000
                 m1 =  10000
                 p1 =  p`div`m1
                 p0 =  p`mod`m1
                 q1 =  q`div`m1
                 q0 =  q`mod`m1


-- (randomLess r a) genera una sucesion pseudoaleatoria de enteros
-- no negativos menores que r, a partir de la semilla entera a >= 0

randomLess    :: Int -> Int -> [Int]
randomLess r a = iterate (nextRandomLess r) (nextRandomLess r a)

nextRandomLess    :: Int -> Int -> Int
nextRandomLess r a = ((a1`div`m1)*r)`div`m1
                     where b  = 31415821
                           m  = 100000000
                           m1 = 10000
                           a1 = ((mult a b) + 1)`mod`m


-- Algunas sucesiones pseudoaleatorias de enteros no negativos:

s500  = take  500 (random 23416758)
s1000 = take 1000 (random 23416758)
s1500 = take 1500 (random 23416758)
s2000 = take 2000 (random 23416758)
s2500 = take 2500 (random 23416758)
s3000 = take 3000 (random 23416758)
s3500 = take 3500 (random 23416758)
s4000 = take 4000 (random 23416758)
s4500 = take 4500 (random 23416758)
s5000 = take 5000 (random 23416758)
s5500 = take 5500 (random 23416758)

-- Algunas sucesiones pseudoaleatorias de enteros no negativos < 1000000:
-- OJO: Salen negativos, aunque no debieran.

t500  = take  500 (randomLess 1000000 23416758)
t1000 = take 1000 (randomLess 1000000 23416758)
t1500 = take 1500 (randomLess 1000000 23416758)
```

```
t2000 = take 2000 (randomLess 1000000 23416758)
t2500 = take 2500 (randomLess 1000000 23416758)
t3000 = take 3000 (randomLess 1000000 23416758)
t3500 = take 3500 (randomLess 1000000 23416758)
t4000 = take 4000 (randomLess 1000000 23416758)
t4500 = take 4500 (randomLess 1000000 23416758)
t5000 = take 5000 (randomLess 1000000 23416758)
```

A.14 Algoritmos de ordenación de listas

```
--                  Algoritmos de ordenacion de listas

module OrdenaCon where

-- Importamos un modulo de generacion de listas pseudoaleatorias de enteros:

import Azar


-- Algoritmo de odenacion por insercion:

ordIns      :: Ord a => [a] -> [a]
ordIns []     = []
ordIns (x:xs) = insOrd x (ordIns xs)

insOrd          :: Ord a => a -> [a] -> [a]
insOrd x    []   = [x]
insOrd x (y:ys)  = if x <= y
                      then x:y:ys
                      else y : insOrd x ys


-- Algoritmo de odenacion por insercion, con el orden dado como parametro:

ordInsCon           :: (a -> a -> Bool) -> [a] -> [a]
ordInsCon mig []     = []
ordInsCon mig (x:xs) = insOrdCon mig x (ordInsCon mig xs)

insOrdCon              :: (a -> a -> Bool) -> a -> [a] -> [a]
insOrdCon mig x    []   = [x]
insOrdCon mig x (y:ys)  = if mig x y
                             then x:y:ys
                             else y : insOrdCon mig x ys

-- Observacion: ordIns = ordInsCon (<=)


-- Algoritmo de ordenacion por mezcla:
```

```
ordMezcla             :: Ord a => [a] -> [a]
ordMezcla    []        = []
ordMezcla    [x]       = [x]
ordMezcla   (x:y:zs)  =  mezclaOrd (ordMezcla us) (ordMezcla vs)
                         where n       =  length (x:y:zs) `div` 2
                               (us,vs) =  splitAt n (x:y:zs)


mezclaOrd              :: Ord a => [a] -> [a] -> [a]
mezclaOrd []     ys     = ys
mezclaOrd (x:xs) []     = x:xs
mezclaOrd (x:xs) (y:ys) = if x <= y
                          then x : mezclaOrd xs (y:ys)
                          else y : mezclaOrd (x:xs) ys


-- Algoritmo de ordenacion por mezcla, con el orden dado como parametro:

ordMezclaCon                :: (a -> a -> Bool) -> [a] -> [a]
ordMezclaCon mig    []       = []
ordMezclaCon mig    [x]      = [x]
ordMezclaCon mig   (x:y:zs)  =  mezclaOrdCon mig (ordMezclaCon mig us)
                                                 (ordMezclaCon mig vs)
                             where n       =  length (x:y:zs) `div` 2
                                   (us,vs) =  splitAt n (x:y:zs)

mezclaOrdCon                  :: (a -> a -> Bool) -> [a] -> [a] -> [a]
mezclaOrdCon mig []     ys     = ys
mezclaOrdCon mig (x:xs) []     = x:xs
mezclaOrdCon mig (x:xs) (y:ys) = if mig x y
                                 then x : mezclaOrdCon mig xs (y:ys)
                                 else y : mezclaOrdCon mig (x:xs) ys

-- Observacion: ordMezcla = ordMezclaCon (<=)


-- Algoritmo de ordenacion rapida de Hoare:

ordRapido             :: Ord a => [a] -> [a]
ordRapido    []        = []
ordRapido    (x:xs)    = ordRapido [ u | u <- xs, u <= x ]
                                   ++ [x] ++
                         ordRapido [ u | u <- xs, u > x ]


-- Algoritmo de ordenacion rapida de Hoare, con el orden dado como parametro:

ordRapidoCon                :: (a -> a -> Bool) -> [a] -> [a]
ordRapidoCon mig    []       = []
```

```
ordRapidoCon mig    (x:xs)    = ordRapidoCon mig [ u | u <- xs, mig u x ]
                                                 ++ [x] ++
                                ordRapidoCon mig [ u | u <- xs, not (mig u x) ]

-- Observacion: ordRapido = ordRapidoCon (<=)


-- Funciones que generan lista de enteros, para pruebas:

pim  :: Int -> [Int]
pim n = [0 .. 2*n-1]


pam  :: Int -> [Int]
pam n = [2*n-1, 2*n-2 .. 0]


pum  :: Int -> [Int]
pum n = mezcla [0 .. n-1] [n .. 2*n-1]


pumba          :: Int -> Int -> [Int]
pumba m   n
   |  m < n     = n:m:pumba (m+1) (n-1)
   |  otherwise = [n]


mezcla                :: [a] -> [a] -> [a]
mezcla  []     []      = []
mezcla  (x:xs) (y:ys)  = x:y:mezcla xs ys

{-- Posibles pruebas:

:s +s

OrdenaCon> ordIns s500
(623983 reductions, 817272 cells, 1 garbage collection)

OrdenaCon> ordMezcla s500
(144431 reductions, 216327 cells)

OrdenaCon> ordRapido s500
(89952 reductions, 123598 cells)


--}
```

A.15 Factura de un supermercado

```
{-- Problema:

    La lectora optica de la caja de un supermercado
    produce una lista codigos de barras, tal como:

         [1234,4719,3814,1112,1113,1234]

    Hay que convertir la lista en una factura del estilo:

         Tio Pepe, 1lt.....................5.40
         Fritos de maiz....................1.21
         Cacahuetes........................0.56
         Chupa Chups (bolsa gigante).......1.33
         Item desconocido..................0.00
         Tio Pepe, 1lt.....................5.40

         Total............................13.90

    La factura se debe imprimir con el formato indicado.
    En particular, los precios aparecen en euros, con decimales.

--}

module Supermercado where

--                            Tipos de datos

type CodBar = Int
--   Codigo de barras de un producto

type Compra = [CodBar]
--   Compra de un cliente

type Nombre = String
--   Nombre de un producto

type Precio = Int
--   Precio de un producto, en centimos de euro

type BaseDatos = [(CodBar, Nombre, Precio)]
--   Ordenada en orden creciente c.r. CodBar, sin repeticiones

type Factura = [(Nombre, Precio)]
--   Facturas
```

```
--                      Base de datos para pruebas

productos :: BaseDatos
productos  = [(1111, "Chupa Chups", 21),
              (1112, "Chupa Chups (bolsa gigante)", 133),
              (1234, "Tio Pepe, 1lt", 540),
              (3814, "Cacahuetes", 56),
              (4719, "Fritos de maiz", 121),
              (5643, "Dodotis", 1010)]


--                    Juegos de datos para pruebas

compra1 :: Compra
compra1  = [1234,4719,3814,1112,1113,1234]

compra2 :: Compra
compra2  = [4719,1111,3814,4718,4719,5643]


--                       Impresion de facturas

printFactura :: Compra -> IO()
printFactura  = putStrLn . showFactura . hazFactura


--                      Confeccion de facturas

hazFactura       :: Compra -> Factura
hazFactura []     = []
hazFactura (cod:cods) = consulta cod productos : hazFactura cods

consulta :: CodBar -> BaseDatos -> (Nombre, Precio)
consulta cod bd = case [(n,p) | (c,n,p) <- bd, c == cod] of
                  []                -> ("Item desconocido", 0)
                  (nom,pre) : resto -> (nom, pre)


--                     Calculo del precio total

sumaPrecios      :: Factura -> Int
sumaPrecios  fact = sum [pre | (nom,pre) <- fact]

-- Definicion equivalente:
-- sumPrecios = sum . map snd


--                    Visualizacion de facturas
```

```
showFactura       :: Factura ->  String
showFactura  fact  = showLineas fact ++ showTotal tot
                     where tot = sumaPrecios fact

showLineas                          :: Factura -> String
showLineas []                        = ""
showLineas ((nom,pre) : restoFact)  = showLinea (nom,pre) ++ "\n" ++
                                      showLineas restoFact

showLinea          :: (Nombre, Precio) -> String
showLinea (nom,pre) = ajusta longLinea nom (showPrecio pre)

showTotal     :: Int -> String
showTotal tot  = espacios longLinea ++ "\n" ++
                 ajusta longLinea "Total" (showPrecio tot)

showPrecio    :: Int -> String
showPrecio pre = show euros ++ "." ++ show decimos ++ show centimos
                 where centimos = pre `mod` 10
                       cociente = pre `div` 10
                       decimos  = cociente `mod` 10
                       euros    = cociente `div` 10

ajusta         :: Int -> String -> String -> String
ajusta l iz dr  = rellenaCon '.' l iz dr


rellenaCon           :: Char -> Int -> String -> String -> String
rellenaCon c l iz dr  = iz ++ copia n c ++ dr
                       where n = l - (length iz + length dr)

copia    :: Int -> a -> [a]
copia n x = [x | i <- [1..n]]

espacios   :: Int -> String
espacios n  = copia n ' '

longLinea :: Int
longLinea  = 40

{-- Ejemplos de ejecucion:

Supermercado> printFactura compra1
Tio Pepe, 1lt......................5.40
Fritos de maiz.....................1.21
Cacahuetes.........................0.56
Chupa Chups (bolsa gigante).........1.33
Item desconocido...................0.00
Tio Pepe, 1lt......................5.40
```

```
Total..............................13.90

Supermercado> printFactura compra2
Fritos de maiz.....................1.21
Chupa Chups........................0.21
Cacahuetes.........................0.56
Item desconocido...................0.00
Fritos de maiz.....................1.21
Dodotis...........................10.10

Total..............................13.29

--}
```

A.16 Descomposición de un texto en líneas

```
module SeparaLineas where

 -- Un texto es una cadena de caracteres.

    type Texto = String

 -- Una linea es una cadena de caracteres
 -- que no incluya el caracter de control '\n'
 -- (salto de linea)

    type Linea = String

 -- (separaLineas txt) devuelve la lista formada
 -- por las lineas del texto txt.
 -- Se supone que un texto con n saltos de linea
 -- tiene n+1 lineas.

    separaLineas :: Texto -> [Linea]
    separaLineas  = separaCon '\n'


    separaCon   :: Eq a => a -> [a] -> [[a]]
    separaCon s  = foldr (saltaEn s) [[]]

    saltaEn              :: Eq a => a -> a -> [[a]] -> [[a]]
    saltaEn s x (xs:xss)
       | s == x           = []:xs:xss
       | s /= x           = (x:xs):xss


 -- (juntalineas lln) devuelve el texto compuesto
 -- por las lineas de la lista de lineas lln.
```

```
 -- Se supone que lln no es vacia.

    juntaLineas :: [Linea] -> Texto
    juntaLineas  = juntaCon '\n'

    juntaCon   :: a -> [[a]] -> [a]
    juntaCon s  = foldr1 (pegaCon s)

    pegaCon         :: a -> [a] -> [a] -> [a]
    pegaCon s xs ys  = xs ++ [s] ++ ys

{-- Pruebas:

SeparaLineas> separaLineas ""
[""]
SeparaLineas> separaLineas "\n"
["",""]
SeparaLineas> separaLineas "Zipi\nZape\nPantuflo"
["Zipi","Zape","Pantuflo"]
SeparaLineas> separaLineas "Zipi\n\nZape\n\nPantuflo"
["Zipi","","Zape","","Pantuflo"]

SeparaLineas> juntaLineas [""]
""
SeparaLineas> juntaLineas ["",""]
"\n"
SeparaLineas> juntaLineas ["Zipi","Zape","Pantuflo"]
"Zipi\nZape\nPantuflo"
SeparaLineas> juntaLineas ["Zipi","","Zape","","Pantuflo"]
"Zipi\n\nZape\n\nPantuflo"

--}
```

A.17 Uso de los operadores `scanl`

```
module Scan where

scanl'             :: (b -> a -> b) -> b -> [a] -> [b]
scanl' f e     []   = [e]
scanl' f e (x:xs)   = e : (scanl' f $! f e x) xs

acuSum, acuSum', acuProd, acuProd' :: Num a => [a] -> [a]
acuSum   = scanl  (+) 0
acuSum'  = scanl' (+) 0
acuProd  = scanl  (*) 1
acuProd' = scanl' (*) 1

cuadrado, cuadrado', factorial, factorial' :: Int  -> Int
cuadrado   n = acuSum   [1,3..] !! n
```

```
cuadrado'  n = acuSum'  [1,3..] !! n
factorial  n = acuProd  [1..]   !! n
factorial' n = acuProd' [1..]   !! n

{-- Pruebas:

Scan> :s +s
Scan> [cuadrado n | n <- [0..9]]
[0,1,4,9,16,25,36,49,64,81]
(1595 reductions, 2342 cells)
Scan> [cuadrado' n | n <- [0..9]]
[0,1,4,9,16,25,36,49,64,81]
(1607 reductions, 2310 cells)
Scan> [factorial n | n <- [0..9]]
[1,1,2,6,24,120,720,5040,40320,362880]
(1746 reductions, 2566 cells)
Scan> [factorial' n | n <- [0..9]]
[1,1,2,6,24,120,720,5040,40320,362880]
(1820 reductions, 2596 cells)

--}
```

A.18 Gráficos de tortuga

```
                          -- Graficos de tortuga --

module Tortuga where


-- Tipos para representar el estado de la tortuga:

type Estado = (Direccion,Pluma,Punto)

type Direccion = Int

-- 0: Direccion del semieje X -
-- 1: Direccion del semieje Y +
-- 2: Direccion del semieje X +
-- 3: Direccion del semieje Y -

type Pluma = Bool

-- True:  Pluma bajada (dibuja)
-- False: Pluma alzada (no dibuja)

type Punto = (Int,Int)
```

```
-- Estado inicial de la tortuga:
-- En el origen, mirando hacia X -, con la pluma alzada.

estadoInicial :: Estado
estadoInicial = (0,False,(0,0))


-- Instrucciones para la tortuga:

type Instruccion = Estado -> Estado

-- Instruccion para avanzar un paso:

avanza :: Instruccion
avanza (0,p,(x,y)) = (0,p,(x-1,y))
avanza (1,p,(x,y)) = (1,p,(x,y+1))
avanza (2,p,(x,y)) = (2,p,(x+1,y))
avanza (3,p,(x,y)) = (3,p,(x,y-1))

-- Instruccion para girar 90 grados a la dcha.:

dcha :: Instruccion
dcha (d,p,(x,y)) = ((d+1)`mod`4,p,(x,y))

-- Instruccion para girar 90 grados a la izqda.:

izqda :: Instruccion
izqda (d,p,(x,y)) = ((d+3)`mod`4,p,(x,y))

-- Instrucciones para subir y bajar la pluma:

sube :: Instruccion
sube (d,p,(x,y)) = (d,False,(x,y))

baja :: Instruccion
baja (d,p,(x,y)) = (d,True,(x,y))


-- Programas para la tortuga:

type Programa = [Instruccion]

-- Funcion auxiliar:

copy :: Int -> a -> [a]
copy n x = [x | i <- [1..n] ]

-- Programa para dibujar un cuadrado de lado l:
```

```
cuadrado :: Int -> Programa
cuadrado l = [baja] ++ concat (copy 4 lado) ++ [sube]
                        where lado = copy l avanza ++ [dcha]

-- Programa para dibujar n cuadrados de lado l,
-- ascendiendo en diagonal hacia la izqda.:

cuadrados ::Int -> Int -> Programa
cuadrados n l = [baja] ++ concat (copy n esquina) ++
                [dcha,dcha] ++ concat (copy n esquina) ++
                [dcha,dcha,sube]
                where esquina = copy l avanza ++ [dcha] ++
                                copy l avanza ++ [izqda]


-- Ejecucion de programas de la tortuga:

-- Funcion que interpreta un programa, dando una lista de estados:

tortuga :: Programa -> [Estado]
tortuga = scanl' aplicarA estadoInicial
          where aplicarA estado instruccion = instruccion estado

-- (scanl' es una variante de scanl que fuerza la evaluacion
-- impaciente de la operacion que va aplicandose a los elementos
-- de la lista procesada)

scanl'        :: (a -> b -> a) -> a -> [b] -> [a]
scanl' f q xs = q:(case xs of
                           []    -> []
                           x:xs' -> strict (scanl' f) (f q x) xs')

-- Funcion strict: dada f :: a -> b, strict f :: a -> b es una version
-- estricta de f, que fuerza la evaluacion del parametro de f.
-- El operador ($!) :: (a -> b) -> a -> b es primitivo.
-- (f $! x) se comporta como (f x), pero forzando la evaluacion de x.

strict     :: (a -> b) -> a -> b
strict f x = f $! x

-- Funcion que visualiza un programa de la tortuga, generando
-- el dibujo obtenido en forma de cadena de caracteres:

visualiza :: Programa -> String
visualiza = presenta . dibujo . traza . tortuga

-- Funcion que ejecuta un programa de la tortuga, mostrando
-- en pantalla el dibujo obtenido:
```

```
ejecuta :: Programa -> IO()
ejecuta = putStr . visualiza

-- Funcion que convierte una lista de estados en una traza,
-- que es la lista de puntos donde se ha dibujado:

type Traza = [Punto]

traza :: [Estado] -> Traza
traza estados = [ (x,y) | (d,True,(x,y)) <- estados ]

-- Tipo de los dibujos, representados por filas como
-- listas de listas de caracteres:

type Dibujo = [Fila]
type Fila   = String

-- Funcion que presenta un dibujo, convirtiendolo en
-- una sola cadena de caracteres con saltos de linea:

presenta :: Dibujo -> String
presenta = ('\n':) . unlines

-- Funcion que genera un dibujo a partir de una traza,
-- convirtiendo previamente el dibujo en una matriz booleana,
-- donde las posiciones True indican donde hay que dibujar.

type Bitmap = [[Bool]]

dibujo :: Traza -> Dibujo
dibujo = interpreta . bitmap

-- Funcion que interpreta una matriz booleana asociando el
-- caracter '*' a True. Podria optarse por interpretar de
-- alguna otra manera.

interpreta :: Bitmap -> Dibujo
interpreta = map (map asterisco)
             where asterisco True  = '*'
                   asterisco False = ' '

-- Funcion que genera una matriz booleana a partir de una traza.
-- La funcion rellena tiene el efecto de que cada punto de una linea
-- se represente como un caracter seguido de un blanco, para compensar
-- el efecto visual de "hueco" entre cada dos lineas.

bitmap :: Traza -> Bitmap
bitmap puntos = [ rellena [ elem (x,y) puntos | x <- ranX ] | y <- ranY ]
                where ranX = rango (map fst puntos)
```

```
                    ranY = reverse (rango (map snd puntos))

-- Funciones auxiliares:

rellena :: [Bool] -> [Bool]
rellena = concat . map hueco
          where hueco b = [b,False]

rango    :: (Ord a, Enum a) => [a] -> [a]
rango zs = [minimum zs .. maximum zs]



-- Pruebas:

-- Dibujo de cuadrados de diferentes lados:

-- ejecuta (cuadrado  5)
-- ejecuta (cuadrado 10)
-- ejecuta (cuadrado 15)
-- ejecuta (cuadrado 20)

-- Dibujo de n cuadrados de lado 5, para diferentes valores de n:

-- ejecuta (cuadrados 1 5)
-- ejecuta (cuadrados 2 5)
-- ejecuta (cuadrados 3 5)
-- ejecuta (cuadrados 4 5)
```

A.19 Proceso incordiante

```
module Incordia where

done :: IO ()
done  = return ()

incordia :: IO ()
incordia  = do putStrLn "Adivina como me llamo: "
               insiste

insiste :: IO ()
insiste  = do linea <- getLine
              if linea == "Incordion"
                 then do putStrLn "Acertaste. Abur."
                         done
                 else do putStrLn "Fallaste. Prueba otra vez."
                         insiste

{-- Prueba:
```

```
Incordia> incordia
Adivina como me llamo:
Felipe
Fallaste. Prueba otra vez.
Tadeo
Fallaste. Prueba otra vez.
Incordion
Acertaste. Abur.


--}
```

A.20 Proceso que invierte las líneas de la entrada

```
module InvierteLineas where

type Entrada     = String
type Salida      = String
type Interaccion = Entrada -> Salida

invLin         :: Interaccion
invLin entrada  = "Hola!\n" ++
                  "Invertire todas las lineas que teclees.\n" ++
                  "Me detendre cuando teclees stop\n\n" ++
                  salida ++
                  "\n\nAdios!"
                  where lineas = takeWhile (/= "stop") (lines entrada)
                        lineasInversas = [reverse linea | linea <- lineas]
                        salida = unlines lineasInversas

inversor :: IO ()
inversor  = interact invLin

inversor' :: IO ()
inversor'  = do putStrLn ("Hola!\n" ++
                          "Invertire todas las lineas que teclees.\n" ++
                          "Me detendre cuando teclees stop\n")
                invierteLineas

invierteLineas :: IO ()
invierteLineas  = do linea <- getLine
                     if linea == "stop"
                        then do putStrLn "\nAdios!"
                                done
                        else do putStrLn (reverse linea)
                                invierteLineas

done :: IO ()
```

```
done  = return ()

{-- Pruebas:

InvierteLineas> inversor
Hola!
Invertire todas las lineas que teclees.
Me detendre cuando teclees stop

dabale arroz a la zorra el abad
daba le arroz al a zorra elabad
madam i am adam
mada ma i madam
stop


Adios!
InvierteLineas> inversor'
Hola!
Invertire todas las lineas que teclees.
Me detendre cuando teclees stop

dabale arroz a la zorra el abad
daba le arroz al a zorra elabad
madam ia am adam
mada ma ai madam
stop

Adios!

--}
```

A.21 Proceso que invierte las líneas de un archivo

```
module Copia where

copion :: IO ()
copion  = do putStr "Teclea la senda origen: "
             origen <- readLn
             putStr "Teclea la senda destino: "
             destino <- readLn
             copiaInv origen destino
             putStrLn ("Copia inversa del archivo "
                          ++ origen ++ "\n" ++
                       "realizada en el archivo "
                          ++ destino)

copiaInv                :: FilePath -> FilePath -> IO ()
copiaInv origen destino  = do cont <- readFile origen
```

```
                            let contInv = unlines
                                        (map reverse (lines cont))
                                in writeFile destino contInv

{-- Prueba:

"pali.txt" contiene:

dabale arroz a la zorra el abad
madam i am adam

Copia> copion
Teclea la senda origen: "pali.txt"
Teclea la senda destino: "ilap.txt"
Copia inversa del archivo pali.txt
realizada en el archivo ilap.txt

"ilap.txt" contiene:

daba le arroz al a zorra elabad
mada ma i madam


--}
```

A.22 Ordenación de listas de personas

```
module Personas where

-- Importacion de un modulo con algoritmos de ordenacion de listas

import OrdenaCon


--  Tipo sinonimo para representar personas, con algunas operaciones

type Persona = (Nombre, DNI, Edad)

type Nombre  = String

type DNI     = String

type Edad    = Int

primero         :: (a,b,c) -> a
primero (x,y,z)  = x

segundo         :: (a,b,c) -> b
```

```
segundo (x,y,z)  = y

tercero         :: (a,b,c) -> c
tercero (x,y,z)  = z

nombre :: Persona -> Nombre
nombre  = primero

dni :: Persona -> DNI
dni  = segundo

edad :: Persona -> Edad
edad  = tercero

nino   :: Persona -> Bool
nino p  = edad p < 14

adulto   :: Persona -> Bool
adulto p  = edad p >= 18

adolescente   :: Persona -> Bool
adolescente p  = not (nino p) && not (adulto p)


--  Diversos ordenes para comparar personas con distintos criterios

migNombre     :: Persona -> Persona -> Bool
migNombre p q  = nombre p <= nombre q

migDNI     :: Persona -> Persona -> Bool
migDNI p q  = dni p <= dni q

migEdad     :: Persona -> Persona -> Bool
migEdad p q  = edad p <= edad q


-- Visualizacion de una persona:

showPersona  :: Persona -> String
showPersona p = showNombre p ++ "\t" ++ showDNI p ++ "\t" ++ showEdad p

showNombre   :: Persona -> String
showNombre p  = ajustaI 25 (nombre p)

showDNI   :: Persona -> String
showDNI p  = ajustaI 12 (dni p)

showEdad   :: Persona -> String
showEdad p  = ajustaI 3 (show (edad p))
```

```
-- Ajuste de un acadena de caracteres al margen izqdo.:

ajustaI     :: Int -> String -> String
ajustaI n xs = xs ++ copia m ' '
               where m = n - length xs

copia    :: Int -> a -> [a]
copia n x = [x | i <- [1..n]]


-- Impresion de una persona:

printPersona :: Persona -> IO()
printPersona  = putStr . showPersona


-- Impresion de una lista de personas:

printPersonas       :: [Persona] -> IO()
printPersonas []     = return ()
printPersonas (p:ps) = do printPersona p
                          putChar '\n'
                          printPersonas ps


--  Muestrario de personas

p1, p2, p3, p4, p5, p6, p7, p8, p9, p10 :: Persona

p1  = ("Rosa Lopez Guerra", "", 12)
p2  = ("Pedro Jimenez Ruiz", "23450876-M", 16)
p3  = ("Herminia Garcia Simancas", "13087542-L", 43)
p4  = ("Juan Gonzalez Cifuentes", "", 8)
p5  = ("Maria Suarez Martin", "609874570-L", 75)
p6  = ("Felipe Romero Zurita", "798045601-Q", 17)
p7  = ("Sonia Hernandez Padron", "60543273-R", 21)
p8  = ("Carlos Delgado Sotillo", "506789401-M", 36)
p9  = ("Marta Zoilo Perez", "", 5)
p10 = ("Hernando Fraile Pelaez", "670546301-E", 80)

personas :: [Persona]
personas  = [p1, p2, p3, p4, p5, p6, p7, p8, p9, p10]


{--  Ejemplos de ejecucion:

Persona> printPersonas personas
```

```
Rosa Lopez Guerra                              12
Pedro Jimenez Ruiz              23450876-M     16
Herminia Garcia Simancas        13087542-L     43
Juan Gonzalez Cifuentes                        8
Maria Suarez Martin             609874570-L    75
Felipe Romero Zurita            798045601-Q    17
Sonia Hernandez Padron          60543273-R     21
Carlos Delgado Sotillo          506789401-M    36
Marta Zoilo Perez                              5
Hernando Fraile Pelaez          670546301-E    80

Persona> printPersonas (ordRapido personas)
Carlos Delgado Sotillo          506789401-M    36
Felipe Romero Zurita            798045601-Q    17
Herminia Garcia Simancas        13087542-L     43
Hernando Fraile Pelaez          670546301-E    80
Juan Gonzalez Cifuentes                        8
Maria Suarez Martin             609874570-L    75
Marta Zoilo Perez                              5
Pedro Jimenez Ruiz              23450876-M     16
Rosa Lopez Guerra                              12
Sonia Hernandez Padron          60543273-R     21

Personas> printPersonas (ordRapidoCon migNombre personas)
Carlos Delgado Sotillo          506789401-M    36
Felipe Romero Zurita            798045601-Q    17
Herminia Garcia Simancas        13087542-L     43
Hernando Fraile Pelaez          670546301-E    80
Juan Gonzalez Cifuentes                        8
Maria Suarez Martin             609874570-L    75
Marta Zoilo Perez                              5
Pedro Jimenez Ruiz              23450876-M     16
Rosa Lopez Guerra                              12
Sonia Hernandez Padron          60543273-R     21

Persona> printPersonas (ordRapidoCon migEdad personas)
Marta Zoilo Perez                              5
Juan Gonzalez Cifuentes                        8
Rosa Lopez Guerra                              12
Pedro Jimenez Ruiz              23450876-M     16
Felipe Romero Zurita            798045601-Q    17
Sonia Hernandez Padron          60543273-R     21
Carlos Delgado Sotillo          506789401-M    36
Herminia Garcia Simancas        13087542-L     43
Maria Suarez Martin             609874570-L    75
Hernando Fraile Pelaez          670546301-E    80

Persona> printPersonas (ordRapidoCon migDNI personas)
Marta Zoilo Perez                              5
```

```
Juan Gonzalez Cifuentes                          8
Rosa Lopez Guerra                                12
Herminia Garcia Simancas        13087542-L       43
Pedro Jimenez Ruiz              23450876-M       16
Carlos Delgado Sotillo          506789401-M      36
Sonia Hernandez Padron          60543273-R       21
Maria Suarez Martin             609874570-L      75
Hernando Fraile Pelaez          670546301-E      80
Felipe Romero Zurita            798045601-Q      17

--}
```

A.23 Formateo de un texto

```
{-- Problema: Procesar un texto sin formatear, tal como:

    The heat bloomed      in December as the
        carnival  season
                kicked into  gear.
    Nearly helpless with sun and glare, I avoided Rio's
    brilliant sidewalks
      and glittering beaches,
    panting in dark   corners
    and waiting out the inverted southern summer.

y que resulte un texto sin espaciados multiples y alineado a la izqda.,
del estilo:

    The heat bloomed in December as the
    carnival season kicked into gear.
    Nearly helpless with sun and glare,
    I avoided Rio's brilliant sidewalks
    and glittering beaches, panting in
    dark corners and waiting out the
    inverted southern summer.

Se supone conocida la longitud de una linea
(35 espacios, en el ejemplo anterior)

--}

module ProcTexto where


--                          Tipos de datos

type Texto = String

type Linea = String
```

```
--   Una linea no debe contener el caracter '\n'

type Palabra = String
--   Una palabra no debe contener los caracteres '\n', '\t', ' '

type LongLinea = Int
--   Numero maximo de caracteres de una linea


--                  Descomposicion del problema

-- Dado un texto, lo procesaremos en dos etapas:
-- Etapa 1: Transformacion del texto en una lista de palabras.
-- Etapa 2: Transformacion de la lista de palabras en un texto formateado.

formatea :: Texto -> Texto
formatea  = componTexto . separaPalabras . saltaBlancos


--       Eliminacion de los "blancos" iniciales de un texto

saltaBlancos :: Texto -> Texto
saltaBlancos  = dropWhile blanco

blanco    :: Char -> Bool
blanco x   = elem x ['\n', '\t', ' ']


--       Transformacion de un texto en una lista de palabras
--      Se espera que le texto contenga al menos una palabra


separaPalabras      :: Texto -> [Palabra]
-- Espera un texto que no comience por un "blanco"
separaPalabras texto = if null texto
                       then []
                       else primPal : separaPalabras (saltaBlancos resTexto)
                     where (primPal, resTexto) = separaPalabra texto

separaPalabra            :: Texto -> (Palabra, Texto)
-- Espera un texto inicialmente no vacio que no comience por un "blanco"
separaPalabra ""          = ("", "")
separaPalabra txt@(x:xs)  = if blanco x
                            then ("", txt)
                            else (x:us, vs)
                          where (us, vs) = separaPalabra xs


--         Transformacion de una lista de palabras en un texto
```

```
longLinea :: LongLinea
longLinea  = 35

componTexto :: [Palabra] -> Texto
componTexto  = uneLineas . separaLineas longLinea

separaLineas               :: LongLinea -> [Palabra] -> [[Palabra]]
separaLineas long palabras  = if null palabras
                                 then []
                                 else primLin : separaLineas long resPalabras
                              where (primLin, resPalabras) = separaLinea long palabras

separaLinea                      :: LongLinea -> [Palabra] -> ([Palabra], [Palabra])
separaLinea long []               = ([], [])
separaLinea long palabras@(w:ws)
  | longw > long                  = ([], palabras)
  | otherwise                     = (w : palabrasPrimLin, resto)
                                    where longw = length w
                                          nlong = long - (longw + 1)
                                          (palabrasPrimLin, resto) = separaLinea nlong ws

uneLineas                   :: [[Palabra]] -> Texto
uneLineas []                 = ""
uneLineas (palabras : resto) = unePalabras palabras ++ "\n" ++ uneLineas resto

unePalabras :: [Palabra] -> String
unePalabras []                   = ""
unePalabras (palabra : palabras) = palabra ++ " " ++ unePalabras palabras



--                             Texto para prueba sencilla

texto  :: Texto
texto   = "The heat bloomed      in December as the" ++ "\n" ++
          "   carnival  season" ++ "\n" ++
          "            kicked into  gear." ++ "\n" ++
          "Nearly helpless with sun and glare, I avoided Rio's" ++ "\n" ++
          "brilliant sidewalks" ++ "\n" ++
          "  and glittering beaches," ++ "\n" ++
          "panting in dark   corners" ++ "\n" ++
          "and waiting out the inverted southern summer."


--                Proceso de escritura en pantalla para prueba sencilla

printTexto :: Texto -> IO()
printTexto = putStr . formatea
```

```
--  Proceso que lee un texto de un archivo y lo guarda formateado en otro archivo

procTexto :: IO ()
procTexto  = do putStr "Senda al archivo origen: "
                origen <- readLn
                putStr "Senda al archivo destino: "
                destino <- readLn
                formateaTexto origen destino
                putStrLn ("El contenido del archivo "
                             ++ origen ++ "\n" ++
                          "se ha formateado y grabado en el archivo "
                             ++ destino)

formateaTexto                :: FilePath -> FilePath -> IO ()
formateaTexto origen destino  = do txt <- readFile origen
                                   writeFile destino (formatea txt)


--         Proceso que muestra en pantalla el contenido de un archivo


muestraArchivo      :: FilePath -> IO ()
muestraArchivo senda = do txt <- readFile senda
                          putStrLn txt


{-- Ejemplos de ejecucion:

ProcTexto> printTexto texto
The heat bloomed in December as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering beaches, panting in
dark corners and waiting out the
inverted southern summer.

ProcTexto> muestraArchivo "texto.txt"
The heat bloomed      in December as the
    carnival  season
                kicked into  gear.
Nearly helpless with sun and glare, I avoided Rio's
brilliant sidewalks
  and glittering beaches,
panting in dark   corners
and waiting out the inverted southern summer.
```

```
ProcTexto> procTexto
Senda al archivo origen: "texto.txt"
Senda al archivo destino: "ftexto.txt"
El contenido del archivo texto.txt
se ha formateado y grabado en el archivo ftexto.txt

ProcTexto> muestraArchivo "ftexto.txt"
The heat bloomed in December as the
carnival season kicked into gear.
Nearly helpless with sun and glare,
I avoided Rio's brilliant sidewalks
and glittering beaches, panting in
dark corners and waiting out the
inverted southern summer.

--}
```

A.24 Pilas

```
--                    Las pilas genericas como TAD


module Pila (Pila, vacia, apila, desapila, cima, esVacia, contenido) where

-- Tipo representante [a]. Oculto.

newtype Pila a = P [a]

-- Invariante de la representacion trivial.
-- valida (P xs) = True

-- Equivalencia abstracta trivial.
-- (P xs) representa una pila con contenido xs.
-- La cabeza de xs corresponde a la cima de la pila.
-- El ultimo elemento de xs corresponde al fondo de la pila.
-- equiv (P xs) (P ys) = (xs = ys)


-- Operaciones publicas. Exportadas.

vacia  :: Pila a
vacia   = P []

apila          :: Pila a -> a -> Pila a
apila (P xs) x  = P (x:xs)

desapila            :: Pila a -> Pila a
desapila (P [])      = error "desapila (Pila.vacia)"
desapila (P (_:xs))  = P xs
```

```
cima          :: Pila a -> a
cima (P [])    = error "cima (Pila.vacia)"
cima (P (x:_)) = x

esVacia        :: Pila a -> Bool
esVacia (P xs)  = null xs

-- Operacion exportada por conveniencia.

contenido        :: Pila a -> [a]
contenido (P xs)  = xs


{-- Evolucion de la representacion interna de una pila.

    Operacion            Representacion
    ---------            --------------

    vacia                P []
    apila 7              P [7]
    apila 5              P [5,7]
    apila 9              P [9,5,7]
    desapila             P [5,7]
    apila 7              P [7,5,7]
    apila 3              P [3,7,5,7]
    desapila             P [7,5,7]
    desapila             P [5,7]

--}

{-- Pruebas sencillas:

Pila> contenido (apila (apila (apila vacia 1) 2) 3)
[3,2,1]
Pila> cima (apila (apila (apila vacia 1) 2) 3)
3
Pila> contenido (desapila (apila (apila (apila vacia 1) 2) 3))
[2,1]

--}
```

A.25 Pilas interactivas

```
--                    Pila Generica interactiva


module ActivaPila (activaPilaTipo) where
```

```
-- Importacion del TAD Pila

import Pila

-- Tipo sinonimo para representar ordenes del usuario.

type Orden = String

-- Constante indefinida de tipo generico.

anything :: a
anything  = error "indefinido!"

-- Saludo mostrado al usuario cuando se activa una pila:

anuncioPilaTipo    :: Show a => a -> String
anuncioPilaTipo x  = "Se ha activado una pila interactiva (inicialmente vacia) \n" ++
                     "capaz de almacenar elementos del tipo determinado por el parametro."

-- Despedida mostrada al usuario cuando se desactiva una pila:

despedidaPila :: String
despedidaPila  = "Se cierra la sesion.\n" ++
                 "La pila queda desactivada."

-- Menu de ordenes para pilas.
-- Aparece cuando se crea una pila interactiva
-- y tambien al ejecutar la orden "ayuda"

-- menuPila :: String

menuPila  = "Ordenes disponibles: \n" ++
            "  ayuda: muestra esta ayuda. \n" ++
            "  vacia: vacia la pila. \n" ++
            "  apila: apila un nuevo elemento. \n" ++
            "  desapila: desapila la cima. \n" ++
            "  cima: muestra la cima. \n" ++
            "  esVacia: consulta si la pila es vacia. \n" ++
            "  contenido: muestra el contenido de la pila. \n" ++
            "  desactiva: termina la sesion, desactivando la pila."

-- Proceso parametrizado para crear una pila interactiva,
-- adecuada para almacenar elementos del mismo tipo que el parametro.
-- Inicialmente, la pila esta vacia.

activaPilaTipo   :: (Read a, Show a) => a -> IO ()
activaPilaTipo x  = do putChar '\n'
                       putStrLn (anuncioPilaTipo x)
                       putStrLn menuPila
```

```
-- Proceso principal.
-- Interactua con una pila dada como parametro.
-- Su efecto es:
--    pedir una orden al usuario;
--    ejecutar la orden;
--    reactivarse a si mismo
--    (excepto si la orden era "cierra")

sesionConPilaTipo     :: (Read a, Show a) => a -> Pila a -> IO ()

sesionConPilaTipo x p  = do putChar '\n'
                            putStr "Orden? "
                            orden <- getLine
                            ejecutaConPilaTipo x p orden

-- Proceso auxiliar.
-- Depende de una pila y una orden dados como parametros.
-- Su efecto es:
--    ejecutar la orden, calculando si es preciso una nueva pila
--    llamar al proceso principal, con la nueva pila.
-- Caso especial:
--    La orden "desactiva" se ejecuta terminando la interaccion.


ejecutaConPilaTipo               :: (Read a, Show a) => a -> Pila a -> Orden -> IO ()

ejecutaConPilaTipo x p "ayuda"    = do putChar '\n'
                                       putStrLn menuPila
                                       sesionConPilaTipo x p

ejecutaConPilaTipo x p "vacia"     = do putChar '\n'
                                        putStrLn "Se ha vaciado la pila!"
                                        sesionConPilaTipo x vacia

ejecutaConPilaTipo x p "apila"     = do putChar '\n'
                                        putStr "Elemento a apilar? "
                                        e <- readLn
                                        sesionConPilaTipo x (apila p e)

ejecutaConPilaTipo x p "desapila"  = do putChar '\n'
                                        if esVacia p
                                           then do putStrLn "Error: desapila vacia!"
                                                   sesionConPilaTipo x p
                                           else sesionConPilaTipo x (desapila p)

ejecutaConPilaTipo x p "cima"      = do putChar '\n'
                                        if esVacia p
```

```
                                        then do putStrLn "Error: cima vacia!"
                                                sesionConPilaTipo x p
                                        else do print (cima p)
                                                sesionConPilaTipo x p

ejecutaConPilaTipo x p "esVacia"   =  do putChar '\n'
                                         if esVacia p
                                            then putStrLn "La pila esta vacia."
                                            else putStrLn "La pila no esta vacia."
                                         sesionConPilaTipo x p

ejecutaConPilaTipo x p "contenido" =  do putChar '\n'
                                         imprimePila (contenido p)
                                         sesionConPilaTipo x p

ejecutaConPilaTipo x p "desactiva" =  do putChar '\n'
                                         putStrLn despedidaPila

ejecutaConPilaTipo x p  _          =  do putChar '\n'
                                         putStrLn "Orden desconocida!"
                                         sesionConPilaTipo x p

-- Proceso auxiliar que muestra el contenido de una pila:

imprimePila    :: Show a => [a] -> IO ()
imprimePila xs  = do putStr "cima -> "
                     imprimeElementos xs
                     putStrLn " <- fondo"

imprimeElementos        :: Show a => [a] -> IO ()
imprimeElementos []      = return ()
imprimeElementos (x:xs)  = do putStr siguiente
                              imprimeElementos xs
                              where siguiente = if null xs
                                                then show x
                                                else show x ++ ", "



{-- Pruebas de activacion de pilas de diferentes tipos:

ActivaPila> activaPilaTipo (anything :: Bool)

ActivaPila> activaPilaTipo (anything :: Int)

ActivaPila> activaPilaTipo (anything :: Float)

ActivaPila> activaPilaTipo (anything :: Char)
```

```
ActivaPila> activaPilaTipo (anything :: String)

ActivaPila> activaPilaTipo (anything :: (In,Int))

ActivaPila> activaPilaTipo (anything :: (Bool,(Int,Int))

ActivaPila> activaPilaTipo (anything :: Either Int Bool)

--}
```

A.26 Colas ingenuas

```
--                      Las colas genericas como TAD:
--                         implementacion ingenua


module ColaIngenua (Cola, vacia, mete, resto, primero, esVacia, contenido) where

-- Tipo representante [a]. Oculto.

newtype Cola a = C [a]

-- Invariante de la representacion trivial.
-- valida (P xs) = True

-- Equivalencia abstracta trivial.
-- (C xs) representa una cola con contenido xs.
-- La cabeza de xs corresponde al primer dato de la cola.
-- El ultimo elemento de xs corresponde al ultimo dato que ha entrado en la cola.
-- equiv (C xs) (C ys) = (xs = ys)


--Operaciones publicas. Exportadas.

vacia  :: Cola a
vacia   = C []

mete          :: Cola a -> a -> Cola a
mete (C xs) x  = C (xs ++ [x])

resto        :: Cola a -> Cola a
resto (C xs)  = if null xs
                  then error "resto (Cola.vacia)"
                  else C (tail xs)

primero        :: Cola a -> a
primero (C xs)  = if null xs
                    then error "primero (Cola.vacia)"
                    else (head xs)
```

```
esVacia        :: Cola a -> Bool
esVacia (C xs)  = null xs

-- Operacion exportada por conveniencia.

contenido        :: Cola a -> [a]
contenido (C xs)  = xs


{-- Evolucion de la representacion interna de una cola.

    Operacion            Representacion
    ---------            --------------

    vacia                C []
    mete 7               C [7]
    mete 5               C [7,5]
    mete 9               C [7,5,9]
    resto                C [5,9]
    mete 7               C [5,9,7]
    mete 3               C [5,9,7,3]
    resto                C [9,7,3]
    resto                C [7,3]

--}

{-- Pruebas sencillas:

Cola> contenido (mete (mete (mete vacia 1) 2) 3)
[1,2,3]
Cola> primero (mete (mete (mete vacia 1) 2) 3)
1
Cola> contenido (resto (mete (mete (mete vacia 1) 2) 3))
[2,3]

--}
```

A.27 Colas eficientes

```
--                  Las colas genericas como TAD:
--                  implementacion mas eficiente


module Cola (Cola, vacia, mete, resto, primero, esVacia, contenido) where

-- Tipo representante ([a],[a])

newtype Cola a = C ([a],[a])
```

```
-- (C pr ir) representa una cola con contenido pr ++ reverse ir
-- Es decir: la lista pr contiene los primeros datos de la cola,
-- y la lista ir contiene el resto de los datos de la cola, en orden inverso.

-- Invariante de la representacion no trivial.
-- (C pr ir) debe cumplir: null pr => null ir, i.e.
-- valida (C pr ir) = not (null pr) || null ir

-- Equivalencia abstracta no trivial.
-- equiv (C pr ir) (C pr' ir') = (pr ++ revrse ir = pr' ++ reverse ir')

-- Operaciones publicas. Exportadas.

vacia  :: Cola a
vacia   = C ([], [])

mete                 :: Cola a -> a -> Cola a
mete (C (pr, ir)) x  = hazValida (C (pr, (x:ir)))

resto              :: Cola a -> Cola a
resto (C (pr, ir))  = if null pr
                         then error "resto (Cola.vacia)"
                         else hazValida (C (tail pr, ir))

primero              :: Cola a -> a
primero (C (pr, ir))  = if null pr
                           then error "primero (Cola.vacia)"
                           else head pr

esVacia              :: Cola a -> Bool
esVacia (C (pr, ir))  = null pr

-- Operacion exportada por conveniencia.

contenido              :: Cola a -> [a]
contenido (C (pr, ir))  = pr ++ reverse ir

-- Operacion auxiliar privada. No exportada.

hazValida              :: Cola a -> Cola a
hazValida (C (pr, ir))  = if null pr
                          then C (reverse ir, [])
                          else C (pr, ir)


{-- Evolucion de la representacion interna de una cola eficiente.

    Operacion            Representacion
```

```
    ---------           --------------

    vacia               C ([],       [])
    mete 7              C ([7],      [])
    mete 5              C ([7],     [5])
    mete 9              C ([7],   [9,5])
    resto               C ([5,9],    [])
    mete 7              C ([5,9],   [7])
    mete 3              C ([5,9], [3,7])
    resto               C ([9],   [3,7])
    resto               C ([7,3],    [])

--}

{-- Pruebas sencillas:

ColaEficiente> contenido (mete (mete (mete vacia 1) 2) 3)
[1,2,3]
ColaEficiente> primero (mete (mete (mete vacia 1) 2) 3)
1
ColaEficiente> contenido (resto (mete (mete (mete vacia 1) 2) 3))
[2,3]

--}
```

A.28 Colas interactivas

```
--                    Cola Generica interactiva


module ActivaCola (activaColaTipo) where

-- Importacion del TAD Cola

import Cola

-- Tipo sinonimo para representar ordenes del usuario.

type Orden = String

-- Constante indefinida de tipo generico.

anything :: a
anything  = error "indefinido!"

-- Saludo mostrado al usuario cuando se activa una cola:

anuncioColaTipo    :: Show a => a -> String
anuncioColaTipo x  = "Se ha activado una cola interactiva (inicialmente vacia) \n" ++
```

```
                    "capaz de almacenar elementos del tipo determinado por el parametro."

-- Despedida mostrada al usuario cuando se desactiva una cola:

despedidaCola :: String
despedidaCola  = "Se cierra la sesion.\n" ++
                 "La cola queda desactivada."

-- Menu de ordenes para colas.
-- Aparece cuando se crea una cola interactiva
-- y tambien al ejecutar la orden "ayuda"

-- menuCola :: String

menuCola  = "Ordenes disponibles: \n" ++
            "  ayuda: muestra esta ayuda. \n" ++
            "  vacia: vacia la cola. \n" ++
            "  mete: pone en la cola un nuevo elemento. \n" ++
            "  resto: elimina el primer elemento. \n" ++
            "  primero: muestra el primer elemento. \n" ++
            "  esVacia: consulta si la cola es vacia. \n" ++
            "  contenido: muestra el contenido de la cola. \n" ++
            "  desactiva: termina la sesion, desactivando la cola."

-- Proceso parametrizado para crear una cola interactiva,
-- adecuada para almacenar elementos del mismo tipo que el parametro.
-- Inicialmente, la cola esta vacia.

activaColaTipo   :: (Read a, Show a) => a -> IO ()
activaColaTipo x  = do putChar '\n'
                       putStrLn (anuncioColaTipo x)
                       putStrLn menuCola
                       sesionConColaTipo x vacia

-- Proceso principal.
-- Interactua con una cola dada como parametro.
-- Su efecto es:
--    pedir una orden al usuario;
--    ejecutar la orden;
--    reactivarse a si mismo
--    (excepto si la orden era "cierra")

sesionConColaTipo     :: (Read a, Show a) => a -> Cola a -> IO ()

sesionConColaTipo x c  = do putChar '\n'
                            putStr "Orden? "
                            orden <- getLine
                            ejecutaConColaTipo x c orden
```

```
-- Proceso auxiliar.
-- Depende de una cola y una orden dados como parametros.
-- Su efecto es:
--    ejecutar la orden, calculando si es preciso una nueva cola
--    llamar al proceso principal, con la nueva cola.
-- Caso especial:
--    La orden "desactiva" se ejecuta terminando la interaccion.


ejecutaConColaTipo               :: (Read a, Show a) => a -> Cola a -> Orden -> IO ()

ejecutaConColaTipo x c "ayuda"    = do putChar '\n'
                                       putStrLn menuCola
                                       sesionConColaTipo x c

ejecutaConColaTipo x c "vacia"     = do putChar '\n'
                                        putStrLn "Se ha vaciado la cola!"
                                        sesionConColaTipo x vacia

ejecutaConColaTipo x c "mete"      = do putChar '\n'
                                        putStr "Elemento a meter? "
                                        e <- readLn
                                        sesionConColaTipo x (mete c e)

ejecutaConColaTipo x c "resto"     = do putChar '\n'
                                        if esVacia c
                                           then do putStrLn "Error: resto vacia!"
                                                   sesionConColaTipo x c
                                           else sesionConColaTipo x (resto c)

ejecutaConColaTipo x c "primero"   = do putChar '\n'
                                        if esVacia c
                                           then do putStrLn "Error: primero vacia!"
                                                   sesionConColaTipo x c
                                           else do print (primero c)
                                                   sesionConColaTipo x c

ejecutaConColaTipo x c "esVacia"   =  do putChar '\n'
                                         if esVacia c
                                            then putStrLn "La cola esta vacia."
                                            else putStrLn "La cola no esta vacia."
                                         sesionConColaTipo x c

ejecutaConColaTipo x c "contenido" =  do putChar '\n'
                                         imprimeCola (contenido c)
                                         sesionConColaTipo x c

ejecutaConColaTipo x c "desactiva" =  do putChar '\n'
                                         putStrLn despedidaCola
```

```
ejecutaConColaTipo x c  _          =  do putChar '\n'
                                         putStrLn "Orden desconocida!"
                                         sesionConColaTipo x c

-- Proceso auxiliar que muestra el contenido de una cola:

imprimeCola    :: Show a => [a] -> IO ()
imprimeCola xs  = do putStr "primero -> "
                     imprimeElementos xs
                     putStrLn " <- ultimo"

imprimeElementos        :: Show a => [a] -> IO ()
imprimeElementos []      = return ()
imprimeElementos (x:xs)  = do putStr siguiente
                              imprimeElementos xs
                              where siguiente = if null xs
                                                then show x
                                                else show x ++ ", "


{-- Pruebas de activacion de colas de diferentes tipos:

ActivaCola> activaColaTipo (anything :: Bool)

ActivaCola> activaColaTipo (anything :: Int)

ActivaCola> activaColaTipo (anything :: Float)

ActivaCola> activaColaTipo (anything :: Char)

ActivaCola> activaColaTipo (anything :: String)

ActivaCola> activaColaTipo (anything :: (In,Int))

ActivaCola> activaColaTipo (anything :: (Bool,(Int,Int))

ActivaCola> activaColaTipo (anything :: Either Int Bool)

--}
```

A.29 Listas ordenadas

```
--                          Las listas ordenados como TAD

module Listord (Listord, nula, inserta, elimina, contiene, esNula,
                contenido, construye) where

-- El tipo representante es [a].
```

```
-- Representa listas ordenadas de elementos de un tipo con orden.

newtype Ord a => Listord a = LO [a]

-- El invariante de la representacion exige una lista ordenada
-- en orden creciente, sin elementos repetidos:
--
-- valido        :: Ord a => Listord a -> Bool
-- valido (LO xs) = esOrdenada xs

-- La equivalencia abstracta es trivial:
--
-- equiv                 :: Ord a => Listord a -> Listord a -> Bool
-- equiv (LO xs) (LO ys)  = (xs = ys)


-- Creacion de una lista ordenada vacia.

nula :: Ord a => Listord a
nula  = LO []


-- Insercion de un elemento en una lista ordenada.

inserta           :: Ord a => Listord a -> a -> Listord a
inserta (LO xs) x  = LO (insOrd xs x)
                     where insOrd []     x = [x]
                           insOrd (y:ys) x
                             | x  < y      = x:y:ys
                             | x == y      = y:ys
                             | x  > y      = y:insOrd ys x


-- Eliminacion de un elemento de una lista ordenada.

elimina           :: Ord a => Listord a -> a -> Listord a
elimina (LO xs) x  = LO (elimOrd xs x)
                     where elimOrd []     x = []
                           elimOrd (y:ys) x
                             | x  < y      = y:ys
                             | x == y      = ys
                             | x  > y      = y:elimOrd ys x


-- Pertenencia de un elemento a una lista ordenada.

contiene           :: Ord a => Listord a -> a -> Bool
contiene (LO xs) x  = estaEn xs x
                      where estaEn []     x = False
```

```
                        estaEn (y:ys) x
                          | x  < y      = False
                          | x == y      = True
                          | x  > y      = estaEn ys x


-- Reconocimiento de  la lista vacia.

esNula         :: Ord a => Listord a -> Bool
esNula (LO xs)  = null xs


-- Contenido de una lista ordenada.

contenido         :: Ord a => Listord a -> [a]
contenido (LO xs)  = xs

-- Construccion de una lista ordenada a partir de una lista
-- de datos ordenada en orden creciente.
-- Debe usarse con prudencia.
-- Si xs no estuviese ordenada,
-- (LO xs) violaria el invariante de la representacion.

construye   :: Ord a => [a] -> Listord a
construye xs = LO xs


{-- Pruebas de ejecucion:

Listord> contenido (inserta (inserta (inserta nula 5) 9) 2)
[2,5,9]
Listord> contiene (inserta (inserta (inserta nula 5) 9) 2) 4
False
Listord> contenido (elimina (inserta (inserta (inserta nula 5) 9) 2) 9)
[2,5]

--}
```

A.30 Listas de búsqueda

```
--             Las listas de busqueda ordenadas por claves como TAD

module Listbus (Listbus, nula, inserta, inserta, insertaCon, borra, consulta,
                esNula, contenido, construye) where

-- El tipo representante es [(c,v)].
-- Representa listas de busqueda con claves de tipo c (de la clase Ord)
-- y valores sociados de tipo v.
```

```
newtype Ord c => Listbus c v = LB [(c,v)]

-- El invariante de la representacion exige una lista ordenada
-- en orden creciente de claves, sin claves repetidas:
--
-- valido          :: Ord c => Listbus c v -> Bool
-- valido (LB cvs)  = esOrdenada [cla | (cla,val) <- cvs]

-- La equivalencia abstracta es trivial:
--
-- equiv                    :: Ord c => Listbus c v -> Listbus c v -> Bool
-- equiv (LA cvs) (LO cvs')  = (cvs = cvs')


-- Creacion de una lista de busqueda vacia.

nula :: Ord c => Listbus c v
nula  = LB []


-- Insercion en una lista de busqueda.
-- Si la clave ya estaba, el nuevo valor insertado reemplaza al viejo.

inserta :: Ord c => Listbus c v -> c -> v -> Listbus c v
inserta  = insertaCon combi
           where combi viejo nuevo = nuevo

-- Insercion en una lista de busqueda, con funcion de combinacion.
-- (insertaCon combi lis cla val) devuelve la lista de busqueda resultante de
-- insertar la clave cla con valor asociado val en la lista de busqueda lis.
-- Si cla ya aparecia en las con valor asociado val', el nuevo valor asociado
-- sera (combi val' val).

insertaCon  :: Ord c => (v -> v -> v) -> Listbus c v -> c -> v -> Listbus c v
insertaCon combi (LB cvs) cla val = LB (insOrdCon combi cvs cla val)

insOrdCon    :: Ord c => (v -> v -> v) -> [(c,v)] -> c -> v -> [(c,v)]
insOrdCon combi []                cla val = [(cla,val)]
insOrdCon combi ((cla',val'):cvs) cla val
   | cla  < cla'                          = (cla,val):(cla',val'):cvs
   | cla == cla'                          = (cla',combi val' val):cvs
   | cla  > cla'                          = (cla',val'):insOrdCon combi cvs cla val



-- Borrado en una lista de busqueda.

borra              :: Ord c => Listbus c v -> c -> Listbus c v
borra (LB cvs) cla  = LB (elimina cvs cla)
```

```
elimina           :: Ord c => [(c,v)] -> c -> [(c,v)]
elimina []                cla = []
elimina ((cla',val'):cvs) cla
   | cla  < cla'                          = (cla',val'):cvs
   | cla == cla'                          = cvs
   | cla  > cla'                          = (cla',val'):elimina cvs cla


-- Consulta en una lista de busqueda.

consulta              :: Ord c => Listbus c v -> c -> Maybe v
consulta (LB cvs) cla  = buscaEn cvs cla

buscaEn                       :: Ord c => [(c,v)] -> c -> Maybe v
buscaEn []               cla   = Nothing
buscaEn ((cla',val):cvs) cla
   | cla  < cla'                 = Nothing
   | cla == cla'                 = Just val
   | cla  > cla'                 = buscaEn cvs cla



-- Reconocimiento de  la lista de busqueda vacia.

esNula          :: Ord c => Listbus c v -> Bool
esNula (LB cvs)  = null cvs


-- Contenido de una lista de busqueda.

contenido          :: Ord c => Listbus c v -> [(c,v)]
contenido (LB cvs)  = cvs

-- Construccion de un arbol ordenado a partir de una lista de parejas
-- (clave, valor) ordenada en orden creciente de claves.
-- Debe usarse con prudencia.
-- Si cvs no estuviese ordenada con respecto a las claves,
-- (LB cvs) violaria el invariante de la representacion.

construye     :: Ord c => [(c,v)] -> Listbus c v
construye cvs  = LB cvs

{-- Pruebas de ejecucion:

Listbus> contenido
(insertaCon (++) (insertaCon (++) (insertaCon (++) nula "m" "M") "c" "C") "m" "M")

[("c","C"),("m","MM")]
```

```
Listbus> consulta
(insertaCon (++) (insertaCon (++) (insertaCon (++) nula "m" "M") "c" "C") "m" "M") "m"

Just "MM"
Listbus> contenido (borra
(insertaCon (++) (insertaCon (++) (insertaCon (++) nula "m" "M") "c" "C") "m" "M") "m")

[("c","C")]

--}
```

A.31 Ejemplos de uso de vectores en Haskell

```
module PruebaArray where

   import Array

-- Vector de los cuadrados de los enteros entre l y u:

   cuadrados       :: (Int,Int) -> Array Int Int
   cuadrados (l,u)  = array (l,u) [(i,i*i) | i <- [l..u]]

   cuadrados'       :: (Int,Int) -> Array Int Int
   cuadrados' (l,u)  = listArray (l,u) [i*i | i <- [l..u]]

{-- Pruebas:

PruebaArray> cuadrados (2,8)
array (2,8) [(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]

PruebaArray> cuadrados' (2,8)
array (2,8) [(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64)]

--}

-- Histograma:

   histograma                 :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
   histograma limites indices  = accumArray (+) 0
                                            limites
                                            [(i,1) | i <- indices,
                                                     inRange limites i]

{-- Prueba:

PruebaArray> histograma (3,5) [i | i <- [1..7], j <- [3..i]]
array (3,5) [(3,1),(4,2),(5,3)]

--}
```

```
-- Mensaje de error, que se usara mas abajo:

   errorProdMat :: String
   errorProdMat  = "ProdMat: producto de matrices con dimensiones incompatibles!"

-- Producto de matrices, usando "array":

   prodMat    :: (Ix a, Ix b, Ix c, Num d) =>
                 Array (a,b) d -> Array (b,c) d -> Array (a,c) d

   prodMat x y = array resultBounds
                     [((i,j), sum [x!(i,k) * y!(k,j) | k <- range (lj,uj)])
                         | i <- range (li,ui),
                           j <- range (lj',uj')                            ]
                 where ((li,lj),(ui,uj))       = bounds x
                       ((li',lj'),(ui',uj'))   = bounds y
                       resultBounds
                        | (lj,uj) == (li',ui') = ((li,lj'),(ui,uj'))
                        | otherwise            = error errorProdMat

-- Producto de matrices, usando "accumArray":

   prodMat'    :: (Ix a, Ix b, Ix c, Num d) =>
                  Array (a,b) d -> Array (b,c) d -> Array (a,c) d

   prodMat' x y = accumArray (+) 0 resultBounds
                     [((i,j), x!(i,k) * y!(k,j))
                         | i <- range (li,ui),
                           j <- range (lj',uj'),
                           k <- range (lj,uj)     ]
                 where ((li,lj),(ui,uj))       = bounds x
                       ((li',lj'),(ui',uj'))   = bounds y
                       resultBounds
                        | (lj,uj) == (li',ui') = ((li,lj'),(ui,uj'))
                        | otherwise            = error errorProdMat


-- Producto de matrices generalizado, usando "array" y
-- reemplazando "sum" y "(*)" por parametros:

   prodMatGen                :: (Ix a, Ix b, Ix c, Num d) =>
                                ([f] -> g) -> (d -> e -> f) ->
                                Array (a,b) d -> Array (b,c) e -> Array (a,c) g

   prodMatGen sumList prod x y = array resultBounds
                     [((i,j), sumList [(x!(i,k)) `prod` (y!(k,j)) | k <- range (lj,uj)])
                         | i <- range (li,ui),
                           j <- range (lj',uj')                            ]
```

```
                                where ((li,lj),(ui,uj))       = bounds x
                                      ((li',lj'),(ui',uj'))   = bounds y
                                      resultBounds
                                       | (lj,uj) == (li',ui') = ((li,lj'),(ui,uj'))
                                       | otherwise            = error errorProdMat


-- "prodMat" es un caso particular de "prodMatGen":
--
--     prodMat = prodMatGen sum (*)
--
-- Otros usos posibles de "prodMatGen":
--
--     prodMatGen maximum (-)
--     prodMatGen and (==)
--
-- Piensese en el comportamiento de estas dos funciones!


-- Dos matrices enanas, para pruebas:

-- m es una matriz de enteros de dimension 2 x 3:
--
--      2  3  4
--      3  4  5

   m :: Array (Int,Int) Int
   m  = array ((1,1),(2,3)) [((i,j),i+j) | i <- [1..2], j <- [1..3]]

-- n es una matriz de enteros de dimension 3 x 2:
--
--      1  2
--      2  4
--      3  6

   n :: Array (Int,Int) Int
   n  = array ((1,1),(3,2)) [((i,j),i*j) | i <- [1..3], j <- [1..2]]

-- Para el producto de m y n se espera una matriz 2 x 2:
--
--     20  40
--     26  52

{-- Las pruebas lo confirman:

PruebaArray> m
array ((1,1),(2,3)) [((1,1),2),((1,2),3),((1,3),4),((2,1),3),((2,2),4),((2,3),5)]

PruebaArray> n
```

```
array ((1,1),(3,2)) [((1,1),1),((1,2),2),((2,1),2),((2,2),4),((3,1),3),((3,2),6)]

PruebaArray> prodMat m n
array ((1,1),(2,2)) [((1,1),20),((1,2),40),((2,1),26),((2,2),52)]

PruebaArray> prodMat' m n
array ((1,1),(2,2)) [((1,1),20),((1,2),40),((2,1),26),((2,2),52)]

PruebaArray> prodMatGen sum (*) m n
array ((1,1),(2,2)) [((1,1),20),((1,2),40),((2,1),26),((2,2),52)]

--}
```

A.32 Arboles binarios con información en los nodos internos

```
--                              Arboles binarios


module Arbin (Arbin(..), hijoIz, hijoDr, raiz, esVacio, preOrd, inOrd, postOrd) where


-- Tipo de datos (Arbin a) de los arboles binarios con datos de tipo a
-- en sus nodos internos. Construido y recursivo.

data Arbin a = Nulo | Bin (Arbin a) a (Arbin a)
             deriving (Eq, Ord, Read, Show)


-- Funciones basicas para consulta y modificacion de arboles binarios.

hijoIz, hijoDr      :: Arbin a -> Arbin a

hijoIz (Bin iz x dr) = iz

hijoDr (Bin iz x dr) = dr

raiz              :: Arbin a -> a

raiz (Bin iz x dr) = x

esVacio              :: Arbin a -> Bool

esVacio Nulo          = True
esVacio (Bin iz x dr) = False


-- Funciones para recorrido de arboles binarios.
```

```
-- Recorrido en preorden.

preOrd :: Arbin a -> [a]

-- Especificacion ejecutable:
--
-- preOrd Nulo          = []
-- preord (Bin iz x dr) = [x] ++ preOrd iz ++ preOrd dr

-- Version optimizada, mas eficiente.
-- Usa una funcion auxiliar ponPreOrd, tal que
--
--           ponPreOrd arbin xs = preOrd arbin ++ xs

preOrd arbin = ponPreOrd arbin []
             where ponPreOrd Nulo          = id
                   ponPreOrd (Bin iz x dr) = (x:) . ponPreOrd iz . ponPreOrd dr


-- Recorrido en postorden.

postOrd :: Arbin a -> [a]

-- Especificacion ejecutable:
--
-- postOrd Nulo          = []
-- postOrd (Bin iz x dr) = postOrd iz ++ postOrd dr ++ [x]

-- Version optimizada, mas eficiente.
-- Usa una funcion auxiliar ponPostOrd, tal que
--
--           ponPostOrd arbin xs = postOrd arbin ++ xs

postOrd arbin = ponPostOrd arbin []
             where ponPostOrd Nulo          = id
                   ponPostOrd (Bin iz x dr) = ponPostOrd iz . ponPostOrd dr . (x:)


-- Recorrido en inorden.

inOrd :: Arbin a -> [a]

-- Especificacion ejecutable:
--
-- inOrd Nulo          = []
-- inOrd (Bin iz x dr) = inOrd iz ++ [x] ++ inOrd dr

-- Version optimizada, mas eficiente.
```

```
-- Usa una funcion auxiliar ponInOrd, tal que
--
--           ponInOrd arbin xs = inOrd arbin ++ xs

inOrd arbin = ponInOrd arbin []
              where ponInOrd Nulo          = id
                    ponInOrd (Bin iz x dr) = ponInOrd iz . (x:) . ponInOrd dr
```

A.33 Prueba de los árboles binarios con información en los nodos internos

```
--                            Prueba de los arboles binarios

module PruebaArbin where

-- Importacion de los arboles binarios:

import Arbin


-- Arbol binario de enteros, para pruebas:

arbin :: Arbin Int
arbin  = Bin (Bin Nulo
                  1
                  (Bin Nulo
                       2
                       Nulo
                   )
              )
              3
              (Bin (Bin Nulo
                        4
                        Nulo
                    )
                    5
                    Nulo
               )


{-- Algunas pruebas:

PruebaArbin> arbin
Bin (Bin Nulo 1 (Bin Nulo 2 Nulo)) 3 (Bin (Bin Nulo 4 Nulo) 5 Nulo)

PruebaArbin> esVacio arbin
False
```

```
PruebaArbin> raiz arbin
3

PruebaArbin> hijoIz arbin
Bin Nulo 1 (Bin Nulo 2 Nulo)

PruebaArbin> hijoDr arbin
Bin (Bin Nulo 4 Nulo) 5 Nulo

PruebaArbin> preOrd arbin
[3,1,2,5,4]

PruebaArbin> postOrd arbin
[2,1,4,5,3]

PruebaArbin> inOrd arbin
[1,2,3,4,5]

--}
```

A.34 Arboles binarios con información en las hojas

```
--                  Arboles binarios con datos en las hojas


module ArbinH (ArbinH(..), hi, hd, info, esHoja, front) where


-- Tipo de datos (ArbinH a) de los arboles binarios con datos de tipo a
-- en sus hojas. Construido y recursivo.

data ArbinH a = Hoja a | Nodo (ArbinH a) (ArbinH a)
              deriving (Eq, Ord, Read, Show)


-- Funciones basicas para consulta y modificacion de arboles binarios.

hi, hd         :: ArbinH a -> ArbinH a

hi (Nodo iz dr) = iz

hd (Nodo iz dr) = dr

info          :: ArbinH a -> a

info (Hoja x) = x
```

```
esHoja              :: ArbinH a -> Bool

esHoja (Hoja x)     = True
esHoja (Nodo iz dr) = False


-- Funcion que calcula la frontera de un arbol de este tipo.

front :: ArbinH a -> [a]

-- Especificacion ejecutable:
--
-- front (Hoja x)     = [x]
-- front (Nodo iz dr) = front iz ++ front dr

-- Version optimizada, mas eficiente.
-- Usa una funcion auxiliar ponFront, tal que
--
--           ponFront arbinH xs = front arbinH ++ xs

front arbinH = ponFront arbinH []
             where ponFront (Hoja x)     = (x:)
                   ponFront (Nodo iz dr) = ponFront iz . ponFront dr
```

A.35 Prueba de los árboles binarios con información en las hojas

```
--                Prueba de los arboles binarios con datos en las hojas

module PruebaArbinH where

-- Importacion de los arboles binarios con datos en las hojas:

import ArbinH


-- Arbol binario con enteros en las hojas, para pruebas:

arbinH :: ArbinH Int
arbinH  = Nodo (Nodo (Hoja 1)
                     (Hoja 2)
               )
               (Nodo (Hoja 3)
                     (Hoja 4)
               )
```

```
{-- Algunas pruebas:

PruebaArbinH> arbinH
Nodo (Nodo (Hoja 1) (Hoja 2)) (Nodo (Hoja 3) (Hoja 4))

PruebaArbinH> esHoja arbinH
False

PruebaArbinH> hi arbinH
Nodo (Hoja 1) (Hoja 2)

PruebaArbinH> hd arbinH
Nodo (Hoja 3) (Hoja 4)

PruebaArbinH> front arbinH
[1,2,3,4]

--}
```

A.36 Arboles binarios calibrados con información en las hojas

```
--                  Arboles binarios calibrados con datos en las hojas


module ArbinHC (ArbinHC(..), hiC, hdC, dato, peso, esHojaC, nodoC, descarta) where


-- Importacion de los arboles binarios con datos en las hojas.

import ArbinH


-- Tipo de datos (ArbinHC a) de los arboles binarios calibrados
-- con datos de tipo a en sus hojas. Construido y recursivo.

data ArbinHC a = HojaC Int a | NodoC Int (ArbinHC a) (ArbinHC a)
              deriving (Eq, Ord, Read, Show)


-- Funciones basicas para consulta y modificacion de arboles binarios.

hiC, hdC           :: ArbinHC a -> ArbinHC a

hiC (NodoC p iz dr) = iz

hdC (NodoC p iz dr) = dr
```

```
dato         :: ArbinHC a -> a

dato (HojaC p x) = x

peso                :: ArbinHC a -> Int

peso (HojaC p x)     = p
peso (NodoC p iz dr) = p

esHojaC               :: ArbinHC a -> Bool

esHojaC (HojaC p x)     = True
esHojaC (NodoC p iz dr) = False


-- Funcion que construye un arbol de este tipo a partir de sus dos hijos,
-- tomando como peso la suma de los pesos.

nodoC      :: ArbinHC a -> ArbinHC a ->ArbinHC a

nodoC iz dr = NodoC p iz dr
            where p = peso iz + peso dr


-- Funcion que descarta los pesos de un arbol de este tipo.

descarta               :: ArbinHC a -> ArbinH a

descarta (HojaC p x)     = Hoja x
descarta (NodoC p iz dr) = Nodo (descarta iz) (descarta dr)
```

A.37 Prueba de los árboles binarios calibrados con información en las hojas

```
--             Prueba de los arboles binarios calibrados con datos en las hojas


module PruebaArbinHC where

-- Importacion de los arboles binarios calibrados con datos en las hojas:

import ArbinHC


-- Arbol binario calibrado con caracteres en las hojas, para pruebas:

arbinHC :: ArbinHC Char
arbinHC  = NodoC 58
```

```
                (NodoC 24
                      (HojaC 11 'A')
                      (HojaC 13 'T')
                 )
                 (NodoC 34
                      (NodoC 17
                            (HojaC 8 'G')
                            (HojaC 9 'R')
                       )
                       (HojaC 17 'E')
                  )


{-- Algunas pruebas:

PruebaArbinHC> arbinHC
NodoC 58 (NodoC 24 (HojaC 11 'A') (HojaC 13 'T'))
         (NodoC 34 (NodoC 17 (HojaC 8 'G') (HojaC 9 'R')) (HojaC 17 'E'))

PruebaArbinHC> esHojaC arbinHC
False

PruebaArbinHC> hiC arbinHC
NodoC 24 (HojaC 11 'A') (HojaC 13 'T')

PruebaArbinHC> hdC arbinHC
NodoC 34 (NodoC 17 (HojaC 8 'G') (HojaC 9 'R')) (HojaC 17 'E')

PruebaArbinHC> descarta arbinHC
Nodo (Nodo (Hoja 'A') (Hoja 'T')) (Nodo (Nodo (Hoja 'G') (Hoja 'R')) (Hoja 'E'))

--}
```

A.38 Arboles de codificación de Huffman

```
--                      Arboles de codificacion de Huffman


module Huffman where

-- Importacion de los arboles binarios con datos en las hojas.

import ArbinH


-- Importacion de los arboles binarios calibrados con datos en las hojas.

import ArbinHC
```

```
-- Importacion de algoritmos de ordenacion parametrizados por el orden.

import OrdenaCon


-- Tipos de datos para representar los textos y sus codigos.

type Texto = String

data Bit = O | I
         deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

type Codigo = [Bit]


-- Tipo de datos para representar los arboles de codificacion.
-- El camino  desde la raiz a cada hoja representa el codigo
-- del caracter almacenado en esa hoja.

type ArbolCod = ArbinH Char


-- Funcion decod que decodifica un texto dado.
-- Usa la funcion auxiliar decod1, que decodifica un caracter.

decod        :: ArbolCod -> Codigo -> Texto
decod ac bits = if null bits
                  then []
                  else x : decod ac restoBits
              where (x, restoBits) = decod1 ac bits

-- Funcion auxiliar decod1.
-- (decod1 ac bits) calcula una pareja (x, restoBits),
-- donde x es el caracter codificado por el prefijo mas corto de bits que sea un codigo,
-- y restoBits es el resto de bits que queden al quitar ese prefijo.

decod1                          :: ArbolCod -> Codigo -> (Char, Codigo)
decod1 (Hoja x)     bits        = (x, bits)
decod1 (Nodo iz dr) (O : bits) = decod1 iz bits
decod1 (Nodo iz dr) (I : bits) = decod1 dr bits



-- Funcion cod que codifica un texto dado.
-- Usa una funcion auxiliar cod1, que codifica un caracter.

cod            :: ArbolCod -> Texto -> Codigo
cod ac []       = []
```

```
cod ac (x:xs)   = cod1 ac x ++ cod ac xs

-- Funcion auxiliar cod1.
-- Para codificar el caracter x con el arbol de codificacion ac,
-- buscamos el camino desde la raiz de ac hasta una hoja que contenga x.
-- La funcion auxiliar caminos calcula la una lista cuyos elementos son
-- todos los caminos desde la raiz de ac hasta las hojas que contengan x.
-- A la postre, esperamos que esta lista sea unitaria.

cod1     :: ArbolCod -> Char -> Codigo
cod1 ac x = head (caminos ac x)

-- Funcion auxiliar caminos. Explicada mas arriba.
-- Ilustra una solucion a un problema de busqueda tipico.

caminos               :: ArbolCod -> Char -> [Codigo]
caminos (Hoja y)     x = if y == x
                           then [[]]  -- un solo camino a x, que es []
                           else []    -- ningun camino a x
caminos (Nodo iz dr) x = [ 0 : bits | bits <- caminos iz x]
                                ++
                         [ 1 : bits | bits <- caminos dr x]



-- Funcion huffman que construye un arbol de codificacion,
-- a partir de una lista de la forma [ ... (ci,qi) ... ]
-- donde cada ci es un caracter y qi es el numero de veces
-- que ci aparece en el texto que se va a codificar.
--
-- Suponemos que la lista esta ordenada en orden no decreciente
-- con respecto a los qi, y que no contiene caracteres repetidos.
--
-- La funcion huffman usa el siguiente algoritmo:
--
--    a) La lista dada se convierte en una lista de arboles
--       calibrados de caracteres, cada uno formado por una sola hoja.
--    b) Se reitera el proceso de juntar los dos primeros arboles de la
--       lista en uno solo, hasta que la lista quede unitaria.
--    c) Se extrae el unico arbol que queda en la lista, y se descartan sus pesos.
--
-- Los arboles de codificacion construidos de esta manera dan lugar a
-- codigos de longitud minima, porque los codigos de los caracteres de
-- uso mas frecuente son mas cortos.

type DatosTexto = [(Char,Int)]


huffman :: DatosTexto -> ArbolCod
```

```
huffman  = descarta . head . transforma . inicializa

inicializa           :: DatosTexto -> [ArbinHC Char]
inicializa datosTexto = [ HojaC q c | (c,q) <- datosTexto ]

transforma :: [ArbinHC Char] -> [ArbinHC Char]
transforma  = until unitaria juntaDos

-- Funcion de reiteracion. Definida en el preludio.
--
-- until          :: (a -> Bool) -> (a -> a) -> a -> a
-- until p f x     = if p x
--                      then x
--                      else until p f (f x)

-- Funcion que reconoce si una lista es unitaria.

unitaria         :: [a] -> Bool
unitaria []       = False
unitaria [x]      = True
unitaria (x:y:zs) = False

-- Funcion que junta los dos primero arboles de la lista en uno solo,
-- e inserta el nuevo arbol en su lugar.

juntaDos :: [ArbinHC a] -> [ArbinHC a]
juntaDos (iz: dr : resto) = insOrdCon migPeso (nodoC iz dr) resto
                          where migPeso arb arb' = peso arb <= peso arb'



-- Funcion que calcula a partir de un texto los datos requeridos por
-- el algoritmo de Huffman.

extraeDatos :: Texto -> DatosTexto
extraeDatos texto = elimReps (ordRapidoCon mig [(x, frecuencia x) | x <- texto])
                  where frecuencia x    = length [y | y <- texto, y == x]
                        mig (x,m) (y,n) = m < n || (m == n && x <= y)

-- Funcion auxiliar.
-- (elimReps xs) calcula el resultado de eliminar repeticiones de
-- elementos iguales consecutivos en la lista xs.
-- Por ejemplo: elimReps [2,1,1,5,3,3,3,1,1,7,2,2] = [2,1,5,3,1,7,2].
-- En el caso de que xs estuviese ordenada, (elimReps xs) no contiene repeticiones.


elimReps       :: Eq a => [a] -> [a]
elimReps []     = []
elimReps (x:xs) = maybeCons x (elimReps xs)
```

```
                where maybeCons x []     = x:[]
                      maybeCons x (y:ys) = if x == y then y:ys else x:y:ys


-- Codificacion de un texto, calculando previamente el arbol de codificacion optimo.

codifica      :: Texto -> Codigo
codifica texto = cod ac texto
               where ac = arbolCodOptimo texto

arbolCodOptimo :: Texto -> ArbolCod
arbolCodOptimo  = huffman . extraeDatos



-- Funcion auxiliar de comprobacion
-- Codifica un texto y decodifica el codigo resultante, recuperando el texto inicial.

comprueba       :: Texto -> Texto
comprueba texto  = decod ac (cod ac texto)
                 where ac = arbolCodOptimo texto


-- Ejemplos para pruebas.

textoChico :: Texto
textoChico  = "SABROSA"

datosChicos :: DatosTexto
datosChicos = [('B',1),('O',1),('R',1),('A',2),('S',2)]

codigoChico :: Codigo
codigoChico  = [I,I,I,O,O,I,O,O,O,O,I,I,I,I,I,O]

arbolito :: ArbolCod
arbolito = Nodo (Nodo (Hoja 'R')
                      ( Nodo (Hoja 'B')
                             (Hoja 'O')
                      )
                 )
                 (Nodo (Hoja 'A')
                       (Hoja 'S')
                 )

texto :: Texto
texto  = "Los arboles de codificacion de Huffman consiguen " ++
         "codigos optimos de longitud variable"

{-- Pruebas:
```

```
Huffman> textoChico
"SABROSA"

Huffman> extraeDatos textoChico
[('B',1),('O',1),('R',1),('A',2),('S',2)]

Huffman> datosChicos
[('B',1),('O',1),('R',1),('A',2),('S',2)]

Huffman> arbolCodOptimo textoChico
Nodo (Nodo (Hoja 'R') (Nodo (Hoja 'B') (Hoja 'O'))) (Nodo (Hoja 'A') (Hoja 'S'))

Huffman> arbolito
Nodo (Nodo (Hoja 'R') (Nodo (Hoja 'B') (Hoja 'O'))) (Nodo (Hoja 'A') (Hoja 'S'))

Huffman> codifica textoChico
[I,I,I,O,O,I,O,O,O,O,I,I,I,I,I,O]

Huffman> codigoChico
[I,I,I,O,O,I,O,O,O,O,I,I,I,I,I,O]

Huffman> decod arbolito codigoChico
"SABROSA"

Huffman> comprueba textoChico
"SABROSA"

Huffman> texto
"Los arboles de codificacion de Huffman consiguen codigos optimos de longitud variable"

Huffman> codifica texto
[O,O,I,O,I,I,O,I,I,I,O,O,O,I,O,I,O,O,I,I,I,I,I,I,O,O,I,I,I,I,I,O,O,I,I,I,O,O,I,O,I,I,I,
O,I,O,O,O,I,O,I,I,I,O,I,I,I,I,O,I,O,I,O,I,O,O,O,I,I,I,I,O,I,O,O,O,I,I,O,O,O,O,O,O,O,I,O,
O,O,O,I,I,O,I,O,O,O,O,O,O,I,I,O,I,O,I,I,O,I,I,I,O,I,I,I,I,O,I,O,I,O,O,I,O,I,O,I,O,O,I,I,
I,I,O,O,O,I,I,O,O,O,I,I,I,I,I,I,O,O,I,I,O,I,O,I,I,O,I,O,I,O,O,O,I,I,O,I,O,I,I,O,O,O,O,O,
O,I,I,O,O,I,I,O,O,I,I,I,I,I,O,O,I,O,I,I,O,I,O,I,O,O,O,I,I,I,I,O,I,O,O,O,I,I,O,O,I,O,I,I,
I,O,O,O,I,O,I,O,I,I,O,O,I,O,O,O,I,I,I,I,O,I,O,O,O,I,I,I,I,I,I,O,I,I,I,O,O,O,I,O,I,I,I,O,
I,I,I,I,O,I,O,I,I,O,O,I,O,O,I,I,O,I,O,I,I,I,O,O,I,O,O,O,I,I,I,I,O,I,I,O,O,I,I,I,I,O,I,I,
O,I,O,O,I,O,O,I,O,O,I,I,I,I,I,I,O,O,O,O,O,O,O,I,I,I,I,I,I,I,O,I,O,O,I,O,I,I,I,O]

Huffman> comprueba texto
"Los arboles de codificacion de Huffman consiguen codigos optimos de longitud variable"

Huffman> comprueba "Arre borriquillo, arre burro arre"
"Arre borriquillo, arre burro arre"

--}
```

A.39 Arboles generales

```
--                              Arboles generales


module Arbol (Arbol(..), Bosque, infoRaiz, hijos, tieneHijos, preOrden, postOrden) where


-- Tipo de datos (Arbol a) de los arboles generales con datos de tipo a
-- en sus nodos. Construido y recursivo.

data Arbol a = Planta a [Arbol a]
             deriving (Eq, Ord, Read, Show)


-- Tipo sinonimo correspondiente a bosques de arboles generales.

type Bosque a = [Arbol a]


-- Funciones basicas para consulta y modificacion de arboles generales.

infoRaiz                      :: Arbol a -> a
infoRaiz (Planta dato bosque) = dato

hijos                       :: Arbol a -> Bosque a
hijos (Planta dato bosque) = bosque

tieneHijos                        :: Arbol a -> Bool
tieneHijos (Planta dato bosque) = not (null bosque)


-- Funciones para recorrido de arboles generales.

-- Recorrido en preorden.

preOrden :: Arbol a -> [a]

-- Especificacion ejecutable:
--
-- preOrden (Planta dato bosque) = [dato] ++ [x | hijo <- bosque, x <- preOrden hijo]


-- Version optimizada, mas eficiente.
-- Usa una funcion auxiliar ponPreOrden, tal que
--
--           ponPreOrden arbol lista = preOrden arbol ++ lista

preOrden arbol = ponPreOrden arbol []
```

```
                where ponPreOrden (Planta dato bosque) =
                        (dato :) . acumula ponPreOrden bosque


-- Recorrido en postorden.

postOrden :: Arbol a -> [a]

-- Especificacion ejecutable:
--
-- postOrden (Planta dato bosque) = [x | hijo <- bosque, x <- postOrden hijo] ++ [dato]


-- Version optimizada, mas eficiente.
-- Usa una funcion auxiliar ponPostOrden, tal que
--
--           ponPostOrden arbol lista = postOrden arbol ++ lista

postOrden arbol = ponPostOrden arbol []
                where ponPostOrden (Planta dato bosque) =
                        acumula ponPostOrden bosque . (dato :)



-- Funcion auxiliar de acumulacion.

acumula         :: (a -> b -> b) -> [a] -> b -> b
acumula f []     = id
acumula f (x:xs) = f x . acumula f xs
```

A.40 Prueba de los árboles generales

```
--                          Prueba de los arboles generales

module PruebaArbol where

-- Importacion de los arboles generales:

import Arbol


-- Arbol general de enteros, para pruebas:

arbol :: Arbol Int
arbol  = Planta 1 [Planta 2 [],
                   Planta 3 [Planta 4 [],
                             Planta 5 []
                             ],
                   Planta 6 []
```

```
                  ]

{-- Algunas pruebas:

PruebaArbol> arbol
Planta 1 [Planta 2 [],Planta 3 [Planta 4 [],Planta 5 []],Planta 6 []]

PruebaArbol> tieneHijos arbol
True

PruebaArbol> hijos arbol
[Planta 2 [],Planta 3 [Planta 4 [],Planta 5 []],Planta 6 []]

PruebaArbol> infoRaiz arbol
1

PruebaArbol> preOrden arbol
[1,2,3,4,5,6]

PruebaArbol> postOrden arbol
[2,4,5,3,6,1]

--}
```

A.41 Arboles ordenados

```
--                          Los arboles ordenados como TAD

module Arbord (Arbord, nulo, pon, quita, esta, esNulo, recorre, construye) where

-- El tipo representante es construido y recursivo.
-- Representa arboles binarios que almacenan en sus nodos
-- internos datos de un tipo a de la clase Ord.

data (Ord a) => Arbord a = Nulo | Bin (Arbord a) a (Arbord a)

-- Invariante de la representacion:
-- valido      :: Ord a => Arbord a -> Bool
-- valido arbol = listaOrdenada (recorre arbol)

-- Equivalencia abstracta:
-- equiv             :: Ord a => Arbord a -> Arbord a -> Bool
-- equiv arbol arbol' = arbol == arbol'


-- Creacion de un arbol ordenado vacio.

nulo :: Ord a => Arbord a
```

```
nulo  = Nulo


-- Insercion de un dato en un arbol ordenado.

pon                 :: Ord a => Arbord a -> a -> Arbord a
pon Nulo          y  = Bin Nulo y Nulo
pon (Bin iz x dr) y
  | y <  x           = Bin (pon iz y) x dr
  | y == x           = Bin iz y dr
  | y >  x           = Bin iz x (pon dr y)


--  Borrado de un dato en un arbol ordenado.

quita                          :: Ord a => Arbord a -> a -> Arbord a
quita   Nulo                y  = Nulo
quita   arbol@(Bin iz x dr) y
  | y <  x                     = Bin (quita iz y) x dr
  | y == x                     = quitaRaiz arbol
  | y >  x                     = Bin iz x (quita dr y)

quitaRaiz              :: Ord a => Arbord a -> Arbord a
quitaRaiz (Bin iz x dr)
  | esNulo iz           = dr
  | esNulo dr           = iz
  | otherwise           = Bin iz minDr restDr
                        where (minDr, restDr) = separaMin dr

separaMin              :: Ord a => Arbord a -> (a, Arbord a)
separaMin (Bin iz x dr)
  | esNulo iz           = (x, dr)
  | otherwise           = (minIz, Bin restIz x dr)
                        where (minIz, restIz) = separaMin iz


-- Busqueda de un dato en un arbol ordenado.

esta                :: Ord a => Arbord a -> a -> Bool
esta Nulo          y  = False
esta (Bin iz x dr) y
  | y <  x           = esta iz y
  | y == x           = True
  | y >  x           = esta dr y


-- Reconocimiento del arbol vacio.

esNulo              :: Ord a => Arbord a -> Bool
```

```
esNulo  Nulo         = True
esNulo (Bin iz x dr) = False


-- Recorrido en inorden de un arbol ordenado.

recorre      :: Ord a => Arbord a -> [a]
recorre arbol = ponInOrd arbol []
              where ponInOrd Nulo          = id
                    ponInOrd (Bin iz x dr) = ponInOrd iz . (x:) . ponInOrd dr


-- Construccion de un arbol ordenado a partir de una lista
-- de datos ordenada en orden creciente,
-- de modo que el arbol tenga la menor profundidad posible, y su
-- recorrido en inorden sea la lista dada.
-- Debe usarse con prudencia.
-- Si xs no estuviese ordenada,
-- (construye xs) no seria un arbol ordenado.

construye   :: Ord a => [a] -> Arbord a
construye xs = if null xs
                  then Nulo
                  else Bin iz x dr
               where n          = length xs `div` 2
                     (us, x:vs) = splitAt n xs
                     iz         = construye us
                     dr         = construye vs
```

A.42 Prueba de los árboles ordenados

```
--                            Prueba de los arboles ordenados

module PruebaArbord where


-- Importacion de los arboles ordenados:

import Arbord


-- Construccion de un arbol ordenado, para pruebas:

arbol :: Arbord Int
arbol  = construye [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,49]

{-- Algunas pruebas:

PruebaArbord> arbol
```

```
ERROR: Cannot find "show" function for:
*** Expression : arbol
*** Of type    : Arbord Int

PruebaArbord> recorre arbol
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,49]

PruebaArbord> esta arbol 23
True

PruebaArbord> recorre (quita arbol 23)
[2,3,5,7,11,13,17,19,29,31,37,41,43,47,49]

PruebaArbord> let arbol' = quita arbol 23 in esta arbol' 23
False

PruebaArbord> let arbol' = quita arbol 23 in esta arbol' 47
True

PruebaArbord> esta arbol 25
False

PruebaArbord> recorre (pon arbol 25)
[2,3,5,7,11,13,17,19,23,25,29,31,37,41,43,47,49]

PruebaArbord> let arbol' = pon arbol 25 in esta arbol' 25
True

PruebaArbord> let arbol' = pon arbol 25 in esta arbol' 12
False

PruebaArbord> esNulo arbol
False

PruebaArbord> esNulo nulo
ERROR: Unresolved overloading
*** Type       : Ord a => Bool
*** Expression : esNulo nulo

PruebaArbord> esNulo (nulo :: Arbord Int)
True

--}
```

A.43 Arboles de búsqueda

```
--                  Los arboles de busqueda como TAD

module Arbus (Arbus, crea, inserta, insertaCon, borra, consulta, esNulo,
```

```
             recorre, construye) where

-- El tipo representante es construido y recursivo.
-- Representa arboles binarios que almacenan en cada nodo
-- interno una clave y un valor asociado.
-- El tipo de las claves debe ser de la clase Ord.

data (Ord c) => Arbus c v = Nulo | Bin (Arbus c v) c v (Arbus c v)

-- Invariante de la representacion:
-- valido      :: Ord c => Arbus c v -> Bool
-- valido arbol = listaOrdenadaRespectoClaves (recorre arbol)

-- Equivalencia abstracta:
-- equiv             :: Ord c => Arbus c v -> Arbus c v -> Bool
-- equiv arbol arbol' = arbol == arbol'


-- Creacion de un arbol de busqueda vacio.

crea :: Ord c => Arbus c v
crea  = Nulo


-- Insercion en un arbol de busqueda.
-- Si la clave ya estaba, el nuevo valor insertado reemplaza al viejo.

inserta :: Ord c => Arbus c v -> c -> v -> Arbus c v
inserta  = insertaCon combi
         where combi viejo nuevo = nuevo


-- Insercion en un arbol de busqueda, con funcion de combinacion.
-- (insertaCon combi arb cla val) devuelve el arbol resultante de
-- insertar la clave cla con valor asociado val en el arbol ordenado arb.
-- Si cla ya aparecia en arb con valor asociado val', el nuevo valor asociado
-- sera (combi val' val).

insertaCon                            :: Ord c => (v -> v -> v) -> Arbus c v ->
                                                  c -> v -> Arbus c v
insertaCon combi Nulo              c v = Bin Nulo c v Nulo
insertaCon combi (Bin iz c' v' dr) c v
  | c <  c'                             = Bin (insertaCon combi iz c v) c' v' dr
  | c == c'                             = Bin iz c' (combi v' v) dr
  | c >  c'                             = Bin iz c' v' (insertaCon combi dr c v)


--  Borrado de una clave en un arbol de busqueda.
```

```
borra                          :: Ord c => Arbus c v -> c -> Arbus c v
borra   Nulo                      c  = Nulo
borra   arbol@(Bin iz c' v'  dr) c
  | c <  c'                          = Bin (borra iz c) c' v' dr
  | c == c'                          = quitaRaiz arbol
  | c >  c'                          = Bin iz c' v' (borra dr c)

quitaRaiz                 :: Ord c => Arbus c v -> Arbus c v
quitaRaiz (Bin iz c v dr)
  | esNulo iz              = dr
  | esNulo dr              = iz
  | otherwise              = Bin iz minClaDr minValDr restDr
                           where (minClaDr, minValDr, restDr) = separaMin dr

separaMin                  :: Ord c => Arbus c v -> (c, v, Arbus c v)
separaMin (Bin iz c v dr)
  | esNulo iz               = (c, v, dr)
  | otherwise               = (minClaIz, minValIz, Bin restIz c v dr)
                            where (minClaIz, minValIz, restIz) = separaMin iz


-- Consulta de una clave en un arbol de busqueda.

consulta                    :: Ord c => Arbus c v -> c -> Maybe v
consulta Nulo              c  = Nothing
consulta (Bin iz c' v dr) c
  | c <  c'                  = consulta iz c
  | c == c'                  = Just v
  | c >  c'                  = consulta dr c


-- Reconocimiento del arbol vacio.

esNulo                :: Ord c => Arbus c v -> Bool
esNulo  Nulo           = True
esNulo (Bin iz c v dr) = False


-- Recorrido en inorden de un arbol de busqueda.

recorre       :: Ord c => Arbus c v ->  [(c,v)]
recorre arbol  = ponInOrd arbol []
                 where ponInOrd Nulo            = id
                       ponInOrd (Bin iz c v dr) = ponInOrd iz . ((c,v):) . ponInOrd dr


-- Construccion de un arbol ordenado a partir de una lista de parejas
-- (clave, valor) ordenada en orden creciente de claves,
-- de modo que el arbol tenga la menor profundidad posible, y su
```

```
-- recorrido en inorden sea la lista dada.
-- Debe usarse con prudencia.
-- Si xs no estuviese ordenada con respecto a las claves,
-- (construye xs) no seria un arbol de busqueda.

construye    :: Ord c => [(c,v)] -> Arbus c v
construye xs  = if null xs
                then Nulo
                else Bin iz c v dr
            where n              = length xs `div` 2
                  (us, (c,v):vs) = splitAt n xs
                  iz             = construye us
                  dr             = construye vs
```

A.44 Prueba de los árboles de búsqueda

```
--                            Prueba de los arboles de busqueda

module PruebaArbus where


-- Importacion de los arboles ordenados:

import Arbus


-- Arbol de busqueda, para pruebas:

arbus :: Arbus String String
arbus  = construye [("big", "grande"),
                    ("deft", "diestro"),
                    ("even", "llano"),
                    ("exercise", "ejercicio"),
                    ("heft", "mango"),
                    ("hell", "infierno"),
                    ("love", "amor"),
                    ("luck", "suerte"),
                    ("mad", "loco"),
                    ("pride", "orgullo"),
                    ("rest", "descanso"),
                    ("soft", "suave"),
                    ("tar", "brea"),
                    ("tender", "tierno")]


-- Funcion de combinacion, para pruebas:

combi             :: String -> String -> String
combi viejo nuevo  = viejo ++ "; " ++ nuevo
```

```
{-- Algunas pruebas:

PruebaArbus> arbus
ERROR: Cannot find "show" function for:
*** Expression : arbus
*** Of type    : Arbus String String

PruebaArbus> recorre arbus
[("big","grande"),("deft","diestro"),("even","llano"),("exercise","ejercicio"),
 ("heft","mango"),("hell","infierno"),("love","amor"),("luck","suerte"),
 ("mad","loco"),("pride","orgullo"),("rest","descanso"),("soft","suave"),
 ("tar","brea"),("tender","tierno")]

PruebaArbus> consulta arbus "exercise"
Just "ejercicio"

PruebaArbus> consulta arbus "execute"
Nothing

PruebaArbus> recorre (borra arbus "exercise")
[("big","grande"),("deft","diestro"),("even","llano"),("heft","mango"),
("hell","infierno"),("love","amor"),("luck","suerte"),("mad","loco"),("pride","orgullo"),
("rest","descanso"),("soft","suave"),("tar","brea"),("tender","tierno")]

PruebaArbus> let arbus' = borra arbus "exercise" in consulta arbus' "exercise"
Nothing

PruebaArbus> let arbus' = borra arbus "exercise" in consulta arbus' "hell"
Just "infierno"

PruebaArbus> recorre (inserta arbus "rest" "resto")
[("big","grande"),("deft","diestro"),("even","llano"),("exercise","ejercicio"),
 ("heft","mango"),("hell","infierno"),("love","amor"),("luck","suerte"),
 ("mad","loco"),("pride","orgullo"),("rest","resto"),("soft","suave"),
 ("tar","brea"),("tender","tierno")]

PruebaArbus> let arbus' = inserta arbus "rest" "resto" in consulta arbus' "rest"
Just "resto"

PruebaArbus> let arbus' = inserta arbus "rest" "resto" in consulta arbus' "tar"
Just "brea"

PruebaArbus> recorre (insertaCon combi arbus "rest" "resto")
[("big","grande"),("deft","diestro"),("even","llano"),("exercise","ejercicio"),
 ("heft","mango"),("hell","infierno"),("love","amor"),("luck","suerte"),
 ("mad","loco"),("pride","orgullo"),("rest","descanso; resto"),("soft","suave"),
 ("tar","brea"),("tender","tierno")]
```

```
PruebaArbus> let arbus' = insertaCon combi arbus "rest" "resto" in consulta arbus' "rest"
Just "descanso; resto"

PruebaArbus> let arbus' = insertaCon combi arbus "rest" "resto" in consulta arbus' "tar"
Just "brea"

PruebaArbus> esNulo arbus
False

PruebaArbus> esNulo crea
ERROR: Unresolved overloading
*** Type       : Ord a => Bool
*** Expression : esNulo crea

PruebaArbus> esNulo (crea :: Arbus String String)
True

--}
```

A.45 Diccionario interactivo

```
--                        Diccionario interactivo ingles-castellano

module Diccionario (Diccionario, guardaDic, recuperaDic, diccionario) where


--                           Importacion de los arboles ordenados

import Arbus


--                                   Tipos sinonimos

type TextoCastellano = String

type TextoIngles     = String

type Diccionario     = Arbus TextoIngles TextoCastellano

type Archivo         = String

type Orden           = String


--                      Salvaguarda y recuperacion de un diccionario

-- Funcion que guarda un diccionario en un archivo.
```

```
-- Se espera que el archivo este en el directorio desde el cual ha arrancado Hugs98.
-- Algoritmo para guardar un diccionario d:
-- 1) (recorre d) da una lista de parejas (ti,tc) :: (TextoIngles, TextoCastellano).
-- 2) Con cada pareja se forma la cadena ti ++ "\n" ++ tc
-- 3) La concatenacion de todas las cadenas anteriores se escribe en el archivo.

guardaDic                     :: Diccionario -> Archivo -> IO ()
guardaDic diccionario archivo  = writeFile archivo (hazTexto diccionario)

hazTexto :: Diccionario -> String
hazTexto  = formatea . recorre

formatea                  :: [(TextoIngles, TextoCastellano)] -> String
formatea []                = ""
formatea ((ti,tc) : resto) = ti ++ "\n" ++ tc ++ "\n" ++ formatea resto

-- Funcion que recupera un diccionario de un archivo.
-- Se espera que el archivo este en el directorio desde el cual ha arrancado Hugs98,
-- y que su contenido haya sido formado por guardaDic.
-- Algoritmo para recuperar un diccionario:
-- 1) Se lee el contenido del archivo, que da xs :: String.
-- 2) A partir de xs se recupera una lista parejas :: [(TextoIngles, TextoCastellano)]
-- 3) (construye parejas) da el diccionario deseado.

recuperaDic         :: Archivo -> IO Diccionario
recuperaDic archivo  = do texto <- readFile archivo
                          return (hazDiccionario texto)

hazDiccionario :: String -> Diccionario
hazDiccionario  = construye . desformatea

desformatea :: String -> [(TextoIngles, TextoCastellano)]
desformatea texto
  | null texto    = []
  | otherwise     = (ti,tc) : desformatea resto
                  where (ti, intermedio) = separaLinea texto
                        (tc, resto)      = separaLinea intermedio

separaLinea       :: String -> (String,String)
separaLinea texto  = (primeraLinea, tail restoTexto)
                   where (primeraLinea, restoTexto) = break (== '\n') texto



--                        Interaccion con un diccionario

-- Proceso de inicializacion.
-- Espera que el diccionario se encuentre en el archivo "diccionario.txt".
-- Muestra el menu de opciones, recupera el diccionario, e inicia una sesion.
```

```
diccionario :: IO ()
diccionario  = do putChar '\n'
                  putStrLn menu
                  dic <- recuperaDic "diccionario.txt"
                  sesionCon dic

-- Texto del menu de ordenes.
-- Aparece cuando se activa el diccionario,
-- y tambien al jecutar la orden "ay"

menu :: String
menu  = "Diccionario interactivo ingles - castellano. \n" ++
        "Ordenes disponibles: \n" ++
        "  ay: muestra esta ayuda. \n" ++
        "  ct: consulta el significado de un texto ingles. \n" ++
        "  nt: incorpora un nuevo texto ingles, junto con su traduccion castellana. \n" ++
        "  na: incorpora una nueva acepcion para la traduccion de un texto ingles. \n" ++
        "  bt: borra un texto ingles, junto con su traduccion. \n" ++
        "  md: muestra el contenido del diccionario. \n" ++
        "  ts: termina la sesion, guardando el diccionario."

-- Proceso principal.
-- Depende de un dccionario dado como parametro.
-- Su efecto es:
--    pedir una orden al usuario;
--    ejecutar la orden;
--    reactivarse a si mismo
--    (excepto si la orden era "terminar sesion")

sesionCon    :: Diccionario -> IO ()
sesionCon dic = do putChar '\n'
                   putStr "Orden? "
                   orden <- getLine
                   ejecutaCon dic orden

-- Proceso auxiliar.
-- Depende de un dicionario y una orden dados como parametros.
-- Su efecto es:
--    ejecutar la orden, calculando si es preciso un nuevo diccionario
--    llamar al proceso principal, con el nuevo diccionario.
-- Caso especial:
--    La orden "terminar sesion" se ejecuta guardando el diccionario
--    y terminado la interaccion.

ejecutaCon          :: Diccionario -> Orden -> IO ()

ejecutaCon dic "ay"  = do putChar '\n'
```

```
                        putStrLn menu
                        sesionCon dic

ejecutaCon dic "ct"  = do putChar '\n'
                        putStr "Texto ingles? "
                        ti <- getLine
                        let rc = consulta dic ti
                            in case rc of
                                 Nothing -> do putStrLn "Texto desconocido."
                                               sesionCon dic
                                 Just tc -> do putStrLn tc
                                               sesionCon dic


ejecutaCon dic "nt"  = do putChar '\n'
                        putStr "Texto ingles? "
                        ti <- getLine
                        putStr "Traduccion castellana? "
                        tc <- getLine
                        sesionCon (inserta dic ti tc)


ejecutaCon dic "na"  = do putChar '\n'
                        putStr "Texto ingles? "
                        ti <- getLine
                        putStr "Nueva acepcion castellana? "
                        tc <- getLine
                        sesionCon (insertaCon combi dic ti tc)
                      where combi viejas nueva  = viejas ++ "; " ++ nueva

ejecutaCon dic "bt"  = do putChar '\n'
                        putStr "Texto ingles? "
                        ti <- getLine
                        sesionCon (borra dic ti)

ejecutaCon dic "md"  = do putChar '\n'
                        putStrLn (hazTexto dic)
                        sesionCon dic

ejecutaCon dic "ts"  = do guardaDic dic "diccionario.txt"
                        putChar '\n'
                        putStrLn "Diccionario guardado."


ejecutaCon dic _     =  do putChar '\n'
                           putStrLn "Orden desconocida"
                           sesionCon dic
```

```
--                        Minidiccionario, para pruebas:

mini :: Diccionario
mini  = construye [("big", "grande"),
                   ("deft", "diestro"),
                   ("even", "llano"),
                   ("exercise", "ejercicio"),
                   ("heft", "mango"),
                   ("hell", "infierno"),
                   ("love", "amor"),
                   ("luck", "suerte"),
                   ("mad", "loco"),
                   ("pride", "orgullo"),
                   ("rest", "descanso"),
                   ("soft", "suave"),
                   ("tar", "brea"),
                   ("tender", "tierno")]

{-- Algunas pruebas. Suponemos ya cargado el modulo "Diccionario.hs"

-- Mediante la funcion "guardaDic", guardamos el diccionario "mini" en "diccionario.txt":

Diccionario> guardaDic mini "diccionario.txt"

-- Seguidamente, lanzamos el proceso "diccionario", y ejecutamos una serie de ordenes:

Diccionario> diccionario

Diccionario interactivo ingles - castellano.
Ordenes disponibles:
  ay: muestra esta ayuda.
  ct: consulta el significado de un texto ingles.
  nt: incorpora un nuevo texto ingles, junto con su traduccion castellana.
  na: incorpora una nueva acepcion para la traduccion de un texto ingles.
  bt: borra un texto ingles, junto con su traduccion.
  md: muestra el contenido del diccionario.
  ts: termina la sesion, guardando el diccionario.

Orden? md

big
grande
deft
diestro
even
llano
exercise
ejercicio
heft
```

```
mango
hell
infierno
love
amor
luck
suerte
mad
loco
pride
orgullo
rest
descanso
soft
suave
tar
brea
tender
tierno


Orden? ct

Texto ingles? luck
suerte

Orden? ct

Texto ingles? pencil
Texto desconocido.

Orden? nt

Texto ingles? pencil
Traduccion castellana? lapiz

Orden? ct

Texto ingles? pencil
lapiz

Orden? ct

Texto ingles? even
llano

Orden? na

Texto ingles? even
```

```
Nueva acepcion castellana? par

Orden? ct

Texto ingles? even
llano; par

Orden? ct

Texto ingles? pencil
lapiz

Orden? na

Texto ingles? pencil
Nueva acepcion castellana? pincel
pincel

Orden? ct

Texto ingles? pencil
lapiz; pincel

Orden? ay

Diccionario interactivo ingles - castellano.
Ordenes disponibles:
  ay: muestra esta ayuda.
  ct: consulta el significado de un texto ingles.
  nt: incorpora un nuevo texto ingles, junto con su traduccion castellana.
  na: incorpora una nueva acepcion para la traduccion de un texto ingles.
  bt: borra un texto ingles, junto con su traduccion.
  md: muestra el contenido del diccionario.
  ts: termina la sesion, guardando el diccionario.

Orden? ct

Texto ingles? window
Texto desconocido.

Orden? nt

Texto ingles? window
Traduccion castellana? ventana

Orden? bt

Texto ingles? pride
```

```
Orden? md

big
grande
deft
diestro
even
llano; par
exercise
ejercicio
heft
mango
hell
infierno
love
amor
luck
suerte
mad
loco
pencil
lapiz; pincel
rest
descanso
soft
suave
tar
brea
tender
tierno
window
ventana


Orden? ts
ts

Diccionario guardado.

-- Volvemos a lanzar "diccionario", y observamos el estado del diccionario.
-- Comprobamos que ha se ha preservado el diccionario final de la sesion anterior.

Diccionario> diccionario

Diccionario interactivo ingles - castellano.
Ordenes disponibles:
  ay: muestra esta ayuda.
  ct: consulta el significado de un texto ingles.
  nt: incorpora un nuevo texto ingles, junto con su traduccion castellana.
```

```
  na: incorpora una nueva acepcion para la traduccion de un texto ingles.
  bt: borra un texto ingles, junto con su traduccion.
  md: muestra el contenido del diccionario.
  ts: termina la sesion, guardando el diccionario.

Orden? md

big
grande
deft
diestro
even
llano; par
exercise
ejercicio
heft
mango
hell
infierno
love
amor
luck
suerte
mad
loco
pencil
lapiz; pincel
rest
descanso
soft
suave
tar
brea
tender
tierno
window
ventana


Orden? fu

Orden desconocida

Orden?

--}
```

A.46 Intérprete de expresiones aritméticas

```
--            Este modulo utiliza un tipo de datos recursivo para representar
--                la sintaxis abstracta de expresiones enteras simples

module Expresion (Ide, Op(..), Exp(..), eval) where


-- Importacion del TAD que representa los estados de memoria:

import Estado

-- Ejemplo de estado, para pruebas:

est :: Estado
est  = asigna s "y" 20 where s = asigna ini "x" 10


-- Tipos que representan los identificadores, operadores y expresiones:

type Ide = String

data Op  = Add | Sub | Mul | Div | Mod
         deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)

data Exp = Cte Int | Var Ide | App Op Exp Exp
         deriving (Eq, Ord, Read, Show)


-- Ejemplo de expresion, para pruebas: (x+3) * ((y-1) mod 2)

expr :: Exp
expr  = App Mul exp1 exp2
        where exp1 = App Add (Var "x") (Cte 3)
              exp2 = App Mod exp3 (Cte 2)
              exp3 = App Sub (Var "y") (Cte 1)



-- Interprete de expresiones: funciones "eval" y "apply":

eval                       :: Exp -> Estado -> Int
eval (Cte val)          s  = val
eval (Var ide)          s  = consulta s ide
eval (App op arg1 arg2) s  = apply op (eval arg1 s) (eval arg2 s)

apply     :: Op -> Int -> Int -> Int
apply Add  = (+)
apply Sub  = (-)
```

```
apply Mul  = (*)
apply Div  = div
apply Mod  = mod

{-- Prueba

Expresion> eval expr est
13

--}
```

A.47 Estado de una máquina simple

```
--  Este modulo implementa un TAD que representa el estado de una maquina simple
--
--        Un estado se representa como un arbol de busqueda que asocia a
--           identificadores (cadenas de caracteres) valores enteros


module Estado (Estado, ini, consulta, asigna, muestra) where

-- Importacion de los arboles de busqueda:

import qualified Arbus

-- Tipo Estado:

newtype Estado = E (Arbus.Arbus String Int)

-- Invariante de la representacion:
-- valido s = True

-- Equivalencia abstracta:
-- equiv s s' = Para todo i :: String (consulta s i == consulta s' i)

-- Operaciones publicas del TAD Estado:

-- Estado inicial.
-- Cualquier id, tiene asociado el valor 0 en "ini".
-- Ver def. de la operacion "consulta".

ini :: Estado
ini  = E Arbus.crea

-- Consulta del valor d eun id. en un estado.

consulta            :: Estado -> String -> Int
consulta (E arb) ide = case Arbus.consulta arb ide of
                         Nothing -> 0
```

```
                        Just v  -> v

-- Asignacion de valor a un id. en un estado.

asigna                :: Estado -> String -> Int -> Estado
asigna (E arb) ide val = E (Arbus.inserta arb ide val)

muestra        :: Estado -> [(String,Int)]
muestra (E arb) = Arbus.recorre arb

{-- Pruebas:

Expresion> expr
App Mul (App Add (Var "x") (Cte 3)) (App Mod (App Sub (Var "y") (Cte 1)) (Cte 2))

Expresion> muestra est
[("x",10),("y",20)]

Expresion> eval expr est
13

--}
```

A.48 Intérprete de programas imperativos

```
--            Este modulo utiliza un tipo de datos recursivo para representar
--                la sintaxis abstracta de programas imperativos simples


module Programa (Prog(..), run) where


-- Importacion del modulo de las expresiones:

import Expresion

import Estado

-- Tipo de datos que representa los programas.
-- En condiciones, convenimos que:
--    un entero positivo representa "cierto";
--    un entero no positivo representa "falso".

data Prog =   Asig Ide Exp      -- asignacion
            | Seq Prog Prog     -- composicion secuencial
            | If Exp Prog Prog  -- distincion de casos
            | While Exp Prog    -- bucle
          deriving (Eq, Ord, Read, Show)
```

```
-- Programa para pruebas:
-- calcula en "res" el valor de "base" elevado a "exp":
--
--   aux := exp;
--   res := 1;
--   while aux do
--     res := res*base;
--     aux := aux-1

prog :: Prog
prog  = Seq start iterate
        where start   = Seq (Asig "aux" (Var "exp"))
                            (Asig "res" (Cte 1))
              iterate = While cond body
              cond    = Var "aux"
              body    = Seq (Asig "res" (App Mul (Var "res") (Var "base")))
                            (Asig "aux" (App Sub (Var "aux") (Cte 1)))

-- Estado para pruebas:
-- "base" y "exp" tienen valores 2 y 3, respectivamente:

est :: Estado
est  = asigna (asigna ini "base" 2) "exp" 3



-- Interprete de programas: funcion "run":

run                :: Prog -> Estado -> Estado
run (Asig ide exp)  = \s -> (asigna s ide (eval exp s))
run (Seq p q)       = run q . run p
run (If cond p q)   = ifThenElse (pos cond) (run p) (run q)
run (While cond p)  = while (pos cond) (run p)


-- Funciones auxiliares: "pos", "ifThenElse" y "while":

pos     :: Exp -> Estado -> Bool
pos e s  = eval e s > 0

ifThenElse         :: (a -> Bool) -> (a -> a) -> (a -> a) -> a -> a
ifThenElse t f g s  = if t s
                      then f s
                      else g s

while       :: (a -> Bool) -> (a -> a) -> a -> a
while t f s  = if t s
               then while t f (f s)
```

```
                else s


{-- Pruebas:

Programa> muestra est
[("base",2),("exp",3)]

Programa> muestra (run prog est)
[("aux",0),("base",2),("exp",3),("res",8)]

--}
```

A.49 Intérprete paso a paso de programas imperativos

```
--            Este modulo utiliza un tipo de datos recursivo para representar
--                la sintaxis abstracta de programas imperativos simples


module ProgramaPP (Prog(..), run, trans, transs) where


-- Importacion del modulo de las expresiones:

import Expresion

import Estado

-- Tipo de datos que representa los programas.
-- En condiciones, convenimos que:
--    un entero positivo representa "cierto";
--    un entero no positivo representa "falso".

data Prog =   Skip              -- programa nulo
            | Asig Ide Exp      -- asignacion
            | Seq Prog Prog     -- composicion secuencial
            | If Exp Prog Prog  -- distincion de casos
            | While Exp Prog    -- bucle
          deriving (Eq, Ord, Read, Show)

-- Funcion booleana que reconoce el programa nulo:

nulo               :: Prog -> Bool
nulo Skip            = True
nulo (Asig ide exp)  = False
nulo (Seq p q)       = False
nulo (If cond p q)   = False
```

```
nulo (While cond p)  = False


-- Programa para pruebas:
-- calcula en "res" el valor de "base" elevado a "exp":
--
--   aux := exp;
--   res := 1;
--   while aux do
--     res := res*base;
--     aux := aux-1

prog :: Prog
prog  = Seq start iterate
        where start   = Seq (Asig "aux" (Var "exp"))
                            (Asig "res" (Cte 1))
              iterate = While cond body
              cond    = Var "aux"
              body    = Seq (Asig "res" (App Mul (Var "res") (Var "base")))
                            (Asig "aux" (App Sub (Var "aux") (Cte 1)))

-- Estado para pruebas:
-- "base" y "exp" tienen valores 2 y 3, respectivamente:

est :: Estado
est  = asigna (asigna ini "base" 2) "exp" 3



-- Interprete de programas: funcion "run":

run                :: Prog -> Estado -> Estado
run Skip            = id
run (Asig ide exp)  = \s -> (asigna s ide (eval exp s))
run (Seq p q)       = run q . run p                         -- run p . run q WRONG!
run (If cond p q)   = ifThenElse (pos cond) (run p) (run q)
run (While cond p)  = while (pos cond) (run p)


-- Funciones auxiliares: "pos", "ifThenElse" y "while":

pos     :: Exp -> Estado -> Bool
pos e s  = eval e s > 0

ifThenElse         :: (a -> Bool) -> (a -> a) -> (a -> a) -> a -> a
ifThenElse t f g s  = if t s
                      then f s
                      else g s
```

```
while       :: (a -> Bool) -> (a -> a) -> a -> a
while t f s  = if t s
                 then while t f (f s)
                 else s

-- Interprete de programas paso a paso : funcion "trans":

trans                :: (Prog, Estado) -> (Prog, Estado)
trans (Skip, s)          = (Skip, s)
trans (Asig ide exp, s)  = (Skip, asigna s ide (eval exp s))
trans (Seq p q, s)       = if nulo p
                           then (q,s)
                           else (Seq p' q, s')
                         where (p',s') = trans (p,s)
trans (If cond p q, s)   = if pos cond s
                           then (p, s)
                           else (q, s)
trans (While cond p, s)  = if pos cond s
                           then (Seq p (While cond p), s)
                           else (Skip, s)

-- Interprete de programas paso a paso : funcion "transs":

transs                :: (Prog, Estado) -> (Prog, Estado)
transs (p,s) = if nulo p
                 then (p,s)
                 else transs (trans (p,s))

-- Propiedad: run p s = snd (transs (p,s))

{-- Pruebas:

ProgramaPP> muestra est
[("base",2),("exp",3)]

ProgramaPP> muestra (run prog est)
[("aux",0),("base",2),("exp",3),("res",8)]

ProgramaPP> (fst conf, muestra (snd conf)) where conf = transs (prog,est)
(Skip,[("aux",0),("base",2),("exp",3),("res",8)])

--}
```