

Programación Funcional

Jeroen Fokker

1996

Universidad de Utrecht
Departamento de Informática

Traducción: Programa MEMI
Universidad Mayor de San Simón
Hielko R. Ophoff & Bernardo Sánchez J.

Revisión: Universidad Carlos III de Madrid
Carlos Delgado Kloos & Natividad Martínez Madrid

Programación Funcional

©Copyright 1992–1996 Departamento de Informática, Universidad de Utrecht

Este texto puede ser reproducido para fines educativos bajo las siguientes condiciones:

- que no se modifique ni reduzca el texto,
- que en especial no se elimine este mensaje,
- y que no se vendan las copias con fines lucrativos.

Primera edición: Septiembre 1992

Segunda edición: Agosto 1995

Segunda edición revisada: Marzo 1996

Índice General

1 Programación Funcional	1
1.1 Lenguajes funcionales	1
1.1.1 Funciones	1
1.1.2 Lenguajes	1
1.2 El intérprete de Gofer	2
1.2.1 Calcular expresiones	2
1.2.2 Definir funciones	4
1.2.3 Comandos del intérprete	5
1.3 Funciones estándar	6
1.3.1 Operadores, funciones predefinidas y funciones primitivas	6
1.3.2 Nombres de funciones y operadores	7
1.3.3 Funciones sobre números	8
1.3.4 Funciones booleanas	9
1.3.5 Funciones sobre listas	10
1.3.6 Funciones de funciones	11
1.4 Definición de funciones	11
1.4.1 Definición por combinación	11
1.4.2 Definición por distinción de casos	12
1.4.3 Definición por análisis de patrones	13
1.4.4 Definición por recursión o inducción	14
1.4.5 Indentación y comentarios	15
1.5 Tipado	16
1.5.1 Clases de errores	16
1.5.2 Expresiones de tipo	18
1.5.3 Polimorfismo	19
1.5.4 Funciones con más de un parámetro	20
1.5.5 Sobrecarga	20
Ejercicios	21
2 Números y funciones	23
2.1 Operadores	23
2.1.1 Operadores como funciones y viceversa	23
2.1.2 Precedencias	23
2.1.3 Asociatividad	24
2.1.4 Definición de operadores	25
2.2 Currificación	25
2.2.1 Instanciar parcialmente	25
2.2.2 Paréntesis	26
2.2.3 Secciones de operadores	27
2.3 Funciones como parámetros	27
2.3.1 Funciones sobre listas	27
2.3.2 Iteración	29
2.3.3 Composición	30

2.4	Funciones numéricas	31
2.4.1	Calcular con enteros	31
2.4.2	Diferenciar numéricamente	33
2.4.3	La raíz cuadrada	34
2.4.4	Ceros de una función	35
2.4.5	Inverso de una función	36
	Ejercicios	38
3	Estructuras de datos	39
3.1	Listas	39
3.1.1	Estructura de una lista	39
3.1.2	Funciones sobre listas	41
3.1.3	Funciones de orden superior sobre listas	45
3.1.4	Ordenamiento de listas	47
3.2	Listas especiales	48
3.2.1	Cadenas	48
3.2.2	Caracteres	49
3.2.3	Funciones de cadenas y caracteres	50
3.2.4	Listas infinitas	52
3.2.5	Evaluación perezosa	52
3.2.6	Funciones sobre listas infinitas	53
3.3	Tuplas	56
3.3.1	Uso de tuplas	56
3.3.2	Definiciones de tipos	58
3.3.3	Números racionales	59
3.3.4	Listas y tuplas	60
3.3.5	Tuplas y currificación	61
3.4	Árboles	61
3.4.1	Definiciones de datos	61
3.4.2	Árboles de búsqueda	64
3.4.3	Usos especiales de definiciones de datos	66
	Ejercicios	68
4	Algoritmos sobre listas	75
4.1	Funciones combinatorias	75
4.1.1	Segmentos y sublistas	75
4.1.2	Permutaciones y combinaciones	77
4.1.3	La notación @	79
	Ejercicios	80
A	Literatura relacionada con la Programación Funcional	81

Capítulo 1

Programación Funcional

1.1 Lenguajes funcionales

1.1.1 Funciones

Los primeros ordenadores se construyeron en los años cuarenta. Los primerísimos modelos fueron ‘programados’ con grandes relés. Pronto se almacenaron los programas en la memoria del ordenador, haciendo que los primeros *lenguajes de programación* hicieran su entrada.

En aquel tiempo el uso de un ordenador era muy costoso y era lógico que el lenguaje de programación guardara mucha relación con la arquitectura del ordenador. Un ordenador consta de una unidad de control y una memoria. Por eso un programa consistía en instrucciones para cambiar el contenido de la memoria. La unidad de control se encargaba de ejecutarlas. De esta manera se creó *el estilo de programación imperativa*. Los lenguajes de programación imperativa como Pascal y C se caracterizan por la existencia de *asignaciones* ejecutadas consecutivamente.

Antes de la existencia de los ordenadores se inventaron métodos para resolver problemas. Por tanto, no existía la necesidad de hablar en términos de una memoria que cambie por instrucciones en un programa. En la matemática de los últimos cuatrocientos años son muy importantes las *funciones*. Éstas establecen la relación entre los parámetros (la ‘entrada’) y el resultado (la ‘salida’) de procesos definidos.

Con cada computación, el resultado depende de una u otra forma de los parámetros. Por esa razón, una función es una buena manera de especificar una computación. Ésta es la base del *estilo de programación funcional*. Un ‘programa’ consiste en la definición de una o más funciones. Para la ejecución de un programa, se dan parámetros a una función y el ordenador tiene que calcular el resultado. Con este tipo de computación existe libertad en la manera de ejecución. ¿Por qué tendría que describirse en qué orden deben ejecutarse las computaciones parciales?

Con el tiempo, al bajar los precios de los ordenadores y al subir los precios de los programadores, llega a ser más importante describir las computaciones en un lenguaje que esté más cerca del ‘mundo del hombre’, que cerca del ordenador. Los lenguajes funcionales se unen a la tradición matemática y no están muy influidos por la arquitectura concreta del ordenador.

1.1.2 Lenguajes

La base teórica de la programación imperativa fue dada (en Inglaterra) por Alan Turing en los años treinta. También la teoría de funciones como modelo de computación proviene de los años veinte y treinta. Los fundadores son, entre otros, M. Schönfinkel (en Alemania y Rusia), Haskell Curry (en Inglaterra) y Alonzo Church (en los Estados Unidos).

Fue en el comienzo de los años cincuenta cuando a alguien se le ocurrió usar esta teoría efectivamente, como base de un lenguaje de programación. El lenguaje Lisp de John McCarthy fue el primer lenguaje de

programación funcional y fue el único por muchos años. Aunque todavía se usa Lisp, no es un lenguaje que reúna las exigencias. Debido a la creciente complejidad de los programas de ordenador, se hizo necesaria una mayor verificación del programa por parte de el ordenador. Por ello el *tipado* adquirió una gran importancia. Por eso no es de extrañar que en los años ochenta se crearan un gran número de lenguajes funcionales tipados. Algunos ejemplos son ML, Scheme (una adaptación de Lisp), Hope y Miranda.

A la larga, cada investigador se dedicó a desarrollar su propio lenguaje. Para detener este crecimiento incontrolado, un grupo de investigadores notables concibió un lenguaje que incluía todas las mejores cualidades de los diferentes lenguajes. Este lenguaje se llama Haskell. Las primeras implementaciones de Haskell fueron hechas a comienzos de los años noventa. Este lenguaje es bastante ambicioso y de difícil implementación. El lenguaje Gofer es un lenguaje como Haskell, pero un poco más simple. Se usa para investigaciones teóricas y para metas educativas. Por su gran disponibilidad gana popularidad rápidamente. La pregunta es si Gofer sobrevivirá. Esto no importa mucho, el estudio de Gofer es útil porque tiene una relación fuerte con Haskell, que está aceptado por mucha gente en el área de la informática.

Los lenguajes ML y Scheme tienen también muchos seguidores. Estos lenguajes han hecho algunas concesiones en la dirección de los lenguajes imperativos. Son de menos utilidad para mostrar lo que es un lenguaje funcional propio. Miranda es un lenguaje funcional propio, pero es difícil obtenerlo.

En este texto se utiliza el lenguaje Gofer. En este lenguaje se usan muchos conceptos de una manera consecuente, lo que hace que pueda ser aprendido fácilmente. Quien conozca Gofer, tendrá pocos problemas en comprender otros lenguajes funcionales.

1.2 El intérprete de Gofer

1.2.1 Calcular expresiones

En un lenguaje funcional se pueden definir funciones. Estas funciones se usan en una expresión cuyo valor tiene que ser calculado. Para calcular el valor de una expresión se necesita un programa que entienda las definiciones de las funciones. Tal programa se llama intérprete.

Para el lenguaje Gofer que se usa en este texto existe un intérprete. Para iniciar el intérprete se teclea el nombre del programa, `gofer`. En la pantalla aparece lo siguiente:

```
%gofer
Gofer Version 2.21 Copyright (c) Mark P Jones 1991

Reading script file "/usr/local/lib/gofer/prelude":
Parsing.....
Dependency analysis.....
Type checking.....
Compiling.....
```

En el fichero `prelude` (que en este ejemplo está en el directorio `/usr/local/lib/gofer`) están las definiciones de las funciones estándar. Lo primero que hace el intérprete de Gofer es analizar estas funciones estándar. ‘Prelude’ significa ‘preludio’: este archivo se examina antes de que se puedan definir nuevas funciones. En el caso de este ejemplo no hay nuevas funciones, por eso el intérprete indica con

```
Gofer session for:
/usr/local/lib/gofer/prelude
```

que se pueden usar solamente las definiciones que están en el prelude. Finalmente, el intérprete comunica que se puede teclear `?:` para obtener algunas instrucciones para el uso:

```
Type :? for help
?
```

El signo de interrogación indica que el intérprete está listo para calcular el valor de una expresión. En el prelude están definidas, entre otras, las funciones aritméticas. Por eso, se puede usar el intérprete como calculadora.

```
? 5+2*3
11
(5 reductions, 9 cells)
?
```

El intérprete calcula el valor de la expresión (el operador `*` indica multiplicación). Después de decir cuál es el resultado, el intérprete indica lo que fue necesario para el cálculo: ‘5 reductions’ (una medida para el tiempo utilizado) y ‘9 cells’ (una medida para la memoria utilizada). El signo de interrogación indica que el intérprete está listo para el próximo cálculo.

También se pueden usar funciones conocidas como `sqrt`, que calcula el ‘square root’ o raíz cuadrada de un número. Por ejemplo:

```
? sqrt(2.0)
1.41421
(2 reductions, 13 cells)
?
```

Como se utilizan muchas funciones en un lenguaje funcional, se pueden omitir los paréntesis en la llamada a una función. En expresiones complicadas son muchos paréntesis menos y esto hace que la expresión sea más clara. Entonces, la llamada en el anterior ejemplo puede escribirse como:

```
? sqrt 2.0
1.41421
(2 reductions, 13 cells)
```

En los libros de matemáticas es costumbre que ‘poner expresiones seguidas’ significa que estas expresiones deben ser multiplicadas. Junto a una llamada a una función se tienen que usar paréntesis. En expresiones de Gofer se usan invocaciones a las funciones más veces que multiplicaciones. Por eso, en Gofer se interpreta ‘poner expresiones seguidas’ como llamada y la multiplicación debe ser anotada explícitamente (con `*`):

```
? sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1.0
(8 reductions, 27 cells)
```

Grandes cantidades de números pueden ser colocados en una *lista*. Las listas se representan con corchetes. Hay varias funciones estándar que operan con listas:

```
? sum [1..10]
55
(91 reductions, 130 cells)
```

En el anterior ejemplo, `[1..10]` es la notación en Gofer para la lista de números desde 1 hasta 10 inclusive. La función estándar `sum` puede ser aplicada a tal lista para calcular la suma (55) de los números en la lista. Igual que en las funciones `sqrt` y `sin`, no se necesitan paréntesis en la llamada a la función `sum`.

Una lista es una de las maneras de reunir datos para que se puedan aplicar funciones a grandes cantidades de datos simultáneamente. Las listas pueden ser el resultado de una función:

```
? sums [1..10]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
(111 reductions, 253 cells)
```

La función estándar `sums` devuelve todos los resultados intermedios junto a la suma de los números en la lista. Todos estos valores están en una lista, que es el resultado de la función `sums`.

Existen varias funciones estándar que manipulan listas. Los nombres generalmente hablan por sí solos: `length` determina el tamaño de una lista, `reverse` pone los elementos de una lista en el orden inverso, y `sort` ordena los elementos de menor a mayor.

```
? length [1,5,9,3]
4
(18 reductions, 33 cells)
? reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
(99 reductions, 199 cells)
? sort [1,6,2,9,2,7]
[1, 2, 2, 6, 7, 9]
(39 reductions, 110 cells)
```

En una expresión se pueden combinar varias funciones. Se puede, por ejemplo, primero ordenar una lista y después invertirla:

```
? reverse (sort [1,6,2,9,2,7])
[9, 7, 6, 2, 2, 1]
(52 reductions, 135 cells)
```

Como de costumbre, $g(f\ x)$ significa que la función f se aplica al parámetro x , y que g se aplica a su resultado. Los paréntesis en este ejemplo son necesarios (también en Gofer), para indicar que $(f\ x)$ en su totalidad funciona como parámetro de la función g .

1.2.2 Definir funciones

En un lenguaje funcional se pueden definir nuevas funciones. Estas funciones se pueden usar junto a las funciones que están definidas en el preludio. La definición de una función se almacena en un fichero. El fichero se puede crear con cualquier procesador de textos.

Sería molesto si se tuviera que salir del intérprete de Gofer para empezar un trabajo en el procesador de textos, etc. Por eso, se permite trabajar con el procesador de textos desde el intérprete sin salir de él. Al salir del procesador de textos, el intérprete puede procesar la nueva definición.

El procesador de textos se arranca con el comando `:edit`, seguido por el nombre de un fichero, por ejemplo:

```
? :edit nuevo
```

Por los dos puntos al principio de la regla, el intérprete sabe que `edit` no es una expresión, sino un comando interno. El procesador de textos ocupa temporalmente el lugar que ocupaba Gofer en la pantalla.¹

En el fichero `nuevo`, por ejemplo, se puede describir la definición de la función factorial. El factorial de un número n (normalmente escrito como $n!$) es el producto de los números desde 1 hasta n inclusive, por ejemplo $4! = 1 * 2 * 3 * 4 = 24$.

En Gofer la definición de la función `fac` sería:

```
fac n = product [1..n]
```

Esta definición usa la notación para ‘lista de números entre dos valores’ y la función estándar `product`.

¹El procesador de textos que se usa está determinado por el valor de la *variable de entorno* `EDITOR`, que se puede cambiar con el comando de Unix `setenv`.

Una vez que la función está escrita, a continuación, se puede salir del procesador de textos. Después aparece el símbolo `?`, que indica que Gofer está activo otra vez. Antes de poder usar la nueva función, Gofer tiene que saber que el nuevo fichero contiene definiciones de funciones. Esto se puede indicar a través del comando `:load`, como se demuestra a continuación:

```
? :load nuevo
Reading script file "nuevo":
Parsing.....
Dependency analysis....
Type checking.....
Compiling.....

Gofer session for:
/usr/local/lib/gofer/prelude
nuevo
?
```

Después de analizar el nuevo fichero, Gofer dice que aparte del fichero `prelude`, también se pueden usar las definiciones del fichero `nuevo`:

```
? fac 6
720
(59 reductions, 87 cells)
```

Es posible añadir definiciones a un fichero. En este caso es suficiente con el comando `:edit`; no hace falta mencionar el nombre del fichero.

Una función que se puede añadir, por ejemplo, es la función número combinatorio: el número de maneras en que se pueden escoger k objetos de un conjunto de n elementos. Según los libros de matemáticas este número es:

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

Esta definición se puede escribir en Gofer casi directamente:

```
comb n k = fac n / (fac k * fac (n-k))
```

Las definiciones de las funciones en un fichero pueden invocar a otras definiciones: `comb`, por ejemplo, puede usar la función `fac`. Además, se pueden usar las funciones estándar.

Después de salir del procesador de textos, Gofer examina el fichero modificado automáticamente. Entonces, no es necesario que se utilice otra vez el comando `:load`. Se puede calcular directamente de cuántas maneras se puede componer una comisión de tres personas de un grupo de diez:

```
? comb 10 3
120
(189 reductions, 272 cells)
```

1.2.3 Comandos del intérprete

Además de `:edit` y `:load`, existen otros comandos que se envían directamente al intérprete (es decir, no deben ser interpretados como expresiones). Todos empiezan por dos puntos.

Existen los siguientes comandos:

:? Éste es comando para hacer un listado de todos los comandos posibles. (‘No hace falta saberlo todo, si se sabe cómo encontrar lo que se quiere saber.’)

:quit Con este comando se termina una sesión de Gofer.

:load *fichero(s)* Después de este comando Gofer conoce las funciones que están definidas en los ficheros especificados. Con **:load** sin nombres de ficheros Gofer elimina todo excepto las definiciones del prelude.

:also *fichero(s)* Con este comando se pueden añadir definiciones de otros ficheros, sin eliminar las definiciones existentes.

:edit *fichero* Éste es un comando para crear o modificar un fichero. Sin nombre, **edit** usa el último fichero que se ha recuperado y, después de salir del procesador de textos, se carga otra vez.

:reload Éste es un comando para leer (cargar) nuevamente el último fichero que se ha utilizado (por ejemplo, si se hicieron cambios en otra ‘ventana’).

:find *nombre_de_una_función* Con este comando el procesador de textos empieza exactamente en el lugar donde está definida la función.

pág. 16 **:type** *expresión* Con este comando se determina el tipo (ver la sección 1.5) de la expresión dada.

:set \pm *letra* Con este comando se pueden aplicar o eliminar algunas opciones. Por ejemplo, Gofer normalmente escribe después de cada computación el número de reducciones y cuánta memoria se ha usado. Después de dar el comando **:set -s** deja de hacerlo. Con **:set +s** da otra vez esta información.

Las opciones más importantes son **s**, **t**, **g** en **i**:

- **s**: Información estadística (reducciones y memoria) después de cada computación.
- **t**: Da el tipo de cada resultado de una computación (ver el párrafo 1.5.2).
- **g**: Se informa sobre *Garbage collection*.
- **i**: Las constantes de enteros se manejan de forma especial.

pág. 18

Las otras opciones no tienen importancia para el uso común de Gofer.

1.3 Funciones estándar

1.3.1 Operadores, funciones predefinidas y funciones primitivas

Además de las definiciones de funciones, se pueden definir también constantes y operadores. Una *constante* es una función sin parámetros. La siguiente es la definición de una función:

```
pi = 3.1415926
```

Un *operador* es una función con dos parámetros, que se escribe *entre* los parámetros y no delante de ellos. En Gofer uno puede definir sus propios operadores. Por ejemplo, la función **comb** se podría definir mejor como operador, por ejemplo como **!^!** :

```
n !^! k = fac n / (fac k * fac (n-k))
```

En el prelude están definidos más de doscientos funciones estándar y operadores. La mayoría de las funciones del prelude son funciones ordinarias, que podría definir uno mismo. La función **sum**, por ejemplo, está en el prelude porque se usa frecuentemente. Si no hubiese estado en el prelude, uno mismo la podría haber definido. Con el comando **:find** uno puede ver sus propias definiciones así como las del prelude. Ésta es una manera fácil de saber lo que hace una función estándar. Por ejemplo la definición de **sum** es

```
sum = foldl' (+) 0
```

Por supuesto, uno tiene que saber también lo que hace la función estándar `foldl'`, pero ésta también se puede buscar...

Otras funciones, como la función `primPlusInt` que suma dos enteros, no están definidas, están de una forma 'mágica' en el intérprete. Esta clase de funciones se llaman *funciones primitivas*. Pueden ser buscadas, pero no se sabrá mucho más si se busca la definición en el prelude:

```
primitive primPlusInt "primPlusInt"
```

El número de funciones primitivas es pequeño. La mayoría de las funciones estándar están definidas en Gofer. Éstas son las llamadas funciones *predefinidas*.

1.3.2 Nombres de funciones y operadores

En la definición de la función

```
fac n = product [1..n]
```

`fac` es el nombre de la función que se define y `n` el nombre de su parámetro.

Los nombres de las funciones y los parámetros tienen que empezar con una letra minúscula. Después pueden seguir más letras (minúsculas y mayúsculas), pero también números, el símbolo `'` y el símbolo `_`. Las letras minúsculas y letras mayúsculas son diferentes para el intérprete. Algunos ejemplos de nombres de funciones y parámetros son:

```
f      sum   x3   g'   tot_de_macht   nombreLargo
```

El símbolo `_` se usa muchas veces para hacer más legibles los nombres largos. Otra manera para esto, es empezar las palabras que forman un nombre con mayúsculas (excepto la primera palabra). Estos métodos se usan también en otros lenguajes de programación.

Los números y apóstrofes en un nombre pueden ser usados para enfatizar que un número de funciones o de parámetros tienen relación unos con otros. Esto es solamente útil para el usuario humano, para el intérprete el nombre `x3` tiene tan poca relación con `x2` como con `qX'a_y`.

Los nombres que empiezan con una letra mayúscula se usan para funciones especiales y constantes, las llamadas *funciones constructoras*. Su definición está descrita en el párrafo 3.4.1.

pág. 61

Existen 16 nombres que no se pueden usar para funciones o variables. Estas *palabras reservadas* tienen un sentido especial para el intérprete. Las palabras reservadas en Gofer son:

```
case      class      data      else
if         in         infix    infixl
infixr    instance let     of
primitive then      type     where
```

Más adelante en este texto, veremos el sentido de las palabras reservadas.

Los nombres de operadores contienen uno o más símbolos. Un operador puede consistir en un símbolo como en `+`, pero también en dos (`&&`) o más (`!^!`) símbolos. Los símbolos que se pueden usar para formar operadores son los siguientes:

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

Operadores permitidos son, por ejemplo:

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

La primera de las dos últimas líneas contiene operadores que están definidos en el preludio. En la segunda línea están los operadores que uno puede definirse. Los operadores que empiezan con dos puntos (:) son usados para funciones constructoras (como los nombres que empiezan por mayúscula).

Existen once combinaciones de símbolos que no se pueden usar como operador, porque tienen un sentido especial en Gofer. Son los siguientes:

```
:: = .. -- @ \ | <- -> ~ =>
```

Quedan bastantes para dar rienda suelta a su creatividad...

1.3.3 Funciones sobre números

Existen dos tipos de números en Gofer:

- Enteros, como 17, 0 y -3;
- Números de punto flotante, como 2.5, -7.81, 0.0, 1.2e3 y 0.5e-2.

La letra e en números reales indica ‘por 10 elevado a’. Por ejemplo 1.2e3 es el número $1.2 \cdot 10^3 = 1200.0$. El número 0.5e-2 significa $0.5 \cdot 10^{-2} = 0.005$.

Se pueden usar los cuatro operadores aritméticos suma (+), resta (-), multiplicación (*) y división (/) para enteros y para números reales:

```
? 5-12
-7
? 2.5*3.0
7.5
```

En la división de enteros se trunca el resultado a su parte entera:

```
? 19/4
4
```

Si necesitamos división exacta, entonces tenemos que usar números reales:

```
? 19.0/4.0
4.75
```

No se pueden usar los dos tipos al mismo tiempo:

```
? 1.5+2
ERROR: Type error in application
*** expression   : 1.5 + 2
*** term         : 1.5
*** type         : Float
*** does not match : Int
```

Estos cuatro operadores son primitivos, tanto para números enteros como para números reales.

En el preludio existen también algunas otras funciones estándar para enteros. Éstas no son primitivas, sino predefinidas, y en consecuencia están escritas en Gofer. Si no hubieran estado en el preludio, habrían podido definirse. Algunas de estas funciones son:

abs	el valor absoluto de un número
signum	-1 para números negativos, 0 para cero, 1 para números positivos
gcd	el máximo común divisor de dos números
^	el operador “potencia de”

Para números reales están definidas algunas operaciones que son primitivas:

```
sqrt  la función raíz cuadrada
sin   la función seno
log   la función logaritmo natural
exp   la función exponente (e-elevado-a)
```

Existen dos funciones primitivas que convierten enteros en números reales y viceversa:

```
fromInteger  entero a número real
round        redondear un número real a un entero
```

Los enteros no pueden ser mayores que un máximo determinado. La mayoría de las veces este máximo es 2^{31} . En ordenadores pequeñas es 2^{15} . Si el resultado llega a ser mayor que el máximo, entonces el resultado será un valor sin sentido, por ejemplo:

```
? 3 * 100000000
300000000
? 3 * 1000000000
-1294967296
```

También los números reales tienen un valor máximo (según la máquina 10^{38} o 10^{308}) y un valor positivo mínimo (10^{-38} o 10^{-308}). Además, la precisión es limitada (8 o 15 números significativos):

```
? 123456789.0 - 123456780.0
8.0
```

Es aconsejable usar enteros cuando se tiene una cantidad discreta. Se pueden usar números reales para cantidades continuas.

En este texto se supone que todos los valores son menores que el máximo permitido.

1.3.4 Funciones booleanas

El operador `<` comprueba si un número es menor que otro número. El resultado es la constante `True` (si es menor) o la constante `False` (si no):

```
? 1<2
True
? 2<1
False
```

Los valores `True` y `False` son los únicos elementos del conjunto de valores booleanos (George Boole de Inglaterra era un matemático). Las funciones (y operadores) que resultan en tales valores se llaman *funciones booleanas*.

Existe un operador `<` (menor que), un operador `>` (mayor que), un operador `<=` (menor o igual que), un operador `>=` (mayor o igual que). También existen los operadores `==` (igual a) y `/=` (distinto de). Ejemplos:

```
? 2+3 > 1+4
False
? sqrt 2.0 <= 1.5
True
? 5 /= 1+4
False
```

Los resultados de las funciones booleanas pueden ser combinados con los operadores `&&` ('and': conjunción lógica) y `||` ('or': disyunción lógica):

```
? 1<2 && 3<4
True
? 1<2 && 3>4
False

? 1==1 || 2==3
True
```

Existe una función `not` que cambia `True` en `False` y viceversa. También existe una función `even` que comprueba si un entero es par o impar:

```
? not False
True
? not (1<2)
False
? even 7
False
? even 0
True
```

1.3.5 Funciones sobre listas

En el preludio están definidas algunas funciones y operadores sobre listas. De éstos solamente uno es primitivo (el operador `:`), las otras son definidas en base a una definición de `Gofer`.

Ya hemos visto algunas funciones sobre listas: `length`, `sum` y `sums`.

El operador `:` coloca un elemento al frente de una lista. El operador `++` concatena dos listas en una sola. Por ejemplo:

```
? 1 : [2,3,4]
[1, 2, 3, 4]
? [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

La función `null` es una función booleana sobre listas. Comprueba si la lista está vacía (lista sin elementos). La función `and` comprueba si todos los elementos de una lista son `True`:

```
? null [ ]
True
? and [ 1<2, 2<3, 1==0]
False
```

Algunas funciones tienen dos parámetros. Por ejemplo, la función `take` tiene dos parámetros: un número y una lista. Si el número es n , el resultado es una lista con los primeros n elementos de la lista:

```
? take 3 [2..10]
[2, 3, 4]
```

1.3.6 Funciones de funciones

En las funciones que hemos visto, los parámetros son números, valores booleanos o listas. Pero un parámetro de una función puede también ser una función. Un ejemplo es la función `map`. Tiene dos parámetros: una función y una lista. La función `map` aplica la función parámetro a todos los elementos de la lista. Por ejemplo:

```
? map fac [1,2,3,4,5]
[1, 2, 6, 24, 120]
? map sqrt [1.0,2.0,3.0,4.0]
[1.0, 1.41421, 1.73205, 2.0]
? map even [1..8]
[False, True, False, True, False, True, False, True]
```

Existen muchas funciones en Gofer que tienen funciones como parámetro. En el capítulo 2 se describen más funciones de este tipo. pág. 23

1.4 Definición de funciones

1.4.1 Definición por combinación

La manera más fácil de definir funciones es por combinación de otras funciones:

```
fac n = product [1..n]
impar x = not (even x)
cuadrado x = x*x
suma_de_cuadrados lista = sum (map cuadrado lista)
```

Las funciones pueden tener más de un parámetro:

```
comb n k = fac n / (fac k * fac (n-k))
formulaABC a b c = [ (-b+sqrt(b*b-4.0*a*c)) / (2.0*a)
                    , (-b-sqrt(b*b-4.0*a*c)) / (2.0*a)
                    ]
```

Las funciones sin parámetros se llaman normalmente constantes:

```
pi = 3.1415926535
e = exp 1.0
```

Toda definición de función tiene por tanto la siguiente forma:

- el nombre de la función
- los nombres de los parámetros (si existen)
- el símbolo =
- una expresión, que puede contener los parámetros, las funciones estándar y otras funciones definidas.

Una función que tiene un valor booleano como resultado, tiene a la derecha del símbolo = una expresión con un valor booleano:

```
negativo x = x < 0
positivo x = x > 0
esCero x = x == 0
```

Note la diferencia en el anterior ejemplo entre = y ==. El símbolo = separa la parte izquierda de la parte derecha de la definición. El símbolo == es un operador, como < y >.

En la definición de la función `formulaABC` se encuentran las expresiones `sqrt(b*b-4.0*a*c)` y `(2.0*a)` dos veces. Esto da mucho trabajo para teclear, pero calcular estas expresiones exige también demasiado tiempo: se calcula dos veces la misma expresión. Para evitar eso, se puede dar (en Gofer) un nombre a (sub)expresiones. La definición mejorada sería `formulaABC'`:

```
formulaABC' a b c = [ (-b+d)/n
                    , (-b-d)/n
                    ]
                    where d = sqrt (b*b-4.0*a*c)
                          n = 2.0*a
```

Con la opción de estadística del intérprete (el comando `:set +s`) se puede ver la diferencia muy bien:

```
? formulaABC' 1.0 1.0 0.0
[0.0, -1.0]
(18 reductions, 66 cells)
? formulaABC 1.0 1.0 0.0
[0.0, -1.0]
(24 reductions, 84 cells)
```

La palabra `where` no es el nombre de una función. Es una de las palabras reservadas que están en la lista de la sección 1.3.2. Detrás de `'where'` hay algunas definiciones. En el caso del ejemplo, las definiciones de las constantes `d` y `n`. Estas constantes se pueden usar en la expresión anterior a la parte de `where`. Fuera de este lugar no se pueden usar: son *definiciones locales*. Parece tal vez extraño que se llamen a `d` y `n` 'constantes', porque el valor puede ser diferente para cada vez que se llama a la función `formulaABC'`. Pero durante la computación de *una* llamada a `formulaABC'`, con valores para `a`, `b` y `c`, aquellos valores son constantes.

1.4.2 Definición por distinción de casos

Algunas veces es necesario distinguir diferentes casos en la definición de una función. Un ejemplo es la función `abs`: con un parámetro negativo la definición es diferente que con un parámetro positivo. Esto se anota en Gofer de la siguiente manera:

```
abs x | x<0 = -x
      | x>=0 = x
```

También, se pueden distinguir más de dos casos. Por ejemplo, en la función `signum`:

```
signum x | x>0 = 1
          | x==0 = 0
          | x<0 = -1
```

Las definiciones de los diferentes casos son precedidas por expresiones booleanas, que se llaman *guardas*.

Si se llama a una función definida de esta forma, se tratan las guardas una a una, hasta que se encuentre una guarda con el valor `True`. Entonces, se evalúa la expresión a la derecha del símbolo `=`. En ocasiones la última guarda es `True` o la constante `otherwise`.

Por tanto, la descripción de la forma de la definición de una función es más amplia que la explicada en la sección anterior. Una descripción más completa de la definición de una función es:

- el nombre de la función;
- los nombres de los parámetros (si existen);
- el símbolo `=` y una expresión, o una o más expresiones guardadas;
- si se desea, la palabra `where` seguida de definiciones locales.

En cada expresión guardada debe existir el símbolo |, una expresión booleana, el símbolo = y una expresión². Esta descripción de funciones todavía no está completa. . .

1.4.3 Definición por análisis de patrones

Se llaman *parámetros formales* a los parámetros de una función en la definición de la misma, por ejemplo *x* e *y* en

```
f x y = x * y
```

En una llamada a la función se dan *parámetros actuales* a la función. Por ejemplo, en la llamada

```
f 17 (1+g 6)
```

17 es el parámetro actual que corresponde a *x*, (1+g 6) es el parámetro actual que corresponde a *y*. Los parámetros actuales se sustituyen en los lugares de los parámetros formales cuando se llama a una función. El resultado del anterior ejemplo es por tanto 17*(1+g 6).

Por tanto, los parámetros actuales son *expresiones*. Los parámetros formales eran hasta ahora solamente *nombres*. En la mayoría de los lenguajes de programación, el parámetro formal tiene que ser un nombre. Existen otras posibilidades en Gofer: un parámetro formal puede ser también un *patrón*.

Un ejemplo de la definición de una función en que se usa un patrón como parámetro formal es:

```
f [1,x,y] = x+y
```

Esta función es correcta solamente si el parámetro es una lista de exactamente tres elementos, con el primer elemento 1. La función suma el segundo y el tercer elemento. La función entonces, no está definida para listas más pequeñas o más grandes, o para listas que tienen otro elemento que 1 como primer elemento. (Es normal que una función no esté definida para todos los parámetros actuales que son posibles. Por ejemplo, la función `sqrt` no está definida para parámetros negativos y el operador `/` no está definido con 0 como segundo parámetro.)

Se pueden definir funciones con diferentes patrones como parámetro formal:

```
suma []      = 0
suma [x]     = x
suma [x,y]   = x+y
suma [x,y,z] = x+y+z
```

Esta función se puede aplicar a listas con cero, uno, dos o tres elementos (en la siguiente sección se define esta función para listas con cualquier tamaño). En todos los casos la función suma los elementos. Llamando a la función, el intérprete ve si el parámetro cumple con una de las definiciones. La llamada `suma [3,4]` cumple con la tercera definición. 3 corresponde al parámetro *x* en la definición y 4 a *y*.

Las siguientes construcciones se pueden utilizar como patrón:

- Números (por ejemplo 3);
- Las constantes `True` y `False`;
- Nombres (por ejemplo, *x*);
- Listas cuyos elementos también son patrones (por ejemplo `[1,x,y]`);
- El operador `:` con patrones a la izquierda y a la derecha (por ejemplo `a:b`);
- El operador `+` con un patrón a la izquierda y un entero a la derecha (por ejemplo `n+1`);
- El operador `*` con un patrón a la derecha y un entero a la izquierda (por ejemplo `2*x`).

Con la ayuda de los patrones se pueden definir varias funciones importantes. Por ejemplo, el operador `&&` del prelude:

²Esta descripción parece por si misma una definición con una definición local para una expresión guardada!

```
False && False = False
False && True  = False
True  && False = False
True  && True  = True
```

Con el operador `:` se pueden construir listas. La expresión `x:y` significa: coloca el elemento `x` delante de la lista `y`. Al incluir el operador `:` se separa el primer elemento del resto de la lista. Con esto podemos escribir dos funciones estándar que son bastante útiles:

```
head (x:y) = x
tail (x:y) = y
```

La función `head` (cabeza) devuelve el primer elemento de una lista; la función `tail` (cola) devuelve todo excepto el primer elemento. Usos de estas funciones en expresiones son, por ejemplo:

```
? head [3,4,5]
3
? tail [3,4,5]
[4, 5]
```

Estas funciones se pueden aplicar a casi todas las listas, solamente están indefinidas para la lista vacía (la lista sin elementos).

En los patrones en que ocurren los operadores `+` o `*`, las variables tienen que ser de enteros. Por ejemplo se puede definir una función `par` que devuelve `True` si el número es par:

```
par (2*n)   = True
par (2*n+1) = False
```

Con la llamada `par 5`, solamente se cumple el segundo patrón (en este caso `n` representa 2). No se cumple el primer patrón puesto que en ese caso `n` tendría que ser 2.5, que no es un entero.

1.4.4 Definición por recursión o inducción

En la definición de una función se pueden usar las funciones estándar y las funciones definidas por el usuario. Pero también se puede usar la propia función que se define en su definición. A tal definición se la llama *definición recursiva*. La siguiente función es una función recursiva:

```
f x = f x
```

El nombre de la función que se define (`f`), está en la expresión que define la función, a la derecha del símbolo `=`. Sin embargo, esta definición no tiene sentido. Para calcular el valor de `f 3`, se tiene que calcular el valor de `f 3` primero y para eso, se tiene que calcular el valor de `f 3`, etc. . .

Las funciones recursivas tienen sentido bajo las siguientes dos condiciones:

- el parámetro de la llamada recursiva es *más simple* que el parámetro de la función que se quiere definir (por ejemplo, numéricamente menor o una lista más corta);
- existe una definición no recursiva para un caso base.

Una definición recursiva de la función factorial es:

```
fac n | n==0 = 1
      | n>0  = n * fac (n-1)
```

Aquí el caso base es `n==0`; en este caso se puede calcular el resultado directamente (sin recursión). En el caso `n>0` existe una llamada recursiva, es decir `fac (n-1)`. El parámetro de esta llamada (`n-1`) es menor que `n`.

Otra manera de distinguir estos dos casos (caso base y caso recursivo) es usando patrones:

```

fac 0      = 1
fac (n+1) = (n+1) * fac n

```

También en este caso el parámetro de la llamada recursiva (n) es menor que el parámetro de la función que se quiere definir ($n+1$).

El uso de patrones tiene mucha relación con la tradición matemática del usar inducción. Por ejemplo, la definición matemática de la función potencia puede ser usada casi directamente como función en Gofer:

```

x ^ 0      = 1
x ^ (n+1) = x * x^n

```

Una definición recursiva en que se usan patrones para distinguir diferentes casos (en lugar de expresiones booleanas) se llama también *definición inductiva*.

Las funciones sobre listas también pueden ser recursivas. Una lista es ‘menor’ que otra si tiene menos elementos. La función `suma` que hemos visto en la sección anterior, puede ser definida de diferentes maneras. Una definición recursiva con expresiones booleanas es:

```

suma lista | lista==[] = 0
           | otherwise = head lista + suma (tail lista)

```

Pero también es posible obtener una versión inductiva (usando patrones):

```

suma []           = 0
suma (cabeza:cola) = cabeza + suma cola

```

En la mayoría de los casos, es más clara una definición con patrones, porque las diferentes partes en el patrón pueden conseguir un nombre directamente (como `cabeza` y `cola` en la función `suma`). En la versión recursiva de `suma` se necesitan las funciones estándar `head` y `tail` para distinguir las diferentes partes en `lista`. En estas funciones también se usan patrones.

La función estándar `length`, que calcula el número de elementos de una lista, puede ser definida inductivamente:

```

length []           = 0
length (cabeza:cola) = 1 + length cola

```

En este caso, el valor del primer elemento de la lista no importa, solamente importa que exista un primer elemento.

En patrones se permite usar el símbolo ‘_’ en este tipo de casos en vez de un nombre:

```

length []           = 0
length (_:cola) = 1 + length cola

```

1.4.5 Indentación y comentarios

En la mayoría de los lugares de un programa, se pueden poner más blancos para hacer el programa más claro. En los anteriores ejemplos, hemos añadido más espacio, para ordenar las definiciones de funciones. Está claro que no se pueden poner blancos en un nombre de una función o en un número. `len gth` es algo diferente de `length`, y `1 7` es algo diferente de `17`.

También se puede añadir una nueva línea para hacer el resultado claro. Hemos hecho esto en la definición de `formulaABC`, porque sin ello la línea sería muy larga.

Pero, a diferencia de otros lenguajes de programación, una nueva línea tiene un significado. Véanse por ejemplo las siguientes construcciones:

```

where
  a = f x y
  b = g z

```

```

where
  a = f x
  y b = g z

```

Entre estas dos construcciones existe bastante diferencia.

En una serie de definiciones, Gofer usa la siguiente manera para decidir que es parte de que:

- una línea indentada *exactamente tanto como* la línea anterior se considera una nueva definición.
- una línea indentada *más* que la anterior, es parte de la línea anterior.
- una línea indentada *menos* que la anterior, no es parte de las definiciones anteriores.

El último caso es importante si la construcción de `where` está en otra definición de `where`. Por ejemplo en

```

f x y = g (x+w)
  where g u = u + v
        where v = u * u
        w = 2 + y

```

`w` es una declaración local de `f`, y no de `g`. La definición de `w` está a la izquierda de la definición de `v`; por tanto no forma parte de la construcción del `where` de `g`. Deja tanto espacio libre como la definición de `g`, por tanto es parte de la construcción del `where` de `f`. Si dejara menos espacio libre, no sería parte de tal construcción. En este caso, Gofer daría un mensaje de error.

Tal vez estas reglas parezcan un poco complicadas, pero no es difícil si se usa la siguiente regla global:

definiciones equivalentes están igualmente indentadas

Por tanto, todas las definiciones de funciones tienen que empezar en la misma columna (por ejemplo, todas en la posición cero).

Comentarios

En cualquier lugar donde se puedan poner blancos, se puede poner un comentario. El intérprete ignora todo comentario. Éste solamente es útil para los lectores del programa. Existen dos tipos de comentarios en Gofer:

- con los símbolos `--` empieza un comentario, que termina al finalizar la línea.
- con los símbolos `{- y -}` empieza el comentario, que termina con los símbolos `-}`.

Una excepción a la primera regla es el caso que `--` sea parte de un operador, por ejemplo en `<-->`. La combinación `--` fué reservada (en el párrafo 1.3.2).

Los comentarios con `{- y -}` se pueden anidar. Cada comentario termina con el correspondiente símbolo de terminación. Por ejemplo, en

```
{- {- hola -} f x = 3 -}
```

no se define una función `f`, todo es comentario.

1.5 Tipado

1.5.1 Clases de errores

Al escribir una función se pueden cometer errores. El intérprete puede avisar de algunos errores. Si la definición de una función no es correcta, el intérprete da un mensaje en el momento de analizar la definición. Por ejemplo, la siguiente definición contiene un error:

```
esCero x = x=0
```

El segundo = tenía que ser el operador == (= significa 'se define como' y == significa 'es igual a'). Al analizar esta definición el intérprete dice:

```
Reading script file "nuevo":
Parsing.....
ERROR "nuevo" (line 18): Syntax error in input (unexpected '=')
```

El intérprete encuentra los errores de sintaxis en el primer paso del análisis: el análisis sintáctico (en inglés: *parsing*). Otros errores pueden ser un paréntesis izquierdo sin su paréntesis derecho correspondiente, o el uso de palabras reservadas (como **where**) en lugares en los que no se permiten.

Además de los errores de sintaxis, existen otros errores que el intérprete puede encontrar. Una posibilidad es usar una función que no está definida. Por ejemplo, con:

```
fac x = producto [1..x]
```

el intérprete dice:

```
Reading script file "nuevo":
Parsing.....
Dependency analysis.....
ERROR "nuevo" (line 19): Undefined variable "producto"
```

Estos errores se encuentran durante el segundo paso: el análisis de dependencia (*dependency analysis*).

El siguiente obstáculo en el análisis es la comprobación de los tipos (*type checking*). Por ejemplo, funciones que operan sobre números no pueden ser usadas con valores booleanos y tampoco con listas.

Con la expresión `1+True` en la definición de una función el intérprete dice:

```
Reading script file "nuevo":
Parsing.....
Dependency analysis...
Type checking.....
ERROR "nuevo" (line 22): Type error in application
*** expression      : 1 + True
*** term            : 1
*** type            : Int
*** does not match : Bool
```

La subexpresión (*term*) `1` tiene el *tipo* `Int` (*entero*). No se puede sumar tal valor a `True`, que es de tipo `Bool`.

Otros errores de tipado se producen si se aplica la función `length` a algo diferente de una lista, como en `length 3`:

```
ERROR: Type error in application
*** expression      : length 3
*** term            : 1
*** type            : Int
*** does not match : [a]
```

Sólamente si no existen errores de tipado en el programa, el intérprete ejecuta el cuarto paso (generación de código).

El intérprete da todos los mensajes de error en el momento de analizar una función. En algunos otros lenguajes se comprueba el tipado en el momento en que se llama a una función. En este caso nunca se está seguro si existen errores de tipado o no. Si el intérprete de Gofer no da mensajes de error, se puede estar seguro de que no existen errores de tipado.

Pero, si el intérprete de Gofer no da mensajes de error, es posible todavía encontrar errores en el programa. Si se pone (por ejemplo) en la definición de la función `suma` el signo menos en vez del signo más, el programa no funciona bien, pero el intérprete no lo sabe. Estos errores, ‘errores lógicos’, son los más difíciles de encontrar.

1.5.2 Expresiones de tipo

El tipo de una expresión se puede determinar con el comando del intérprete `:type` seguido de la expresión de la que se quiere conocer el tipo. Por ejemplo:

```
? :type 3+4
3 + 4 :: Int
```

El símbolo `::` significa: ‘es del tipo’. La expresión no se evalúa con el comando `:type`, solamente se determina el tipo.

Existen cuatro tipos estándar:

- **Int**: el tipo de los enteros;
- **Float**: el tipo de los números de punto flotante (*floating-point*);
- **Bool**: el tipo de los valores booleanos: **True** y **False**;
- **Char**: el tipo de los caracteres (se tratarán en el párrafo 3.2.2).

pág. 49

Estos tipos deben ser escritos comenzando con mayúscula.

Las listas pueden ser de diferentes tipos. Existen por ejemplo, listas de enteros, listas de valores booleanos y listas de listas de enteros. Todas estas listas tienen un tipo distinto:

```
? :type [1,2,3]
[1,2,3] :: [Int]
? :type [True,False]
[True,False] :: [Bool]
? :type [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

El tipo de una lista se denota con el tipo de los elementos de la lista entre corchetes: `[Int]` es el tipo de una lista de enteros. Todos los elementos de una lista tienen que ser del mismo tipo. Si no, se obtiene un mensaje de error de tipo:

```
? [1,True]
ERROR: Type error in list
*** expression   : [1,True]
*** term         : True
*** type         : Bool
*** does not match : Int
```

También las funciones tienen un tipo. El tipo de una función está determinado por el tipo del parámetro y el tipo del resultado. Por ejemplo, el tipo de la función `sum` es:

```
? :type sum
sum :: [Int] -> Int
```

La función `sum` opera sobre listas de enteros y como resultado devuelve un solo entero. El símbolo `->` en el tipo de la función representa una flecha (\rightarrow).

Otros ejemplos de tipos de funciones son:

```
sqrt :: Float -> Float
even :: Int   -> Bool
sums  :: [Int] -> [Int]
```

Se suele decir: “`even` tiene el tipo de `int` a `bool`”.

Es posible crear listas de funciones, si estas funciones (como números, valores booleanos y listas) son de un mismo tipo, es posible hacer listas de funciones. Las funciones en la lista tienen que tener el mismo tipo, ya que todos los elementos de una lista tienen que tener el mismo tipo. Un ejemplo de una lista de funciones es:

```
? :type [sin,cos,tan]
[sin,cos,tan] :: [Float -> Float]
```

Las tres funciones `sin`, `cos` y `tan` son todas funciones de `float` a `float`, en consecuencia pueden ser elementos de una lista que tiene el tipo ‘lista de funciones de `float` a `float`’.

El intérprete mismo puede determinar el tipo de una expresión. Esto se realiza cuando el intérprete comprueba el tipado de un programa. Sin embargo, está permitido escribir el tipo de una función en el programa. La definición de función se realizaría de la siguiente forma:

```
sum      :: [Int] -> Int
sum []   = 0
sum (x:xs) = x + sum xs
```

Aunque la declaración del tipo es superflua, tiene dos ventajas:

- se comprueba si la función tiene el tipo que está declarado.
- la declaración del tipo ayuda a entender la función.

No hace falta escribir la declaración directamente delante la definición. Por ejemplo, se pueden escribir primero las declaraciones de los tipos de todas las funciones que están definidas en el programa. Las declaraciones en este caso funcionan más o menos como índice.

1.5.3 Polimorfismo

Para algunas funciones sobre listas no interesa el tipo de los elementos de la lista. La función estándar `length` puede determinar el tamaño de una lista de enteros, pero también de una lista de valores booleanos, y –por qué no– de una lista de funciones. El tipo de la función `length` se define como sigue:

```
length :: [a] -> Int
```

Este tipo indica que esta función se aplica sobre una lista, pero que el tipo de los elementos de esta lista no importa. Este tipo se indica por medio de una *variable de tipo*, en el ejemplo se utiliza la letra `a`. Variables de tipo se escriben con una letra minúscula, al contrario que los tipos fijos como `Int` y `Bool`.

La función `head`, que devuelve el primer elemento de una lista, tiene el tipo:

```
head :: [a] -> a
```

Esta función se aplica también a listas, sin que importe cual sea el tipo de los elementos. Pero su resultado es del tipo de los elementos de la lista (es el primer elemento de la lista). Por tanto, se usa para el tipo del resultado la misma variable que para el tipo de los elementos de la lista.

Un tipo que contiene variables de tipo, se llama *tipo polimórfico*. Las funciones con un tipo polimórfico se llaman funciones polimórficas. El fenómeno en si se llama *polimorfismo*.

Las funciones polimórficas, como `length` y `head`, tienen en común que usan solamente la *estructura* de una lista. Una función no polimórfica como `sum`, usa también propiedades de los *elementos* de la lista (como la posibilidad de sumarlos, etc.).

Debido a que se pueden usar funciones polimórficas en diferentes situaciones, muchas funciones estándar en el preludio son funciones polimórficas.

No solamente las funciones sobre listas pueden ser polimórficas. La función polimórfica más simple es la función identidad (la función que devuelve su parámetro):

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

La función `id` se puede aplicar a elementos de cualquier tipo (el resultado es del mismo tipo). Por tanto, las siguientes llamadas `id 3`, `id True`, `id [True,False]`, `id [[1,2,3],[4,5]]` y `id id` son posibles. (`id id` devuelve `id`.)

1.5.4 Funciones con más de un parámetro

pág. 5

Aquellas funciones con más de un parámetro también tienen un tipo. En el tipo existe una flecha entre los parámetros y entre el último parámetro y el resultado. La función `comb` en el párrafo 1.2.2 tiene dos parámetros enteros y un resultado entero. Por tanto su tipo es:

```
comb :: Int -> Int -> Int
```

pág. 11

La función `formulaABC` en el párrafo 1.4.1 tiene tres números reales como parámetros y una lista de números reales como resultado. Por eso el tipo es:

```
formulaABC :: Float -> Float -> Float -> [Float]
```

pág. 11

En el párrafo 1.3.6 hemos visto la función `map`. Esta función tiene dos parámetros: una función y una lista. La función se aplica a todos los elementos de la lista, de modo que el resultado también es una lista. El tipo de `map` es como sigue:

```
map :: (a->b) -> [a] -> [b]
```

El primer parámetro de `map` es una función (llamemos a esta función ‘función parámetro’) entre tipos cualesquiera (`a` y `b`), no hace falta que sean iguales. El segundo parámetro de la función es una lista. Sus elementos tienen que ser del tipo del parámetro de la función parámetro (se aplica la función a los elementos de la lista). El resultado de `map` es una lista cuyos elementos son del mismo tipo (`b`) que el resultado de la función parámetro.

En la declaración del tipo de `map`, el tipo del primer parámetro (`a->b`) tiene que estar entre paréntesis. Si no fuese así, se entendería que `map` tiene tres parámetros: `a`, `b` y `[a]`; y a `[b]` como resultado. Este no sería el tipo correcto, ya que `map` tiene dos parámetros: (`a->b`) y `[a]`.

También los operadores tienen un tipo. Los operadores son simplemente funciones de dos parámetros que se colocan en una posición distinta (entre los parámetros y no delante). Para el tipo no existe diferencia. Por ejemplo el tipo del operador `&&` es:

```
(&&) :: Bool -> Bool -> Bool
```

1.5.5 Sobrecarga

El operador `+` se puede aplicar a dos enteros (`Int`) o a dos números reales (`Float`). El resultado es del mismo tipo. Por tanto, el tipo de `+` puede ser `Int->Int->Int` o `Float->Float->Float`. Sin embargo `+` no es realmente un operador polimórfico. Si el tipo fuera `a->a->a`, entonces el operador tendría que funcionar con parámetros de `Bool`, lo que no es posible.

Tal función, que es polimórfica de forma restringida, se llama función *sobrecargada* (en inglés: *overloaded function*). Para dar tipos a funciones u operadores sobrecargados, se clasifican los tipos en clases (en inglés: *classes*). Una clase es un grupo de tipos con una característica en común. En el preludio están definidas algunas clases:

- `Num` es la clase de tipos cuyos elementos se pueden sumar, multiplicar, restar y dividir (tipos numéricos);
- `Ord` es la clase de tipos cuyos elementos se pueden ordenar;
- `Eq` es la clase cuyos elementos se pueden comparar (en inglés: *equality types*).

El operador `+` tiene el siguiente tipo:

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

Se debe leer: '`+` tiene el tipo `a->a->a`, donde `a` es un tipo en la clase `Num`'.

Cuidado con el símbolo `=>` (`o ⇒`), que es fundamentalmente diferente del símbolo `->`.

Otros ejemplos de operadores sobrecargados son:

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

Las funciones definidas por el usuario también pueden ser sobrecargadas. Por ejemplo, la función:

```
cuadrado x = x * x
```

tiene el tipo

```
cuadrado :: Num a => a -> a
```

porque el operador `*` que se usa, es sobrecargado.

El uso de clases y las definiciones de clases se discutirán más adelante. Las hemos mencionado aquí para poder tipar operadores y funciones sobrecargadas.

Ejercicios

Ejercicio 1.1

Escriba una función que cuente cuantos números negativos existen en una lista.

Ejercicio 1.2

Escriba una función `diag` que tenga una lista de caracteres como parámetro y que dé como resultado los caracteres en una diagonal. Ejemplo:

```
? diag ['a','b','c','d','e']
a
 b
  c
   d
    e
```

Ejercicio 1.3

Escriba una función `cuadrado` que dada una lista de caracteres, presente tantas copias de esta serie de caracteres (cada copia en una nueva línea), de manera que el número de las letras en horizontal sea igual al número de las letras que hay verticalmente. Tenga en cuenta que una cadena de caracteres es en realidad una lista de caracteres. Ejemplo:

```
? cuadrado "abcde02"
abcde02
abcde02
abcde02
abcde02
abcde02
abcde02
abcde02
```

Ejercicio 1.4

Escriba una función `dividir`, de manera que dada una lista de caracteres dé como resultado otra lista, pero ahora dividida en líneas. Cada vez que haya dos caracteres seguidos que sean iguales se insertará en el resultado una nueva línea (entre los dos caracteres iguales):

```
? dividir "abbcddeffabdde"  
ab  
bcdef  
f  
fabd  
de
```

El problema aquí es cómo comparar los dos elementos seguidos. ¿Puede usar la función `map` para esto? Una posibilidad es usar patrones sobre una lista (lista vacía y lista con un elemento o más). Con el patrón tiene en este caso el primer elemento de la lista y su cola. Con la función `head` puede conseguir el primer elemento de la cola. . .

Ejercicio 1.5

Escriba una función `noDoble` que, dada una cadena de caracteres, dé como resultado esta lista de caracteres, pero cuando existan dos o más caracteres seguidos iguales en la lista, en el resultado esté solamente una vez este carácter:

```
? noDoble "abccccfabaddeff"  
abcfabadeff
```

Capítulo 2

Números y funciones

2.1 Operadores

2.1.1 Operadores como funciones y viceversa

Un operador es una función de dos parámetros, que se escribe entre éstos en vez de delante. Los nombres de funciones se forman con letras y cifras; los nombres de operadores se forman con símbolos (ver el párrafo 1.3.2 para saber como son las reglas exactamente).

pág. 7

Algunas veces es deseable escribir un operador delante de sus parámetros, o una función entre sus parámetros. En Gofer existen dos notaciones especiales para esto:

- un operador entre paréntesis se comporta como la función correspondiente;
- una función entre *comillas inversas* se comporta como el operador correspondiente.

(Una comilla inversa es el símbolo ```, no se debe confundir con el `'`, el *apóstrofo*.)

Se permite escribir `(+) 1 2` en vez de `1+2`. Esta notación se usa también para declarar el tipo del operador `+`:

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

Delante del símbolo `::` debe existir una expresión. Un operador suelto no es una expresión, una función sí.

Por otro lado, también es posible escribir `1 `f` 2` en vez de `f 1 2`. Esto se usa sobre todo para hacer una expresión más clara; la expresión `3 `comb` 5` se lee más fácilmente que `comb 3 5`. Claro que esto se puede solamente si la función tiene dos parámetros.

2.1.2 Precedencias

En matemáticas, la multiplicación tiene mayor precedencia que la suma. También Gofer conoce estas precedencias: la expresión `2*3+4*5` tiene el valor 26, y no 50, 46 o 70.

Existen en Gofer más niveles de precedencias. Los operadores de comparación, como `<` y `==`, tienen una precedencia menor que los operadores aritméticos. Así por ejemplo, `3+4<8` tiene el sentido que se espera “se compara 3+4 con 8 (el resultado es `False`)”, y no “se suma 3 al resultado de 4<8 (el resultado sería un error de tipado)”.

En total, existen nueve niveles de precedencias. Los operadores del prelude tienen las siguientes precedencias:

```

nivel 9  ., !!
nivel 8  ^
nivel 7  *, /, 'div', 'rem', 'mod'
nivel 6  +, -
nivel 5  :, ++, \
nivel 4  ==, /=, <, <=, >, >=, 'elem', 'notElem'
nivel 3  &&
nivel 2  ||
nivel 1  (no se usa en el preludio)

```

(No conocemos todavía todos estos operadores, sobre algunos se discute en éste, o en un próximos capítulos.)

Para apartarse de las precedencias fijas se pueden utilizar paréntesis en una expresión, para agrupar las subexpresiones que se quieran calcular primero: por ejemplo, en `2*(3+4)*5` se calcula primero `3+4`.

La llamada a funciones (el operador invisible entre `f` y `x` en `f x`) tiene la máxima precedencia. La expresión `cuadrado 3 + 4` calcula el cuadrado de 3, y suma al resultado 4. Si se escribe `cuadrado 3+4`, también se llama primero a la función y después se calcula la suma. Para calcular el cuadrado de 7, se necesitan paréntesis para romper la precedencia alta de la llamada a función: `cuadrado (3+4)`.

pág. 13

También en la definición de funciones por análisis de patrones (ver el párrafo 1.4.3) es importante observar que la llamada a función tiene la precedencia más alta. En la definición

```

sum []      = 0
sum (x:xs) = x + sum xs

```

son esenciales los paréntesis alrededor de `x:xs`. Sin paréntesis, esto se interpreta como `(sum x):xs`, y éste no es un patrón válido.

2.1.3 Asociatividad

Con las reglas de precedencias no está claro lo que tiene que hacerse con los operadores de igual precedencia. Para sumar no importa, pero, por ejemplo, importa mucho para restar: ¿es el resultado de `8-5-1` el valor 2 (primero 8 menos 5, y después menos 1), o 4 (primero 5 menos 1, y después 8-4)?

Para cada operador en Gofer se define en qué orden se tiene que calcular. Para un operador, por ejemplo \oplus , existen cuatro posibilidades:

- el operador \oplus *asocia por la izquierda*, esto significa que $a \oplus b \oplus c$ se calcula como $(a \oplus b) \oplus c$;
- el operador \oplus *asocia por la derecha*, esto significa que $a \oplus b \oplus c$ se calcula como $a \oplus (b \oplus c)$;
- el operador \oplus es *asociativo*, esto significa que no importa en qué orden se calcule $a \oplus b \oplus c$;
- el operador \oplus es *no asociativo*, esto significa que no se permite escribir $a \oplus b \oplus c$; siempre se tiene que indicar con paréntesis cuál es el orden del cálculo.

Para los operadores del preludio, se eligió su asociatividad según la tradición matemática. Cuando no estuvo claro, se hicieron los operadores no asociativos. Para los operadores asociativos, se ha elegido entre asociativo por la izquierda o por la derecha. (Se ha elegido el orden más eficiente. No importa para el resultado.)

Los siguientes operadores **asocian por la izquierda**:

- el operador invisible ‘aplicación de una función’; por tanto `f x` y significa `(f x)` y (la razón para esto se explica en la sección 2.2);
- el operador `!!` (ver el párrafo 3.1.2);
- el operador `-`; por tanto el valor de `8-5-1` es 2 (como de costumbre en matemáticas) y no 4.

Los siguientes operadores **asocian por la derecha**:

- el operador `^` (potencia); por tanto el valor de `2^2^3` es $2^8 = 256$ (como de costumbre en matemáticas) y no es $4^3 = 64$;
- el operador `:` (‘coloca un nuevo elemento como cabeza de una lista’), de modo que `1:2:3:x` es una lista que empieza con los valores 1, 2 y 3.

pág. 25

pág. 43

Los siguientes operadores son **no asociativos**:

- el operador / y los operadores relacionados `div`, `rem` y `mod`; por tanto el resultado de la expresión `64/8/2` no es 4, ni 16, sino que devuelve el mensaje de error:

```
ERROR: Ambiguous use of operator "/"with "/"
```

- el operador `\|`;
- los operadores de comparación `==`, `<` etc.: Si quiere comprobar si `x` está entre 2 y 8, entonces no escriba `2<x<8`, sino `2<x && x<8`.

Los siguientes operadores son **asociativos**:

- Los operadores `*` y `+` (se computan estos operadores como operadores que asocian por la izquierda, según la tradición en matemáticas);
- los operadores `++`, `&&` y `||` (se computan estos operadores como operadores que asocian por la derecha, porque es más eficiente);
- el operador `.` (ver el párrafo 2.3.3).

pág. 30

2.1.4 Definición de operadores

Quien defina un operador debe indicar cuales son su precedencia y asociatividad. En el prelude está descrito de la siguiente manera que el `^` tiene el nivel de precedencia 8 y asocia por la derecha:

```
infixr 8 ^
```

Para operadores asociativos por la izquierda, existe la palabra reservada `infixl`, y para operadores no asociativos existe la palabra `infix`:

```
infixl 6 +, -
infix 4 ==, /=, 'elem'
```

El elegir precedencias inteligentemente puede reducir el uso de paréntesis en expresiones. Véase otra vez el operador '`n comb k`' del párrafo 1.2.2:

```
n 'comb' k = fac n / (fac k * fac (n-k))
```

o con un símbolo:

```
n !^! k = fac n / (fac k * fac (n-k))
```

Tal vez quiera calcular en algún momento $\binom{a+b}{c}$. Para esto, es mejor dar a '`comb`' una precedencia menor que la precedencia de `+`; entonces se puede escribir `a+b'comb'c` sin paréntesis. Por otro lado, se quieren calcular también expresiones como $\binom{a}{b} < \binom{c}{d}$. Si damos a '`comb`' una precedencia mayor que la precedencia de `<`, no se necesitarán tampoco paréntesis para estas expresiones.

La mejor precedencia para el operador '`comb`' es por tanto 5 (menor que `+` (6), pero mayor que `<` (4)). Porque no es costumbre calcular una expresión de la forma `a'comb'b'comb'c`, lo mejor es hacer el operador no asociativo. La definición de la precedencia resulta en:

```
infix 5 !^!, 'comb'
```

2.2 Currificación

2.2.1 Instanciar parcialmente

Se supone que `mas` es una función que suma dos enteros. Esta función puede recibir dos parámetros en una expresión, por ejemplo `mas 3 5`.

pág. 5

También se pueden dar *menos* parámetros a una función. Si `mas` recibe solamente un parámetro, por ejemplo, `mas 1`, entonces el resultado es una función que espera un parámetro. Se puede usar esta función para definir otra función, por ejemplo, la función `inc` (de incrementar):

```
inc :: Int -> Int
inc = mas 1
```

Llamar a una función con menos parámetros de los que ésta espera se denomina *instanciar parcialmente*.

Otra aplicación de una función instanciada parcialmente es que puede funcionar como parámetro para otra función. Por ejemplo, el parámetro función de la función `map` (que aplica una función a todos los elementos de una lista), es muchas veces una función instanciada parcialmente.

```
? map (mas 5) [1,2,3]
[6, 7, 8]
```

La expresión `mas 5` se puede interpretar como ‘la función que suma 5 a algo’. En el ejemplo, esta función es aplicada a todos los elementos de la lista `[1,2,3]` por la función `map`.

Con la posibilidad de instanciar parcialmente sabemos un poco más sobre los tipos de las funciones. Si `mas 1`, al igual que `inc`, tiene el tipo `Int->Int`, entonces `mas` parece ser una función de `Int` (el tipo de 1) a ese tipo:

```
mas :: Int -> (Int->Int)
```

Por decidir que `->` asocia por la derecha, no necesitamos los paréntesis:

```
mas :: Int -> Int -> Int
```

Ésta es exactamente la notación para el tipo de una función de dos parámetros, que fue descrita en el párrafo 1.5.4.

pág. 20

En realidad, no existen ‘funciones de dos parámetros’. Existen solamente funciones de un parámetro. Estas funciones pueden devolver otra función que tiene un parámetro. De este manera, parece que la función original tiene dos parámetros.

Esta estrategia se denomina *currificación* (describir las funciones de más de un parámetro por funciones de un parámetro que devuelven otra función), y se debe a Haskell Curry, matemático inglés. La función misma se llama función currifcada. (Este homenaje no es totalmente justificado, ya que este método fue usado con anterioridad por M. Schönfinkel).

2.2.2 Paréntesis

El operador invisible ‘aplicación de función’ asocia por la izquierda. Esto significa: el intérprete lee la expresión `mas 1 2` como `(mas 1) 2`. Eso cumple exactamente con el tipo de `mas`: es una función que espera un entero (1 en el ejemplo) y que retorna una función que espera un entero (2 en el ejemplo).

La asociatividad por la derecha de aplicación de función sería estúpida: en `mas (1 2)` se aplicaría primero 1 a 2 (!?) y después `mas` al resultado.

Si existe una serie de nombres en una expresión, entonces, el primero tiene que ser una función que acepte a los siguientes como parámetros:

```
f a b c d
```

se interpreta como:

```
((((f a) b) c) d)
```

Si a es del tipo A , b del tipo B , etcétera, entonces el tipo de f es:

```
f :: A -> B -> C -> D -> E
```

o, si se escriben todos los paréntesis:

```
f :: A -> (B -> (C -> (D -> E)))
```

Sin paréntesis, esto es mucho más claro. Se ha elegido la asociatividad de \rightarrow y la aplicación de función de tal manera que no se ve la currificación: la aplicación de función asocia por la izquierda y \rightarrow asocia por la derecha. Por tanto, la regla es: *si no están escritos los paréntesis, entonces están escritos implícitamente de tal manera que funciona la currificación*. Solamente se necesitan paréntesis si se quiere modificar esto. Eso pasa, por ejemplo, en los siguientes casos:

- En el tipo, si la función tiene una función como *parámetro* (con currificación, una función tiene una función como *resultado*). Por ejemplo, el tipo de `map` es:

```
map :: (a->b) -> [a] -> [b]
```

Los paréntesis en $(a \rightarrow b)$ son esenciales, de otra manera, `map` parecería una función con tres parámetros.

- En una expresión, si se quiere dar *el resultado* de una función a otra función, y no la función misma. Por ejemplo, si se quiere calcular el cuadrado del seno de un número:

```
cuadrado (sin pi)
```

Sin paréntesis, primero se aplicaría `cuadrado` a `sin (??)` y el resultado se aplicaría a `pi`.

2.2.3 Secciones de operadores

Para instanciar parcialmente los operadores, existen dos notaciones especiales:

- con $(\oplus x)$, se instancia parcialmente el operador \oplus con x como parámetro *por la derecha*;
- con $(x \oplus)$, se instancia parcialmente el operador \oplus con x como parámetro *por la izquierda*.

Estas notaciones se llaman *secciones de operadores*.

Se puede definir algunas funciones con secciones de operadores:

```
inc           = (+1)
dosVeces     = (2*)
mitad        = (/2.0)
inverso      = (1.0/)
cuadrado     = (^2)
2aLaPotenciaDe = (2^)
unDigito     = (<=9)
esCero       = (==0)
```

La aplicación más importante de secciones de operadores es el uso de un operador instanciado parcialmente como parámetro de una función:

```
? map (2*) [1,2,3]
[2, 4, 6]
```

2.3 Funciones como parámetros

2.3.1 Funciones sobre listas

En un lenguaje de programación funcional, las funciones se comportan de muchas maneras tal como otros valores (listas, números etc.). Por ejemplo las funciones

- tienen un *tipo*.

- pueden retornar otras funciones como *resultado* (algo que se usa mucho con la currificación).
- pueden ser *parámetros* de otras funciones.

Con la última posibilidad, se pueden escribir funciones globales, cuyo resultado depende de una función (o más) que es uno de los parámetros de la función global.

A las funciones que tienen funciones como parámetros se las llama *funciones de orden superior*, para distinguirlas de funciones numéricas simples.

La función `map` es un ejemplo de una función de orden superior. Esta función está basada en el principio general ‘recorrer todos los elementos de una lista’. La función parámetro de `map` se aplica a cada elemento de la lista.

Se puede definir la función `map` de la siguiente manera:

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

En la definición se usan patrones y se distinguen dos casos: o el segundo parámetro es una lista sin elementos, o es una lista con por lo menos un elemento (`x` es el primer elemento, `xs` es el resto). Es una función recursiva: si la lista no está vacía, entonces se llama a la función `map` otra vez. El parámetro es más corto en este caso (`xs` en vez de `x:xs`), de modo que en última instancia se utilizará la parte no recursiva de la función.

Otra función sobre listas de orden superior que se usa muchas veces es `filter` (*filtro*). Esta función devuelve los elementos de una lista que cumplen alguna condición. Esta condición se da a la función `filter` en forma de una función booleana. Por ejemplo:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

Si los elementos de la lista son del tipo `a`, entonces la función que represente la condición es del tipo `a->Bool`. También la definición de `filter` es recursiva:

```
filter      :: (a->Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                 | otherwise = filter p xs
```

Existen dos casos: o la lista está vacía, o no. Si la lista no está vacía, entonces si el primer elemento de la lista (`x`) cumple con la condición `p`, el elemento `x` es parte de la lista resultante. Se filtran los otros elementos con una llamada recursiva de `filter`.

Las funciones útiles de orden superior pueden buscarse viendo las correspondencias entre definiciones de funciones. Véanse, por ejemplo, las definiciones de las funciones `sum` (que calcula la suma de los elementos de una lista), `product` (que calcula el producto de los elementos de una lista) y `and` (que calcula si todos los elementos de una lista son `True`):

```
sum [] = 0
sum (x:xs) = x + sum xs
product [] = 1
product (x:xs) = x * product xs
and [] = True
and (x:xs) = x && and xs
```

La estructura de estas tres definiciones es la misma. Difieren en el resultado si la lista está vacía y en el operador que se usa para conectar el primer elemento al resultado de la llamada recursiva (`+`, `*` o `&&`).

Una función de orden superior puede ser construida representando estas dos situaciones mediante parámetros:

```
foldr op e []      = e
foldr op e (x:xs) = x 'op' foldr op e xs
```

Dada esta función, las otras tres funciones pueden ser definidas instanciando parcialmente la función global (`foldr`):

```
sum      = foldr (+) 0
product = foldr (*) 1
and      = foldr (&&) True
```

Porque la función `foldr` es muy útil, ha sido definida en el prelude.

Se puede explicar el nombre de la función `foldr` (*plegar*). El valor de

```
foldr (+) e [w,x,y,z]
```

es igual al valor de la expresión

```
(w + (x + (y + (z + e))))
```

La función `foldr`, ‘pliega’ la lista a un valor colocando un operador (uno de los dos parámetros), entre los elementos de la lista. La función empieza por la derecha con el valor inicial (el otro parámetro). Existe también una función `foldl`, que empieza por la izquierda.

Las funciones de orden superior como `map` y `foldr`, tienen similar papel en lenguajes funcionales que las estructuras de control (como `for` y `while`) tienen en lenguajes imperativos. Sin embargo, las estructuras de control son primitivas en estos lenguajes, mientras que en lenguajes funcionales uno puede definir estas funciones. Esto hace a los lenguajes funcionales más flexibles: hay pocas primitivas, pero uno puede definir todo.

2.3.2 Iteración

Se usa *iteración* muchas veces en matemáticas. Significa: toma un valor inicial y aplica a éste una función hasta que el resultado cumpla con una condición.

Se puede describir la iteración perfectamente con una función de orden superior. En el prelude se llama esta función `until`. El tipo es:

```
until :: (a->Bool) -> (a->a) -> a -> a
```

La función tiene tres parámetros: la condición que dice como tiene que ser el resultado final (una función `a->Bool`), la función que se aplica cada vez (una función `a->a`), y el valor inicial (del tipo `a`). El resultado final también es del tipo `a`.

La definición de `until` es recursiva. En este caso, no se usan patrones para distinguir el caso recursivo del caso no recursivo, sino que se usan expresiones booleanas:

```
until p f x | p x      = x
            | otherwise = until p f (f x)
```

Si el valor inicial (`x`) cumple directamente con la condición `p`, entonces el valor inicial también es el resultado final.

Como todas las funciones de orden superior, se puede llamar a `until` con funciones instanciadas parcialmente. La siguiente expresión calcula la primera potencia de dos mayor que 1000 (empieza con 1 y duplica hasta que el resultado es mayor que 1000):

```
? until (>1000) (2*) 1
1024
```

El parámetro de la llamada recursiva no siempre es menor que el parámetro formal. Esto es porque `until`, no siempre da un resultado. Con la llamada `until (<0) (+1) 1`, el cálculo nunca terminará porque el resultado nunca cumplirá con la condición.

Si el ordenador nunca da una respuesta es porque está en una recursión infinita. La computación puede ser detenida tecleando la tecla 'CTRL' junto con la tecla 'c':

```
? until (<0) (+1) 1
ctrl-C
{Interrupted!}
?
```

2.3.3 Composición

Si f y g son funciones, entonces $g \circ f$ es la notación matemática para ' g compuesta con f ': la función que aplica primero f , y después aplica g al resultado. También en Gofer el operador que compone dos funciones es muy útil. Si existe un operador '`despues`', entonces es posible definir lo siguiente:

```
impar      = not 'despues' par
casiCero   = (<1) 'despues' abs
```

Se puede definir el operador '`despues`' como operador de precedencia grande:

```
infixr 8 'despues'
g 'despues' f = h
  where h x = g (f x)
```

No se pueden hacer composiciones de todas las funciones. El rango de f tiene que ser el mismo que el dominio de g . Si f es una función $a \rightarrow b$, entonces g puede ser la función $b \rightarrow c$. La composición de las dos funciones es una función que es del tipo $a \rightarrow c$. Se puede ver esto también en el tipo de `después`:

```
despues :: (b->c) -> (a->b) -> (a->c)
```

Dado que `->` asocia por la derecha, el tercer par de paréntesis no es necesario:

```
despues :: (b->c) -> (a->b) -> a -> c
```

La función `despues` parece una función de tres parámetros; por el mecanismo de currificación esto es lo mismo que una función de dos parámetros que devuelve una función (y también es igual a una función de un parámetro que devuelve una función de un parámetro que devuelve una función). Efectivamente, se puede definir `después` como función de tres parámetros:

```
despues g f x = g (f x)
```

Por tanto, no se tiene que crear una función `h` con una construcción de `where` (pero se puede). En la definición de `impar` (arriba), `despues` es instanciada parcialmente con `not` y `par`. No se ha dado el tercer parámetro, esto pasa si se llama a la función `impar`.

La utilidad del operador `despues` parece reducida porque funciones como `impar` pueden ser definidas también por

```
impar x = not (par x)
```

Sin embargo, una composición de dos funciones puede ser un parámetro de otra función de orden superior. En este caso, es fácil si esta composición no necesita un nombre. La siguiente expresión resulta en una lista de números impares entre 1 y 100:

```
? filter (not 'despues' par) [1..100]
```

En el preludio está definido el operador 'composición de funciones'. Se lo anota como punto (porque el símbolo \circ no está en el teclado). Por tanto, se puede escribir:

```
? filter (not.par) [1..100]
```

Este operador es muy útil si se componen muchas funciones. Por tanto, se puede programar completamente en el nivel de funciones (ver también el título de este texto). Ya no se ven cosas como números y listas. ¿No se lee mejor $f=g.h.i.j.k$ que $f\ x=g(h(i(j(k\ x))))$?

2.4 Funciones numéricas

2.4.1 Calcular con enteros

Con una división de enteros (`Int`), se trunca a la parte entera: $10/3$ es 3. Sin embargo, no siempre es necesario usar números del tipo `Float` con divisiones. Por el contrario, muchas veces el *resto* de la división es más interesante que la fracción decimal. El resto de la división $10/3$ es 1.

La función estándar `rem` (*remainder*) calcula el resto de una división:

```
? 345 'rem' 12
9
```

Calcular el resto es útil por ejemplo en los siguientes casos:

- Calcular horas. Si son las 9 ahora, entonces 33 horas más tarde serían las $(9+33)\text{'rem' }24 = 20$.
- Calcular días. Codifica los días con 0=domingo, 1=lunes, ..., 6=sábado. Si es el día 3 (miercoles) ahora, entonces dentro de 40 días sería $(3+40)\text{'rem' }7 = 1$ (lunes).
- Determinar si un número es divisible. Un número es divisible por n si el resto de la división por n es igual a cero.
- Determinar las cifras de un número. La última cifra de un número x es $x\text{'rem' }10$. La penúltima cifra es $(x/10)\text{'rem' }10$. La antepenúltima cifra es $(x/100)\text{'rem' }10$, etc.

Como ejemplo bastante profundo, siguen dos aplicaciones de cálculo con enteros: calcular una lista de números primos, y calcular el día de la semana, dada la fecha.

Calcular una lista de números primos

Un número es divisible por otro número, si el resto de la división es igual a cero. La función `divisible` determina si un número es divisible por otro número:

```
divisible    :: Int -> Int -> Bool
divisible t n = t 'rem' n == 0
```

Los divisores de un número son los números por los que un número es divisible. La función `divisores`, determina la lista de divisores de un número:

```
divisores    :: Int -> [Int]
divisores x = filter (divisible x) [1..x]
```

La función `divisible` es instanciada parcialmente con `x`; los elementos de `[1..x]` que son divisores de `x` son filtrados por la llamada de `filter`.

Un número es primo si tiene exactamente dos divisores: 1 y él mismo. La función `primo` ve si la lista de divisores consiste en estos dos números:

```
primo  :: Int -> Bool
primo x = divisores x == [1,x]
```

La función `numerosPrimos` calcula todos los números primos hasta un límite:

```
numerosPrimos  :: Int -> [Int]
numerosPrimos x = filter primo [1..x]
```

Es claro que este método no es el más eficiente para calcular números primos, pero es muy fácil de programar: las funciones son una traducción directa de las definiciones matemáticas.

Calcular el día de la semana

¿Qué día de la semana es la última Nochevieja de este siglo?

```
? dia 31 12 1999
viernes
```

Si se sabe el número del día (según el código de arriba: 0 = domingo etc.), entonces se puede escribir la función `dia` fácilmente:

```
dia d m a = diaSemana (numeroDia d m a)
diaSemana 0 = "domingo"
diaSemana 1 = "lunes"
diaSemana 2 = "martes"
diaSemana 3 = "miercoles"
diaSemana 4 = "jueves"
diaSemana 5 = "viernes"
diaSemana 6 = "sabado"
```

La función `diaSemana` usa siete patrones para seleccionar el texto adecuado (una palabra entre comillas (") es un texto; ver el párrafo 3.2.1).

La función `numeroDia` selecciona un domingo en un pasado lejano y suma:

- el número de años pasados por 365
- una corrección para los años bisiestos pasados;
- los tamaños de los meses pasados del año actual;
- el número de días pasados del mes actual.

El resultado es un número bastante grande. De este número, se calcula el resto con división por 7; este es el número del día que se necesita.

Desde la reforma del calendario del papa Gregorius en 1752, existe la siguiente regla para los años bisiestos (años con 366 días):

- un año divisible por 4 es un año bisiesto (por ejemplo 1972);
- excepción: si es divisible por 100, entonces no es un año bisiesto
- excepción de la excepción: si es divisible por 400, entonces es un año bisiesto (por ejemplo 2000)

Como punto cero de los números de días elegimos el año ficticio 0 (para esto, extrapolamos hacia atrás hasta este año). En este caso, la función `numeroDia` es más simple: el primero de enero en el año (ficticio) 0 sería un domingo.

```
numeroDia d m a = ( (a-1)*365
```

```

+ (a-1)/4
- (a-1)/100
+ (a-1)/400
+ sum (take (m-1) (meses a))
+ d
) 'rem' 7

```

La llamada `take n xs` devuelve los primeros `n` elementos de la lista `xs`. La función `take` puede ser definida por:

```

take 0 xs = []
take (n+1) (x:xs) = x : take n xs

```

La función `meses` tiene que retornar la cantidad de días de los meses en un año especificado:

```

meses a = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  where feb | bisiestro a = 29
           | otherwise = 28

```

Para definir la función `bisiestro`, usamos las reglas de arriba:

```

bisiestro a = divisible a 4 && (not(divisible a 100) || divisible a 400)

```

Otro método para definir esta función es:

```

bisiestro a | divisible a 100 = divisible a 400
           | otherwise = divisible a 4

```

Ahora tenemos todo lo necesario para la definición de la función `dia`. Tal vez sea aconsejable decir que la función `dia` puede ser usada solamente para años después de la reforma de calendario del papa Gregorius:

```

dia d m a | a>1752 = semanaDia (numeroDia d m a)

```

Una llamada de `dia` con un año anterior a 1753, retorna un mensaje de error automáticamente.

(Fin de los ejemplos.)

Hemos usado dos diferentes estrategias en el desarrollo de los dos ejemplos. En el segundo ejemplo, empezamos con la función pedida `dia`. Se necesitaron las funciones de ayuda `semanaDia` y `numeroDia`. Para `numeroDia`, necesitamos una función `meses`, y para `meses` una función `bisiestro`. Este enfoque se llama *top-down*: empezar con lo más importante (lo más global), y después poco a poco, realizar una descripción más detallada.

En el primer ejemplo, se usó el enfoque *bottom-up*: primero escribimos la función `divisible`, con ayuda de esta función escribimos la función `divisores`, con esa la función `primo`, y por último la función pedida `numerosPrimos`.

Para el resultado final, no importa que método se use (el intérprete acepta cualquier orden de las funciones). Pero, para programar, es útil saber que método se quiere usar. También es posible usar los dos (hasta que el 'top' encuentre el 'bottom').

2.4.2 Diferenciar numéricamente

En el cálculo con números del tipo `Float`, muchas veces no es posible obtener una respuesta exacta. Por ejemplo, se redondea el resultado de una división a un número fijo de decimales:

```

? 10.0/6.0
1.6666667

```

Para el cálculo de algunas operaciones, como `sqrt`, se usa también una aproximación. Por tanto, es aceptable que el resultado de una función sobre `Float` definida por el usuario, sea una aproximación del resultado exacto.

Un ejemplo es el cálculo de la función derivada. La definición matemática de la derivación f' de la función f es: El valor exacto de este límite no puede ser calculado por el ordenador. Una aproximación puede ser calculada mediante un valor de h suficientemente pequeño (tampoco demasiado pequeño, porque entonces la división dará errores de aproximación inaceptables).

La operación ‘derivación’ es una función de orden superior: la entrada es una función y el resultado es una función. La definición en Gofer puede ser:

```
diff    :: (Float->Float) -> (Float->Float)
diff f  = f'
  where f' x = (f (x+h) - f x) / h
        h    = 0.0001
```

Por el mecanismo de currificación puede omitirse el segundo par de paréntesis en el tipo, porque `->` asocia por la derecha.

```
diff :: (Float->Float) -> Float -> Float
```

Por tanto, se puede ver a la función `diff` como una función de dos parámetros: la función que se quiere diferenciar, y el punto en que se quiere calcular la función derivada. Con este enfoque la definición podía haber sido:

```
diff f x = (f (x+h) - f x) / h
  where h = 0.0001
```

Las dos definiciones son exactamente iguales. Tal vez, la segunda versión sea mejor, porque la definición es más clara (no es necesario dar primero un nombre a la función f' y después definirla). Por otro lado, en la primera definición está claro que se puede interpretar `diff` como transformación de una función.

Se puede usar la función `diff` fácilmente para instanciación parcial como en la definición:

```
derivacion_de_seno = diff (sin)
```

El valor de h está en las dos definiciones de `diff` en la parte de `where`. De esta manera, se puede cambiar el valor fácilmente (esto se puede también en la expresión misma, pero entonces, se tendría que hacer el cambio dos veces con el peligro de olvidar uno).

Más flexible es usar el valor de h como parámetro de `diff`:

```
flexDiff h f x = (f (x+h) - f x) / h
```

Por definir h como primer parámetro de `flexDiff`, se puede instanciar parcialmente la función para hacer diferentes versiones de `diff`:

```
burdoDiff = flexDiff 0.01
finoDiff  = flexDiff 0.0001
superDiff = flexDiff 0.000001
```

2.4.3 La raíz cuadrada

La función `sqrt` es primitiva en Gofer. Calcula la raíz cuadrada (*square root*) de un número. En esta sección mostramos un método para definir esta función, si no fuera primitiva. Este método muestra una

pág. 36

técnica que se usa mucho en el cálculo con números del tipo `Float`. En el párrafo 2.4.5, el método se generaliza para inversos de otras funciones, aparte de la `sqrt`. En esa sección se explica porqué el método que usamos aquí funciona.

Se tiene la siguiente propiedad de la raíz cuadrada de un número x :

si y es una aproximación de \sqrt{x}
entonces $\frac{1}{2}(y + \frac{x}{y})$ es una aproximación mejor.

Se puede usar esta propiedad para calcular la raíz de un número x : toma 1 como primera aproximación, y calcula cada vez mejores aproximaciones hasta que el resultado sea bastante aceptable. El valor de y es bastante aceptable como aproximación de \sqrt{x} , si y^2 es una aproximación bastante aceptable (¿definición recursiva?) de x .

Para el valor de $\sqrt{3}$ tenemos las siguientes aproximaciones y_0, y_1 , etc. como sigue:

```

y0 =                = 1
y1 = 0.5 * (y0 + 3/y0) = 2
y2 = 0.5 * (y1 + 3/y1) = 1.75
y3 = 0.5 * (y2 + 3/y2) = 1.732142857
y4 = 0.5 * (y3 + 3/y3) = 1.732050810
y5 = 0.5 * (y4 + 3/y4) = 1.732050807

```

El cuadrado de la última aproximación difiere solamente en 10^{-18} de 3.

Se puede usar la función `until` en el párrafo 2.3.2 para el proceso ‘mejorar el resultado hasta que sea bastante aceptable’: pág. 29

```

raiz x = until bastanteAccept mejorar 1.0
  where mejorar y = 0.5*(y+x/y)
        bastanteAccept y = y*y ~ = x

```

El operador `~ =` es el operador ‘casi igual a’, que se puede definir como:

```

infix 5 ~ =
a ~ = b = a-b<h && b-a<h
  where h = 0.000001

```

La función de orden superior `until` se aplica a las funciones `mejorar` y `bastanteAccept` y al valor de inicialización `1.0`. Aunque `mejorar` está al lado de `1.0`, no se la aplica directamente a `1.0`; en vez de eso, se usan los dos como parámetros de `until`.

Por el mecanismo de curriificación es como si los paréntesis estuvieran de la siguiente manera: `((until bastanteAccept) mejorar) 1.0`. Durante la ejecución de `until`, resulta que se aplica `mejorar` a `1.0`.

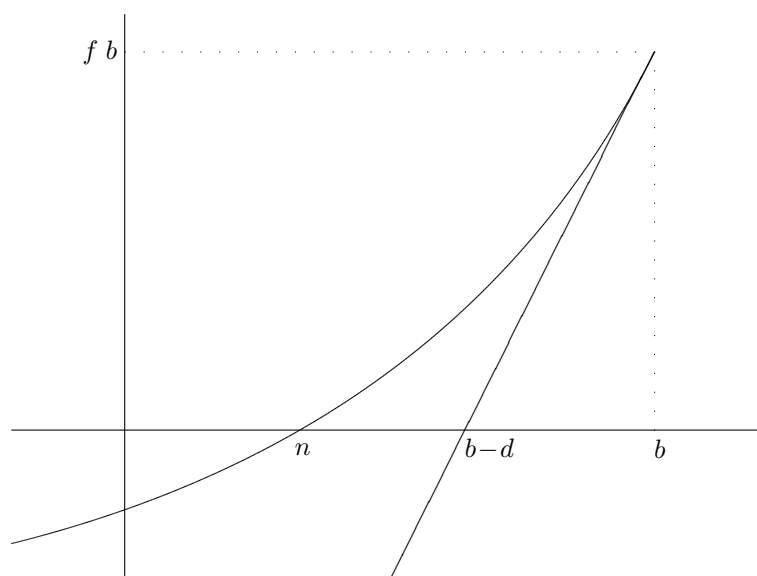
Otra cosa interesante es que las funciones `mejorar` y `bastanteAccept` usan su parámetro `y`, pero también el parámetro `x` de `raiz`. Para estas funciones, `x` funciona como constante. (Comparar con las definiciones de las constantes `d` y `n` en la definición de `formulaABC` en el párrafo 1.4.1).

pág. 11

2.4.4 Ceros de una función

Otro problema numérico que puede ser resuelto por iteración con la función `until`, es el cálculo de los ceros de una función.

Examinamos la función f en la que se quiere calcular el punto cero n . Supongamos que b es una aproximación del cero. Entonces el punto de intersección de la tangente a f en b con el eje x , es una aproximación mejor para el cero (ver figura).



El punto de intersección que buscamos está a una distancia d de la primera aproximación b . Se puede calcular el valor de d de la siguiente manera. La pendiente de la tangente a f en el punto b es igual a $f'(b)$. También es esta pendiente de la tangente igual a $f(b)/d$. Por tanto $d = f(b)/f'(b)$.

Con esto, tenemos una función de mejora de la aproximación: si b es una aproximación para el punto cero de f , entonces $b - f(b)/f'(b)$ es una mejor aproximación. Este método se llama ‘el método de Newton’. (Este método no siempre funciona: con funciones con extremos locales puede ser que no se encuentre una buena solución. No detallamos esto aquí.)

La función `puntoCero` es casi igual a `raiz`:

```
puntoCero f = until bastanteAccept mejorar 1.0
  where mejorar b = b - f b / diff f b
        bastanteAccept b = f b ~ 0.0
```

La aproximación será bastante aceptable si el resultado de la función f aplicada a la aproximación casi equivale a cero. Como primera aproximación se ha elegido 1.0, pero también podría haber sido 17.93. El punto tiene que estar en el dominio de f .

2.4.5 Inverso de una función

El cero de la función f con $f x = x^2 - a$ es \sqrt{a} . Por tanto, la raíz de a puede ser calculada buscando el cero de f . Ahora tenemos la función `puntoCero`, por tanto se puede escribir también la función `raiz` de la siguiente manera:

```
raiz a = puntoCero f
  where f x = x*x-a
```

La raíz cúbica también puede ser calculada con este método:

```
raiz3 a = puntoCero f
  where f x = x*x*x-a
```

En realidad, muchos inversos de funciones pueden ser calculados usando esta función en la definición de f , por ejemplo:

```
arcsin a = puntoCero f
```



```

    where f x = sin x - a
arccos a = puntoCero f
    where f x = cos x - a

```

Existe un patrón en estas definiciones. Ésa es una señal para definir una función de orden superior que describa la generalización del patrón (ver el párrafo 2.3.1, donde se ha definido `foldr` como generalización de `sum`, `product` y `and`). La función de orden superior es en este caso la función `inverso`, que tiene como parámetro una función `g` de la cual se quiere calcular el inverso:

pág. 29

```

inverso g a = puntoCero f
    where f x = g x - a

```

Si se ve el patrón, las funciones de orden superior no son más difíciles que las otras definiciones. Esas otras definiciones son casos especiales de la función de orden superior, se las puede escribir con instanciación parcial:

```

arcsin = inverso sin
arccos = inverso cos
ln      = inverso exp

```

Se puede usar la función `inverso` como función de dos parámetros (una función y un `Float`) y un resultado del tipo `Float`, o como función de un parámetro (una función) y una función como resultado. Esto nos indica que el tipo de `inverso` es:

```

inverso :: (Float->Float) -> Float -> Float

```

que puede ser escrito como:

```

inverso :: (Float->Float) -> (Float->Float)

```

porque `->` asocia por la derecha.

La función `raiz` en el párrafo 2.4.3 usa también el método de Newton. Esto resultaría, reemplazando en la definición de `raiz` de arriba:

pág. 34

```

raiz a = puntoCero f
    where f x = x*x-a

```

la llamada `puntoCero f` por la definición de ella:

```

raiz a = until bastanteAccept mejorar 1.0
    where mejorar b = b - f b / diff f b
          bastanteAccept b = f b ~= 0.0
          f x = x*x-a

```

En este caso especial no hace falta calcular `diff f` numéricamente: la derivación de la función `f` es (aquí) igual a `2*x`. Por tanto, se puede simplificar la fórmula para `mejorar b`:

$$\begin{aligned}
 & b - \frac{f\ b}{f'\ b} \\
 = & b - \frac{b^2 - a}{2b} \\
 = & b - \frac{b^2}{2b} + \frac{a}{2b} \\
 = & \frac{b}{2} + \frac{a/b}{2} \\
 = & 0.5 * (b + a/b)
 \end{aligned}$$

Esta es exactamente la fórmula de mejora que se ha usado en el párrafo 2.4.3.

pág. 34

Ejercicios

Ejercicio 2.1

pág. 32

En el párrafo 2.4.1 se describe una función que calcula en base a una fecha que día es (fue) esa fecha. Escriba con uso (si quiere) de las funciones descritas en este párrafo una función *fecha* que, dado un número de un día y el número de un año, calcule qué fecha (mes y número del día en el mes) y día (miércoles, jueves, etc.) representa. Por ejemplo:

```
? fecha 1 1993  
mes: enero  
dia: viernes 1
```

```
? fecha 300 1993  
mes: octubre  
dia: miercoles 27
```

Capítulo 3

Estructuras de datos

3.1 Listas

3.1.1 Estructura de una lista

Las listas se usan para agrupar varios elementos. Estos elementos tienen que ser *del mismo tipo*. Para cada tipo existe un tipo 'lista de ese tipo'. Por tanto, existen listas de enteros, listas de floats y listas de funciones de entero a entero. Pero también se pueden agrupar en una lista listas del mismo tipo. El resultado: listas de listas de enteros, listas de listas de listas de booleanos etc.

El tipo de una lista se indica escribiendo el tipo de sus elementos entre corchetes. Por tanto, los tipos de las listas arriba mencionadas se representan por: `[Int]`, `[Float]`, `[Int->Float]`, `[[Int]]` y `[[[Bool]]]`.

Existen diferentes maneras de construir listas: enumeración, construcción con `:`, e intervalos numéricos.

Enumeración

La enumeración de los elementos de una lista es a menudo la manera más fácil para construir una lista. Los elementos tienen que ser del mismo tipo. Algunos ejemplos de enumeraciones de listas con sus tipos son:

```
[1, 2, 3]           :: [Int]
[1, 3, 7, 2, 8]    :: [Int]
[True, False, True] :: [Bool]
[sin, cos, tan]    :: [Float->Float]
[ [1,2,3], [1,2] ] :: [[Int]]
```

Para el tipo de la lista, no importa cuantos elementos contenga esta. Una lista con tres enteros y otra con dos enteros, tienen ambas el tipo `[Int]`. Por eso, las listas `[1,2,3]` y `[1,2]` pueden ser elementos de una lista de listas (véase el quinto ejemplo).

No hace falta que los elementos de una lista sean constantes; pueden ser expresiones:

```
[ 1+2, 3*x, length [1,2] ] :: [Int]
[ 3<4, a==5, p && q ]      :: [Bool]
[ diff sin, inverse cos ]  :: [Float->Float]
```

Las expresiones en cada lista tienen que ser del mismo tipo.

No hay restricciones en el número de los elementos de una lista. Es posible que una lista solamente contenga un elemento:

```
[True]      :: [Bool]
[[1,2,3]]   :: [[Int]]
```

Una lista con un elemento se llama en inglés *singleton list*. La lista `[[1,2,3]]` es un *singleton-list*: es una lista de listas que contiene un elemento (la lista `[1,2,3]`).

Cuidado con la diferencia entre una *expresión* y un *tipo*. Un tipo entre corchetes es un tipo (por ejemplo `[Bool]` o `[[Int]]`). Una expresión entre corchetes es una expresión (un *singleton list*, por ejemplo `[True]` o `[3]`).

El número de elementos de una lista también puede ser cero. Una lista con cero elementos se llama la *lista vacía*. La lista vacía es de un tipo polimórfico: es una ‘lista de no importa que’. Dentro un tipo polimórfico existen variables de tipo que representan cualquier tipo (ver el párrafo 1.5.3), por tanto, el tipo de la lista vacía es `[a]`:

```
[] :: [a]
```

La lista vacía puede ser usada en cualquier lugar de una expresión en el que se necesite una lista. El contexto determina el tipo:

```
sum []           [] es una lista vacía de números
and []          [] es una lista vacía de booleanos
[ [], [1,2], [3] ] [] es una lista vacía de números
[ [1<2,True], [] ] [] es una lista vacía de booleanos
[ [[1]], [] ]   [] es una lista vacía de listas de números
length []       [] es una lista vacía (no importa de que)
```

Construcción con :

Otra manera para construir listas es utilizando del operador `:`. Este operador añade un elemento al principio (a la izquierda) de una lista y construye de esta manera una lista más larga.

```
(:) :: a -> [a] -> [a]
```

Si por ejemplo, `xs` es la lista `[3,4,5]`, entonces `1:xs` es la lista `[1,3,4,5]`. Usando la lista vacía y el operador `:`, se puede construir cualquier lista. Por ejemplo `1:(2:(3:[]))` es la lista `[1,2,3]`. El operador `:` asocia por la derecha, por ello se puede escribir simplemente `1:2:3:[]`.

En realidad, esta forma es la única manera ‘real’ de construir listas. Una enumeración de una lista es muchas veces más clara, pero tiene el mismo sentido que la expresión con el operador `:`. Por eso, la enumeración cuesta tiempo:

```
? [1,2,3]
[1, 2, 3]
(7 reductions, 29 cells)
```

Cada llamada de `:` (que no se ve, sino que está) cuesta dos reducciones.

Intervalos numéricos

La tercera manera de construir una lista es con la notación de intervalos: dos expresiones numéricas separadas por dos puntos y rodeadas de corchetes:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

(Aunque se puede usar el punto como símbolo en operadores, `..` no es un operador. Es una de las combinaciones de símbolos que fue reservada para usos especiales en el párrafo 1.3.2.)

El valor de la expresión `[x..y]` se calcula por una llamada de `enumFromTo x y`. La definición de la función `enumFromTo` es:

```
enumFromTo x y | y < x      = []
               | otherwise = x : enumFromTo (x+1) y
```

Si `y` es más pequeño que `x`, entonces el resultado es la lista vacía; de otro modo `x` es el primer elemento, el siguiente elemento es uno mayor (a menos que se haya pasado el límite `y`), y así sucesivamente.

La notación para intervalos numéricos no es más que una simplificación que hace el uso del lenguaje un poco más fácil; sin esta construcción no se perdería nada, ya que podemos usar la función `enumFromTo`.

3.1.2 Funciones sobre listas

Muchas veces las funciones se definen por medio de *patrones*. La función se define para el caso de la lista vacía y para una lista que tiene la forma `x:xs`, puesto que cada lista o es vacía, o tiene un primer elemento `x`, que está delante de una lista `xs` (`xs` puede ser vacía).

Ya hemos visto algunas definiciones de funciones sobre listas: `head` y `tail` en el párrafo 1.4.3, `suma` y `length` en el párrafo 1.4.4, y `map`, `filter` y `foldr` en el párrafo 2.3.1. Aunque estas funciones están definidas en el preludio, es importante que se vean sus definiciones. Primero, porque son buenos ejemplos de definiciones de funciones sobre listas. Segundo, porque la definición da muchas veces una descripción más clara de lo que hace una función estándar.

En este párrafo siguen más definiciones de funciones sobre listas. Muchas de estas funciones son recursivas: con el patrón `x:xs`, se llaman a si mismas con el parámetro `xs` (que es menor que `x:xs`).

Comparar y ordenar listas

Se pueden comparar y ordenar listas (con operadores como `==` y `<`) con la condición de que se puedan comparar y ordenar sus elementos. En otras palabras el tipo `[t]` pertenece a la clase de los tipos comparables `Eq` si `t` está en esta clase y pertenece a la clase de los tipos ordenables `Ord` si `t` está en esta clase.

Dos listas son iguales si tienen exactamente los mismos elementos en el mismo orden. El operador de igualdad sobre listas se define de la siguiente manera:

```
[] == []      = True
[] == (y:ys)  = False
(x:xs) == []  = False
(x:xs) == (y:ys) = x==y && xs==ys
```

En esta definición, los dos parámetros pueden ser la lista vacía o no; para cada una de las cuatro combinaciones existe una definición. En el cuarto caso se llama a la igualdad para los primeros elementos (`x==y`), y se llama recursivamente al operador sobre los restos de las listas (`xs==ys`).

Si se pueden ordenar los elementos de una lista (con `<`, `≤` etc.), también se pueden ordenar las listas. Esto se realiza según *el orden lexicográfico*. El primer elemento de las listas decide el orden de las listas, a no ser que sean iguales. En ese caso, decide el segundo elemento, a no ser que sean iguales, *etcetera*. Por tanto, las siguientes expresiones son verdaderas: `[2,3]<[3,1]` y `[2,1]<[2,2]`. Si una de las listas es el comienzo de la otra, la más corta es la menor, por ejemplo `[2,3]<[2,3,4]`. La palabra ‘etcetera’ en la anterior descripción, indica que necesitamos recursión en la definición:

```
[] <= ys      = True
(x:xs) <= []  = False
(x:xs) <= (y:ys) = x<=y || (x==y && xs<=ys)
```

Con los operadores == y <=, se pueden definir además otros operadores de comparación:

```
xs /= ys = not (xs==ys)
xs >= ys = ys<=xs
xs < ys = xs<=ys && xs/=ys
xs > ys = ys< xs
```

Estos operadores también podrían haberse definido recursivamente.

Concatenar listas

Se pueden unir dos listas del mismo tipo en una sola lista con el operador ++. Esta operación se llama *concatenación*. Por ejemplo: [1,2,3]++[4,5] es igual a la lista [1,2,3,4,5]. Concatenación de una lista con la lista vacía (ya sea por delante o por detrás) no cambia esta lista: [1,2]++[] es igual a [1,2].

El operador ++ es una función estándar, pero ésta se puede definir en Gofer (esto se realiza en el preludio). Por tanto, no es un operador primitivo como :. La definición es:

```
(++)      :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs++ys)
```

Se usa el primer parámetro para el análisis por patrones. En el caso de que el primer parámetro no sea la lista vacía, se realiza una llamada recursiva con una lista más corta (xs) como parámetro.

Existe otra función que concatena listas. Esta función, `concat`, se aplica a una lista de listas. Todas las listas en la lista de listas se concatenan a una lista más grande. Por ejemplo:

```
? concat [ [1,2,3], [4,5], [], [6] ]
[1, 2, 3, 4, 5, 6]
```

La definición de `concat` es la siguiente:

```
concat      :: [[a]] -> [a]
concat []   = []
concat (xs:xss) = xs ++ concat xss
```

El primer patrón, [], es la lista vacía; en realidad es una lista vacía de listas. El resultado es en este caso la lista vacía. La lista no está vacía en el segundo caso. Existe una lista (xs) en frente de la lista de listas. El resto de la lista de listas es xss. Se une todas las listas en xss con la llamada recursiva de `concat`. Con el resultado se une la primera lista xs.

Atención a la diferencia entre ++ y `concat`: el operador ++ opera sobre *dos* listas, la función `concat` opera sobre *una lista de listas*. Vulgarmente se llaman ambas ‘concatenación’. (Compárese la situación con el operador &&, que comprueba si dos booleanos son True, y la función `and` que comprueba si todos los elementos de una lista de booleanos son True.

Seleccionar partes de una lista

En el preludio se definen una serie de funciones que se pueden usar para seleccionar partes de una lista. Para algunas funciones el resultado es una lista más corta, con otras el resultado es un elemento.

Ya que una lista se construye con una cabeza y una cola, es fácil recuperar la cabeza y la cola de una lista:

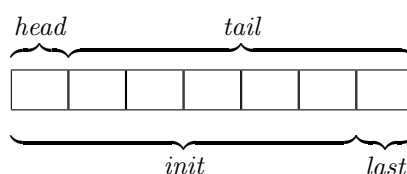
```
head      :: [a] -> a
head (x:xs) = x
tail     :: [a] -> [a]
tail (x:xs) = xs
```

Estas funciones realizan un análisis por patrones al parámetro, pero no existe una definición para el patrón `[]`. Si se llama a estas funciones con una lista vacía, el resultado es un mensaje de error.

No es tan fácil escribir una función que seleccione el *último* elemento de una lista. Para eso se necesita recursión:

```
last      :: [a] -> a
last (x:[]) = x
last (x:xs) = last xs
```

Esta función tampoco está definida para la lista vacía. De la misma forma que la función `tail` se corresponde con la función `head`, `init` se corresponde con la función `last`. Una representación esquemática de estas cuatro funciones es:

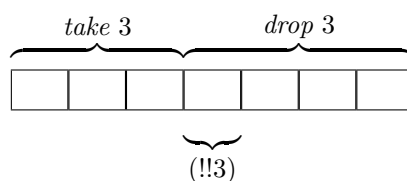


La función `init` selecciona todo *excepto* el último elemento de la lista. También para esto necesitamos recursión:

```
init      :: [a] -> [a]
init (x:[]) = []
init (x:xs) = x: init xs
```

El patrón `x:[]` puede ser escrito como `[x]`.

En el párrafo 2.4.1 se presentó la función `take`. Además de una lista, `take` tiene un entero `n` como parámetro. Este entero dice cuántos de los primeros elementos de la lista están en el resultado. El opuesto de `take` es la función `drop`, que elimina los primeros `n` elementos de la lista. Por último existe un operador `!!`, que selecciona un elemento determinado de la lista. Esquemáticamente:



Las definiciones de estas funciones son:

```
take, drop :: Int -> [a] -> [a]
take 0 xs   = []
take n []   = []
take (n+1) (x:xs) = x : take n xs
drop 0 xs   = xs
drop n []   = []
drop (n+1) (x:xs) = drop n xs
```

Si la lista es más pequeña que el valor especificado por `take`, se selecciona la lista completa. En el caso de `drop`, se elimina la lista completa (el resultado es la lista vacía). La razón es la segunda regla en las definiciones: esta regla dice que, si se aplica la función a la lista vacía, el resultado es siempre la lista vacía, a pesar del parámetro entero. Sin estas reglas, `take` y `drop` habrían sido indefinidas para listas demasiado cortas.

El operador `!!` selecciona un elemento de una lista. El primer elemento tiene índice cero, por tanto `xs!!3` devuelve el *cuarto* elemento de la lista `xs`. No se puede aplicar el operador sobre listas demasiado cortas, porque en este caso no se puede devolver ningún resultado razonable. La definición del operador `!!` es:

```

infixl 9 !!
(!!)      :: [a] -> Int -> a
(x:xs) !! 0      = x
(x:xs) !! (n+1) = xs !! n

```

Este operador busca en la lista desde el principio hasta el elemento que se pida. Por tanto, si el parámetro es muy grande se gasta mucho tiempo. Lo mejor es utilizar esta función solamente en casos estrictamente necesarios. El operador es útil si se quiere seleccionar un elemento de una lista. Por ejemplo, la función `diaSemana` podría definirse como:

```

diaSemana d = [ "domingo", "lunes", "martes", "miercoles",
                "jueves", "viernes", "sabado" ] !! d

```

Si se tiene que seleccionar todos los elementos de una lista uno por uno, entonces es mejor usar las funciones `map` o `foldr`.

Inversión del orden de una lista

La función `reverse` del preludio invierte los elementos de la lista. La definición recursiva es simple. El inverso de una lista vacía es también una lista vacía. Para una lista no vacía, se tiene que invertir el orden de la cola y se tiene que situar el primer elemento al final de este resultado. Por tanto, una definición podría ser:

```

reverse []      = []
reverse (x:xs) = reverse xs ++ [x]

```

Propiedades de listas

Una propiedad importante de una lista es su tamaño. Se puede calcular el tamaño con la función `length`. La definición de esta función, que está en el preludio, es:

```

length      :: [a] -> Int
length []   = 0
length (x:xs) = 1 + length xs

```

En el preludio está también una función `elem` que comprueba si un elemento determinado está en la lista. Se puede definir esta función como sigue:

```

elem      :: a -> [a] -> Bool
elem e xs = or (map (==e) xs)

```

La función compara todos los elementos de `xs` con `e` (instanciación parcial del operador `==`). El resultado es una lista booleana, en la cual `or` controla si existen uno o más elementos con el valor `True`. También se puede escribir la definición con el operador de composición de funciones:

```

elem e = or . (map (==e))

```

La función `notElem` comprueba si un determinado elemento no está en la lista:

```

notElem e xs = not (elem e xs)

```

Otra definición para ella es:

```

notElem e = and . (map (/=e))

```


3.1.3 Funciones de orden superior sobre listas

Las funciones pueden ser más flexibles utilizando funciones como parámetros. Muchas funciones sobre listas tienen una función como parámetro. A estas se les llama funciones de orden superior.

map, filter y foldr

Anteriormente hemos visto las funciones `map`, `filter` y `foldr`. Estas funciones hacen algo con todos los elementos de una lista, dependiendo del parámetro función. La función `map` aplica su parámetro función a todos los elementos de una lista:

```

xs = [ 1 , 2 , 3 , 4 , 5 ]
      ↓ ↓ ↓ ↓ ↓
map cuadrado xs = [ 1 , 4 , 9 , 16 , 25 ]

```

La función `filter` elimina los elementos de una lista que no cumplan con una condición (representada por una función booleana):

```

xs = [ 1 , 2 , 3 , 4 , 5 ]
      × ↓ × ↓ ×
filter even xs = [ 2 , 4 ]

```

La función `foldr` inserta un operador entre todos los elementos de una lista, empezando a la derecha con un valor dado:

```

xs = [ 1 , 2 , 3 , 4 , 5 ]
      ↓ ↓ ↓ ↓ ↓
foldr (+) 0 xs = (1 + (2 + (3 + (4 + (5 + 0))))))

```

Las tres funciones estándar están definidas recursivamente en el prelude. Se discutieron en el párrafo 2.3.1. pág. 29

```

map          :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = x : map f xs

filter       :: (a->Bool) -> [a] -> [a]
filter p []  = []
filter p (x:xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs

foldr        :: (a->b->b) -> b -> [a] -> b
foldr op e [] = e
foldr op e (x:xs) = x 'op' foldr op e xs

```

Muchas veces, la recursión puede estar escondida al usar estas funciones estándar. Las otras funciones parecen entonces más claras. En la definición de la función `or`, que comprueba si en una lista booleana si existe por lo menos un valor `True`, se usa este principio:

```
or = foldr (||) False
```

También es posible definir esta función directamente con recursión, sin usar la función `foldr`:

```

or []       = False
or (x:xs)   = x || or xs

```

Muchas funciones pueden ser escritas como combinación de una llamada de `foldr` y una llamada de `map`. La función `elem` del anterior párrafo es un ejemplo de eso:

```
elem e = foldr (||) False . map (==e)
```

Pero también esta función puede ser definida directamente, sin usar funciones estándar. La recursión es necesaria en este caso:

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

takeWhile y dropWhile

Una variante de la función `filter` es la función `takeWhile`. Esta función tiene, como la función `filter`, un predicado (una función con un resultado del tipo `Bool`) y una lista como parámetro. La diferencia es que `filter` siempre inspecciona todos los elementos de la lista. La función `takeWhile` empieza al principio de la lista y para de buscar cuando encuentra un elemento que no satisface el predicado. Por ejemplo: `takeWhile even [2,4,6,7,8,9]` da el resultado `[2,4,6]`. El elemento 8 no está en el resultado, con la función `filter` habría estado en él. El elemento 7 hace que la función `takeWhile` detenga la búsqueda. La definición en el prelude es:

```
takeWhile :: (a->Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

Compare esta definición con la de `filter`.

De igual forma la función `drop` corresponde a `take`, la función `dropWhile` corresponde a `takeWhile`. Esta elimina la parte inicial de una lista que no cumple con una condición. Por ejemplo: `dropWhile even [2,4,6,7,8,9]` es `[7,8,9]`. Una definición es:

```
dropWhile :: (a->Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = x:xs
```

foldl

La función `foldr` inserta un operador entre todos los elementos de una lista empezando a la derecha de la lista. La función `foldl` hace lo mismo, pero empieza a la izquierda. Igual que `foldr`, `foldl` tiene un parámetro extra que indica cual es el resultado para la lista vacía.

Un ejemplo del uso de `foldl` aplicada a una lista con cinco elementos es el siguiente:

```
xs = [ 1 , 2 , 3 , 4 , 5 ]
      ↓   ↓   ↓   ↓   ↓
foldl (+) 0 xs = (((((0 + 1) + 2) + 3) + 4) + 5)
```

Para definir esta función es útil ver primero dos ejemplos:

```
foldl (⊕) a [x,y,z] = ((a ⊕ x) ⊕ y) ⊕ z
foldl (⊕) b [ y,z] = ( b ⊕ y) ⊕ z
```

Por tanto, la llamada de `foldl` a la lista `x:xs` (con `xs=[y,z]` en el ejemplo) es igual a `foldl xs a` a condición de que se use en la llamada recursiva como valor inicial `a ⊕ x` en vez de `a`. Con esta observación se puede escribir una definición:

```

foldl op e []      = e
foldl op e (x:xs) = foldl op (e'op'x) xs

```

Para operadores asociativos como `+` no importa mucho si se usa `foldr` o `foldl`. Para operadores no asociativos como `-`, el resultado con `foldr` es diferente en general al obtenido con `foldl`.

3.1.4 Ordenamiento de listas

Todas las funciones sobre listas que hemos visto son todavía bastante simples. Cada elemento de la lista se visita recursivamente una vez para determinar el resultado.

Una función que no se puede escribir de esta manera es el ordenamiento de una lista en orden creciente. Se tiene que cambiar la posición de muchos de los elementos de la lista.

A pesar de ello no es difícil escribir una función de ordenamiento. Se pueden usar las funciones estándar. Existen diferentes métodos para resolver el problema de ordenamiento. En otras palabras: existen diferentes *algoritmos*. Discutiremos dos algoritmos. En los dos es necesario que se puedan ordenar los elementos de la lista. Por tanto, es posible ordenar una lista de enteros, o una lista de lista de enteros, pero no es posible ordenar una lista de funciones. Este hecho está expresado en el tipo de la función `ordenar`:

```
ordenar :: Ord a => [a] -> [a]
```

`Ordenar` opera sobre listas de cualquier tipo `a`, a condición de que el tipo `a` esté en la clase de los tipos cuyos elementos se puede ordenar (`Ord`).

Ordenamiento por inserción

Supongamos que tenemos una lista ordenada. En este caso podemos insertar un nuevo elemento en el lugar correspondiente con la siguiente función:

```

insert      :: Ord a => a -> [a] -> [a]
insert e []      = [e]
insert e (x:xs)
  | e<=x      = e : x : xs
  | otherwise  = x : insert e xs

```

Si inicialmente tenemos la lista vacía, el nuevo elemento es el único. Si la lista no está inicialmente vacía, y el elemento `x` es la cabeza de la lista, lo que se tiene que hacer depende del valor de `e`. Si `e` es menor que `x`, entonces `e` es la nueva cabeza de la lista. Si no, `x` sigue como cabeza y se tiene que insertar `e` en la cola de la lista. Un ejemplo del uso de `insert`:

```

? insert 5 [2,4,6,8,10]
[2, 4, 5, 6, 8, 10]

```

Para el uso de `insert` es absolutamente necesario que el parámetro lista esté ordenado. Solamente en este caso el resultado también estará ordenado.

La función `insert` también se puede ser usar para ordenar una lista que no está ordenada. Supongamos que se quiere ordenar la lista `[a,b,c,d]`. Por tanto, se puede empezar con una lista vacía e insertar primero el último elemento `d` en ella. El resultado es una lista ordenada en la que se puede insertar `c`. El resultado sigue ordenado, igualmente después de que se haya insertado `b`. Finalmente, se puede insertar el elemento `a` y el resultado es una versión ordenada de `[a,b,c,d]`. La expresión que se calcula es:

```
a 'insert' (b 'insert' (c 'insert' (d 'insert' [])))
```

La estructura de esta expresión es exactamente la de `foldr`, con `insert` como operador y `[]` como valor inicial. Por tanto, un algoritmo de ordenamiento puede ser:

```
isort = foldr insert []
```

con la función `insert` como fue definida anteriormente. Este algoritmo se llama en inglés *insertion sort*.

Ordenamiento por fusión

Otro algoritmo para ordenar usa la posibilidad de fusionar dos listas ordenadas en una lista ordenada. Para eso sirve la función `merge`:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

Si una de las listas es la lista vacía, entonces la otra lista es el resultado. Si ninguna es la lista vacía, entonces el más pequeño de los dos primeros elementos es la cabeza del resultado, y los elementos que restan son fusionados por una llamada recursiva de `merge`.

Al igual que `insert`, `merge` espera que los parámetros estén ordenados. Solamente en este caso, el resultado también está ordenado.

En base a esta función `merge`, se puede escribir un algoritmo de ordenamiento. Este algoritmo usa la propiedad de que la lista vacía y las listas con solamente un elemento siempre están ordenadas. Una lista más grande puede ser dividida en dos partes con (casi) el mismo tamaño. Las dos partes pueden ser ordenadas por una llamada recursiva del algoritmo de ordenamiento. Finalmente, los dos resultados ordenados se pueden fusionar con

```
msort xs
  | medio <= 1 = xs
  | otherwise = merge (msort ys) (msort zs)
where ys = take medio xs
      zs = drop medio xs
      medio = tamaño / 2
      tamaño = length xs
```

Este algoritmo se llama *merge sort*. En el preludio están definidas las funciones `insert` y `merge`, y una función `sort` que funciona como `isort`.

3.2 Listas especiales

3.2.1 Cadenas

pág. 32

En un ejemplo en el párrafo 2.4.1 se utilizaron textos como valores, por ejemplo, "lunes". Tal valor se llama *cadena*. Una cadena es una lista, cuyos elementos son caracteres.

Por tanto, se pueden usar todas las funciones sobre listas para cadenas. Por ejemplo, la expresión "mar"+"tes" devuelve la cadena "martes", y el resultado de la expresión `tail (take 3 "gofer")` es la cadena "gof".

Las cadenas se anotan entre comillas. Las comillas indican que el texto es el valor de una cadena y no el nombre de una función. Por tanto, "until" es una cadena de cinco elementos, `until` es el nombre de una función. Por eso se tiene que escribir las cadenas siempre entre comillas. Solamente el intérprete no las usa en el resultado de una expresión:

```
? "mar"+"tes"
martes
```

Los elementos de una cadena son del tipo `Char`. Esto es una abreviatura de la palabra *character*. No solamente las letras son caracteres, también las cifras y los signos de puntuación. El tipo `Char` es uno de los tipos base de Gofer (los otros tres son `Int`, `Float` y `Bool`).

Los valores del tipo `Char` son anotados por un símbolo entre *apóstrofes* (*quotes*, por ejemplo `'B'` o `'*'`). Cuidado con la diferencia con las comillas inversas, que se usan para convertir una función en un operador. Las siguientes expresiones tienen sentidos muy diferentes:

```
"f"   una lista de un elemento;
'f'   un carácter;
'f'   una función f, usada como operador.
```

La notación de una cadena con comillas no es más que una abreviatura por una secuencia de caracteres. La cadena `"hola"` significa lo mismo que la lista `['h','o','l','a']`, o `'h':'o':'l':'a':[]`.

Ejemplos que muestran este principio son las expresiones `hd "aap"` (que devuelve `'a'`) y `takeWhile (=='s') "sssttt"` (que devuelve la cadena `"sss"`).

3.2.2 Caracteres

Los valores del tipo `Char` pueden ser letras, cifras y signos de puntuación. Es importante usar comillas, porque normalmente estos signos significan algo diferente:

expresión	tipo	significado
<code>'x'</code>	<code>Char</code>	la letra <code>'x'</code>
<code>x</code>	<code>...</code>	el nombre de un parámetro (por ejemplo)
<code>'3'</code>	<code>Char</code>	la cifra <code>'3'</code>
<code>3</code>	<code>Int</code>	el número 3
<code>','</code>	<code>Char</code>	el signo de puntuación punto
<code>.</code>	<code>(b->c)->(a->b)->a->c</code>	el operador composición de funciones

Existen 128 (en algunos ordenadores 256) diferentes valores del tipo `Char`:

- 52 letras
- 10 cifras
- 32 signos de puntuación y el signo de espacio
- 33 símbolos especiales
- (128 signos extras, como letras con acentos etc.)

Existe un signo de puntuación que da problemas en una cadena: la comilla. Con una comilla en una cadena termina la cadena. Si se quiere representar una comilla en una cadena, se tiene que escribir el símbolo `\` delante de la comilla. Por ejemplo:

```
"El dijo \"hola\" y siguió caminando"
```

Esta solución da un nuevo problema, porque ahora no podemos incluir el símbolo `\` mismo en una cadena. Si se quiere incluir este símbolo en una cadena, se tiene que duplicar:

```
"el simbolo \\ se llama backslash"
```

Tal símbolo doble representa un carácter. Por tanto, la longitud de la cadena `"\\\"\\\"\\\""` es 4. También se puede escribir estos símbolos entre comillas simples, como en los siguientes ejemplos:

```
punto doble   = ':'
comilla       = '\"'
backslash     = '\\\'
apostrofo    = '\'
```

Los 33 símbolos especiales se usan para influir la tipografía del texto. Los más importantes son el ‘retorno de carro’ (línea nueva) y el ‘tabulador’. También estos caracteres se pueden ser representar con el uso del backslash: ‘\n’ es el carácter retorno de carro, y ‘\t’ es el carácter tabulador. Se puede usar el carácter retorno de carro para producir un resultado con más de una línea:

```
? "UNO\nDOS\nTRES"
UNO
DOS
TRES
```

Todos los caracteres tienen un número según un código ISO (International Standards Organisation)¹ Existen dos funciones estándar primitivas: una calcula el código de un carácter, y la otra devuelve el carácter relacionado a un código:

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

Por ejemplo:

```
? ord 'A'
65
? chr 51
'3'
```

Los caracteres están ordenados según este código. El tipo `Char` es parte de la clase `Ord`. El orden de las letras es el orden alfabético, estando todas las letras mayúsculas antes que las letras minúsculas. Este orden también se aplica a las cadenas; las cadenas son listas y tienen un orden lexicográfico basado en el orden de sus elementos:

```
? sort ["gato", "perro", "Jimena", "Ariel"]
["Ariel", "Jimena", "gato", "perro"]
```

3.2.3 Funciones de cadenas y caracteres

En el preludio están definidas algunas funciones sobre caracteres. Con estas funciones se puede determinar el grupo al que pertenece un carácter.

```
isSpace, isUpper, isLower, isAlpha, isDigit, isAlnum :: Char->Bool
isSpace c      = c == ' ' || c == '\t' || c == '\n'
isUpper c      = c >= 'A' && c <= 'Z'
isLower c      = c >= 'a' && c <= 'z'
isAlpha c      = isUpper c || isLower c
isDigit c      = c >= '0' && c <= '9'
isAlphanum c   = isAlpha c || isDigit c
```

Estas funciones son útiles para distinguir diferentes casos en una definición de una función de caracteres.

En el código ISO, el código del carácter ‘3’ no es 3, sino 51. Por suerte, los números están en orden sucesivo. Para determinar el valor numérico del carácter correspondiente a un dígito, se tiene que aplicar la función `ord`, y restar 48. Esto es lo que hace la función `digitValue`:

```
digitValue :: Char -> Int
digitValue c = ord c - ord '0'
```

¹Muchas veces se llama este código el código `ascii` (American Standard Code for Information Interchange). Ahora que el código está reconocido internacionalmente, se le debería llamar código `iso`.

Si se quiere aplicar la función a un dígito y no a otro valor, se puede escribir la siguiente definición:

```
digitValue c | isDigit c = ord c - ord '0'
```

La operación inversa es ejecutada por la función `digitChar`: esta función recibe un entero entre 0 y 9 devuelve el carácter correspondiente al dígito.

```
digitChar :: Int -> Char
digitChar n = chr (n + ord '0')
```

Estas dos funciones no están definidas en el prelude (pero si se necesitan, se las puede definir).

En el prelude ya se encuentran dos funciones para convertir letras minúsculas en letras mayúsculas y viceversa:

```
toUpper, toLower :: Char->Char
toUpper c | isLower c = chr (ord c - ord 'a' + ord 'A')
          | otherwise = c
toLower c | isUpper c = chr (ord c - ord 'A' + ord 'a')
          | otherwise = c
```

Con la función `map` se pueden aplicar estas funciones a todos los elementos de una cadena:

```
? map toUpper "Hola!"
HOLA!
? map toLower "Hola!"
hola!
```

Todas las funciones polimórficas que se pueden aplicar a listas, se pueden aplicar también a cadenas. Existen también algunas funciones en el prelude que se aplican únicamente a cadenas (listas de caracteres):

```
words, lines :: [Char] -> [[Char]]
unwords, unlines :: [[Char]] -> [Char]
```

La función `words` subdivide una cadena en una lista de subcadenas. Cada una de ellas contiene una palabra de la cadena de entrada. Se separa las palabras por espacios (como en `isSpace`). La función `lines` hace lo mismo, pero en este caso subdivide las subcadenas que están separadas en la cadena de entrada por el carácter retorno de carro (`'\n'`). Ejemplos:

```
? words "esta es una cadena"
["esta", "es", "una", "cadena"]
? lines "primera linea\nsegunda linea"
["primera linea", "segunda linea"]
```

Las funciones `unwords` y `unlines` hacen lo contrario: una lista de palabras, una lista de líneas respectivamente, resultan en una gran lista de caracteres:

```
? unwords ["estas", "son", "las", "palabras"]
estas son las palabras
? unlines ["primera linea", "segunda linea"]
primera linea
segunda linea
```

Obsérvese que en el resultado no existen comillas: estas nunca están si el resultado de una expresión es una cadena.

Una variante de la función `unlines` es la función `layn`. Esta numera los renglones del resultado:

```
? layn ["primera linea", "segunda linea"]
  1) primera linea
  2) segunda linea
```

Las definiciones exactas de estas funciones están en el preludio; lo más importante ahora es que se las pueda usar para conseguir un resultado claro.

3.2.4 Listas infinitas

El número de elementos de una lista puede ser infinito. La siguiente función `desde` devuelve una lista infinita:

```
desde n = n : desde (n+1)
```

Es claro que un ordenador no puede contener un número infinito de elementos. Afortunadamente, se puede ver la parte inicial de la lista mientras se está calculando el resto de la lista:

```
? desde 5
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, control-C
{Interrupted!}
?
```

Si ya vio bastantes elementos, puede detener el cálculo tecleando control-C.

Una lista infinita se puede usar también como resultado intermedio en una computación, aunque el resultado final sea finito. Esto ocurre por ejemplo con el problema: ‘calcula todas las potencias de tres que sean menores que 1000’. Las primeras diez potencias de tres pueden ser calculadas por la llamada:

```
? map (3^) [1..10]
[3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049]
```

Los elementos que son menores que 1000 pueden ser elegidos haciendo uso de la función `takeWhile`:

```
? takeWhile (<1000) (map (3^) [1..10])
[3, 9, 27, 81, 243, 729]
```

¿Pero, cómo se sabe si 10 elementos son suficientes? La solución es que no se usa una lista `[1..10]`, sino la lista infinita `desde 1`, para calcular con esa *todas* las potencias de tres. Esto es absolutamente suficiente. . .

```
? takeWhile (<1000) (map (3^) (desde 1))
[3, 9, 27, 81, 243, 729]
```

Este método puede ser aplicado gracias al hecho de que el intérprete es bastante perezoso: siempre trata de aplazar el trabajo. Por eso, no se va a calcular el resultado de `map (3^) (desde 1)` completamente (tardaría un tiempo infinito). En vez de eso, primero calcula el primer elemento de la lista. Este se devuelve a aquellos que quieren algo con la lista, en este caso la función `takeWhile`. Solamente si se ha utilizado este elemento y `takeWhile` pide el siguiente elemento, se calcula el segundo elemento. En algún momento, `takeWhile` no pedirá el siguiente elemento (después de que el primer número sea ≥ 1000). Por tanto, los otros elementos no serán calculados por `map`.

3.2.5 Evaluación perezosa

El método de evaluación (la manera en que se calculan las expresiones) de Gofer se llama *evaluación perezosa* (en inglés ‘lazy evaluation’). Con evaluación perezosa se calcula una expresión (parcial) solamente

si realmente se necesita el valor para calcular el resultado. El opuesto de evaluación perezosa es *evaluación voraz* (en inglés ‘eager evaluation’). Con evaluación voraz se calcula directamente el resultado de la función, si se conoce el parámetro actual.

Poder usar listas infinitas es posible gracias a evaluación perezosa. En lenguajes que usan evaluación voraz (como todos los lenguajes imperativos y algunos de los lenguajes funcionales más antiguos), las listas infinitas no son posibles.

La evaluación perezosa tiene más ventajas. Véase por ejemplo la función `primo` en el párrafo 2.4.1, que comprueba si un número es primo: pág. 31

```
primo :: Int -> Bool
primo x = divisores x == [1,x]
```

¿Calculará esta función todos los divisores de `x`, y comparará el resultado con `[1,x]`? ¡No, esto es demasiado trabajo! Con la llamada de `primo 30` pasa lo siguiente: Primero se calcula el primer divisor de 30: 1. Se compara este valor con el primer elemento de la lista `[1,30]`. Para el primer elemento las listas son iguales. Después se calcula el segundo divisor de 30: 2. Se compara el resultado con el segundo valor de `[1,30]`: los segundos elementos no son iguales. El operador `==` ‘sabe’ que las dos listas nunca pueden ser iguales si existe un elemento que difiere. Por eso se puede devolver directamente el valor `False`. ¡Los otros divisores de 30 no se calculan!

El comportamiento perezoso del operador `==` es debido a su definición. La regla recursiva en su definición en el párrafo 3.1.2 es: pág. 41

```
(x:xs) == (y:ys) = x==y && xs==ys
```

Si `x==y` devuelve el valor `False`, no hace falta calcular `xs==ys`: el resultado siempre es `False`. Este comportamiento perezoso del operador `&&` es debida a su definición:

```
False && x = False
True  && x = x
```

Si el parámetro a la izquierda tiene el valor `False`, no se necesita el valor del parámetro a la derecha para calcular el resultado. (Ésta es la definición real de `&&`. La definición en el párrafo 1.4.3 también funciona, pero no muestra el comportamiento deseado). pág. 13

Las funciones que necesitan todos los elementos de una lista, no pueden ser aplicadas a listas infinitas. Ejemplos de tales funciones son `sum` y `length`. La evaluación perezosa tampoco ayuda a calcular el resultado en tiempo finito con las llamadas `sum (desde 1)` y `length (desde 1)`. El ordenador entra en un cálculo infinito, y nunca devolverá una respuesta (a no ser que no se necesite el resultado del cálculo, ya que en este caso, no se realiza el cálculo...).

3.2.6 Funciones sobre listas infinitas

En el preludio están definidas algunas funciones que devuelven listas infinitas.

La función `desde` en el párrafo 3.2.4 se llama en realidad `enumFrom`. Normalmente no se usa esta función porque en vez de `enumFrom n` también se puede escribir `[n..]`. (Compárese la notación `[n..m]` con `enumFromTo n m`, que vimos en el párrafo 3.1.1). pág. 52

Una lista infinita en que se repite un solo elemento puede ser definida con la función `repeat`: pág. 40

```
repeat :: a -> [a]
repeat x = x : repeat x
```

La llamada `repeat 't'` devuelve la lista infinita `"ttttttttt..."`

Una lista infinita que es generada por `repeat`, puede ser usada como resultado intermedio por una función que tiene un resultado finito. La función `copy` por ejemplo, hace un número finito de copias de un elemento:

```
copy    :: Int -> a -> [a]
copy n x = take n (repeat x)
```

Gracias a la evaluación perezosa, `copy` puede usar el resultado infinito de `repeat`. Las funciones `repeat` y `copy` están definidas en el preludio.

La función más flexible es siempre una función de orden superior (una función con una función como parámetro). La función `iterate` tiene una función y un elemento inicial como parámetros. El resultado es una lista infinita, en que cada siguiente elemento es el resultado de la aplicación de la función al elemento anterior. Por ejemplo:

```
iterate (+1) 3      es [3, 4, 5, 6, 7, 8, ...
iterate (*2) 1      es [1, 2, 4, 8, 16, 32, ...
iterate (/10) 5678  es [5678, 567, 56, 5, 0, 0, ...
```

La definición de `iterate`, que está definida en el preludio, es como sigue:

```
iterate    :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

pág. 29

Esta función se parece un poco a la función `until`, que definimos en el párrafo 2.3.2. También `until` tiene una función y un elemento inicial como parámetros, y aplica la función repetidamente al elemento inicial. La diferencia es que `until` para si el resultado cumple con una condición (también es un parámetro de `until`). Además, `until` devuelve solamente el resultado final (el resultado que cumple con la condición), mientras que `iterate`, concatena todos los resultados parciales en una lista. Tiene que hacer esto, porque en listas infinitas no existe un último elemento...

Vamos a ver dos ejemplos en que se usa `iterate` para resolver un problema práctico: la presentación de un número como cadena y la generación de la lista de todos los primos.

Presentación de un número como cadena

La función `intString` hace de un número una cadena de caracteres que contiene los dígitos de este número. Por ejemplo: `intString 5678` devuelve la cadena "5678". Gracias a esta función es posible combinar el resultado de un cálculo con una cadena, por ejemplo, `intString (3*17)+"lineas"`.

La función `intString` se puede definir ejecutando una serie de funciones sucesivamente. Primero tenemos que dividir repetidamente el número por 10, usando `iterate` (como en el tercer ejemplo de `iterate` arriba). La cola infinita de ceros no interesa, por tanto se la puede eliminar con `takeWhile`. Las cifras que necesitamos son las últimas cifras de los números en la lista; la última cifra de un número es el resto de dividir por 10. Las cifras están todavía en el orden inverso, pero esto se puede resolver usando la función `reverse`. Finalmente, se tienen que cambiar las cifras (del tipo `Int`) por los caracteres correspondientes (del tipo `Char`).

Un diagrama en base a un ejemplo explica la idea:

```

5678
  ↓ iterate (/10)
[5678, 567, 56, 5, 0, 0, ...
  ↓ takeWhile (/=0)
[5678, 567, 56, 5]
  ↓ map ('rem'10)
[8, 7, 6, 5]
  ↓ reverse
[5, 6, 7, 8]
  ↓ map digitChar
['5', '6', '7', '8']
```

La función `intString` es la composición de estas cinco funciones. Cuidado con el orden de las aplicaciones de las funciones; el operador composición de funciones `(.)` tiene el sentido ‘después de’:

```
intString :: Int -> [Char]
intString = map digitChar
            . reverse
            . map ('rem'10)
            . takeWhile (/=0)
            . iterate (/10)
```

¡La programación funcional consiste en programar con funciones!

La lista de todos los primos

En el párrafo 2.4.1 se definió una función `primo`, que determina si un número es primo. La lista (infinita) de todos los primos se puede calcular con:

pág. 31

```
filter primo [2..]
```

La función `primo` busca todos los divisores de un número. Si un divisor es bastante grande, se necesita mucho tiempo hasta que la función sabe que el número no es primo.

Usando hábilmente la función `iterate`, es posible obtener un método más eficiente. Este método también empieza con la lista infinita `[2..]`:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...]
```

El primer número, 2, se puede incluir en la lista de los primos. A continuación se pueden eliminar 2 y sus múltiplos de la lista. Lo que queda es:

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...]
```

El primer número, 3, es un primo. Se puede eliminar este número y sus múltiplos en la lista:

```
[5, 7, 11, 13, 17, 19, 23, 25, 29, 31, ...]
```

Otra vez el mismo proceso, pero ahora con 5:

```
[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...]
```

Se puede seguir de esta manera. Se aplica cada vez la función ‘elimina los múltiplos del primer elemento’ al resultado anterior. Esta es una aplicación de `iterate` con `[2..]` como valor inicial:

```
iterate elimina [2..]
where elimina (x:xs) = filter (not.multiplo x) xs
      multiplo x y = divisible y x
```

Como el valor inicial es una lista infinita, el resultado de la nueva función es una *lista infinita de listas infinitas*. Esta super lista está construida como sigue:

```
[ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
  , [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...
  , [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, ...
  , [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, ...
  , [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 51, 53, ...
  , ...
```

Esta cosa nunca se puede ver totalmente; si se trata de evaluarla, solamente se puede ver la parte inicial de la primera lista. Pero la lista completa no importa, los primos son los primeros elementos de las listas. Por tanto, los primos son calculados tomando de cada lista infinita su primer elemento (usando la función `head`):

```
numerosPrimos :: [Int]
numerosPrimos = map head (iterate eliminar [2..])
                where eliminar (x:xs) = filter (not.multiplo x) xs
```

Por evaluación perezosa, se calcula de cada lista de la super lista exactamente la parte que es necesaria para la respuesta deseada. Si se quiere conocer el siguiente primo, se calcula un poco más de cada lista.

Muchas veces (como en este ejemplo) es difícil comprender qué pasa exactamente durante la ejecución. Pero tampoco es necesario saberlo: si se está escribiendo un programa, se puede suponer que las listas infinitas realmente existen; el 'cómo' ejecutar las cosas es algo para el intérprete. Con el mecanismo de evaluación perezosa se optimiza el orden de la ejecución.

3.3 Tuplas

3.3.1 Uso de tuplas

Cada elemento en una lista debe ser del mismo tipo. No es posible crear una lista de enteros y caracteres. Sin embargo, algunas veces es necesario agrupar elementos de diferentes tipos. Por ejemplo, la información en un registro de personas puede contener nombres (cadenas), si alguien es masculino o no (booleano) y fechas de nacimiento (enteros). Estos datos se corresponden, pero no se los puede poner en una lista.

Para esto, además de listas, tenemos otra manera de construir tipos compuestos: las *tuplas*. Una *tupla* consiste en un número fijo de valores, que están agrupados como una entidad. Los valores pueden ser de diferentes tipos (pero no es obligatorio).

Las tuplas se anotan con paréntesis, los elementos que se agrupan están entre paréntesis. Ejemplos de tuplas son:

<code>(1, 'a')</code>	una tupla con los elementos 1 (entero) y 'a' (carácter);
<code>("mono", True, 2)</code>	una tupla con tres elementos: la cadena "mono", el booleano <code>True</code> y el número 2;
<code>([1,2], sqrt)</code>	una tupla con dos elementos: la lista de enteros <code>[1,2]</code> , y la función del tipo de-float-a-float <code>sqrt</code> ;
<code>(1, (2,3))</code>	una tupla con dos elementos: el número 1, y la tupla de los números 2 y 3.

Para cada combinación de tipos se crea un nuevo tipo. El orden de los elementos importante. El tipo de una tupla está definido por los tipos de sus elementos entre paréntesis. Por tanto, las cuatro expresiones de arriba tienen los siguientes tipos:

```
(1, '{a}')      :: (Int, Char)
("mono", True, 2) :: ([Char], Bool, Int)
([1,2], sqrt)   :: ([Int], Float->Float)
(1, (2,3))      :: (Int, (Int,Int))
```

Una tupla con dos elementos se llama *pareja*. Las tuplas con tres elementos se llaman ternas. No existen tuplas de un elemento: la expresión `(7)` simplemente es un entero; tal expresión se puede escribir entre paréntesis. Sin embargo, existe la tupla de 0 elementos: el valor `()`, que tiene `()` como tipo.

En el preludio están definidas algunas funciones que se aplican a parejas o ternas. Las siguientes son ejemplos de cómo se tienen que definir funciones sobre tuplas: mediante análisis por patrones.

```

fst      :: (a,b) -> a
fst (x,y) = x
snd      :: (a,b) -> b
snd (x,y) = y
fst3     :: (a,b,c) -> a
fst3 (x,y,z) = x
snd3     :: (a,b,c) -> b
snd3 (x,y,z) = y
thd3     :: (a,b,c) -> c
thd3 (x,y,z) = z

```

Estas funciones son polimórficas, pero por supuesto también es posible escribir funciones que operen solamente con un tipo de tupla determinado:

```

f      :: (Int,Char) -> [Char]
f (n,c) = intString n ++ [c]

```

Si se quieren agrupar dos elementos del mismo tipo, se puede usar una lista. Pero, en algunos casos, una tupla es más útil. Un punto en dos dimensiones se describe por ejemplo por dos números de tipo `Float`. Tal punto puede ser descrito por una lista o por una pareja. En los dos casos es posible definir funciones que hagan algo con puntos. Por ejemplo, una función que calcule la distancia hasta el origen. La función `distanciaL` es la versión que usa una lista, `distanciaT` es la versión para una tupla:

```

distanciaL :: [Float] -> Float
distanciaL [x,y] = sqrt (x*x+y*y)
distanciaT :: (Float,Float) -> Float
distanciaT (x,y) = sqrt (x*x+y*y)

```

Mientras se llame la función correctamente, no existe diferencia. Sin embargo, podría suceder que se llamase a la función (por error al teclear, o error lógico) en algún lugar en el programa con tres coordenadas. Con el uso de `distanciaT`, el intérprete da un aviso durante el análisis de las funciones: una tupla con tres elementos es diferente de una tupla con dos elementos. En el caso de `distanciaL` el intérprete no dice nada: el programa no tiene conflictos de tipos. Pero cuando se usa el programa, resulta que la función `distanciaL` no está definida para listas con tres elementos. El uso de tuplas en vez de listas es entonces útil para detectar errores lo antes posible.

Otro caso en que las tuplas son realmente útiles es cuando se necesitan funciones con más de un resultado. Las funciones de varios parámetros son posibles gracias al mecanismo de currificación; las funciones que tienen varios resultados solamente son posibles agrupando los resultados en una tupla. La tupla en total es entonces un resultado.

Un ejemplo de una función que en realidad tiene dos resultados, es la función `splitAt` que está definida en el preludio. Esta función devuelve los resultados de `take` y `drop` de una vez. Por tanto, se puede definir la función de la siguiente forma:

```

splitAt    :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

```

El trabajo de estas dos funciones se puede hacer de una vez, por eso se ha definido realmente la función `splitAt` de forma más eficiente:

```

splitAt    :: Int -> [a] -> ([a], [a])
splitAt 0  xs    = ([], xs)
splitAt n  []    = ([], [])
splitAt (n+1) (x:xs) = (x:ys, zs)
                    where (ys,zs) = splitAt n xs

```

La llamada `splitAt 2 "gofer"` devuelve la pareja `("go","fer")` como resultado. En la definición se puede ver (con una llamada recursiva) cómo se puede usar una tupla resultante: utilizando patrones (en el ejemplo `(ys,zs)`).

3.3.2 Definiciones de tipos

Cuando se usan mucho las listas y tuplas, las declaraciones de tipos llegan a ser bastante complicadas. Por ejemplo, con las funciones sobre puntos como la función `distancia` de la anterior página. Las funciones más simples son entendibles:

```
distancia  :: (Float,Float) -> Float
diferencia :: (Float,Float) -> (Float,Float) -> Float
```

Sin embargo, es más complicado con listas sobre puntos, y con funciones de orden superior:

```
sup_poligono  :: [(Float,Float)] -> Float
transf_poligono :: ((Float,Float)->(Float,Float))
                 -> [(Float,Float)] -> [(Float,Float)]
```

Una *definición de tipo* es muy útil en estos casos. Con una definición de tipo es posible dar un nombre (más claro) a un tipo, por ejemplo:

```
type Punto = (Float,Float)
```

Con esta definición se pueden escribir las declaraciones de tipo de forma más clara:

```
distancia  :: Punto -> Float
diferencia :: Punto -> Punto -> Float
sup_poligono  :: [Punto] -> Float
transf_poligono :: (Punto->Punto) -> [Punto] -> [Punto]
```

Mejor todavía sería hacer una definición de tipo para ‘polígono’:

```
type Poligono  = [Punto]
sup_poligono  :: Poligono -> Float
transf_poligono :: (Punto->Punto) -> Poligono -> Poligono
```

Con definiciones de tipo hay que tener presentes algunos puntos:

- la palabra `type` es una palabra reservada especialmente para estas definiciones de tipo;
- el nombre de un tipo debe empezar por una letra mayúscula (es una constante, no es una variable);
- una *declaración* de un tipo especifica el tipo de una función; una *definición* de tipo define un nuevo nombre para un tipo.

El intérprete usa el nombre definido simplemente como una abreviatura. Si se pide el tipo de una expresión al intérprete, éste muestra `(Float,Float)` en vez de `Punto`. Si se dan dos diferentes nombres a un tipo, por ejemplo en:

```
type Punto      = (Float,Float)
type Complejo   = (Float,Float)
```

Entonces se pueden usar los dos nombres indistintamente. Un `Punto` es lo mismo que un `Complejo` y éste es lo mismo que un `(Float,Float)`. En el párrafo 3.4.3 se describe un método para definir `Punto` como un tipo realmente *nuevo*.

3.3.3 Números racionales

Una aplicación en la cual es muy útil usar tuplas es una implementación de números racionales. Los números racionales forman el conjunto matemático \mathbf{Q} de números que se pueden escribir como *fracciones*. No se puede calcular con números racionales utilizando números reales (`Float`): si se quiere calcular exactamente, el resultado de $\frac{1}{2} + \frac{1}{3}$ debe devolver la fracción $\frac{5}{6}$, y no el número `Float 0.833333`.

Los números racionales (fracciones) pueden representarse por un numerador y un denominador, ambos números enteros. Por tanto, tiene sentido la siguiente definición de tipo:

```
type Ratio = (Int,Int)
```

A continuación definimos nombres especiales para fracciones utilizadas frecuentemente:

```
qCero   = (0, 1)
qUno    = (1, 1)
qDos    = (2, 1)
qMedio  = (1, 2)
qTercio = (1, 3)
qCuarto = (1, 4)
```

Queremos escribir funciones para las operaciones básicas sobre números racionales:

```
qMul    :: Ratio -> Ratio -> Ratio
qDiv    :: Ratio -> Ratio -> Ratio
qSum    :: Ratio -> Ratio -> Ratio
qRes    :: Ratio -> Ratio -> Ratio
```

Un problema es que un valor puede ser representado por varias fracciones. Un ‘medio’ por ejemplo, está representado por la tupla (1,2), pero también por (2,4) y (17,34). Por ello, puede ser que el resultado de dos por un cuarto ‘difiera de’ un medio ((1,2)). Para resolver este problema necesitamos una función `simplificar`, que simplifique una fracción. Aplicando esta función después de cada operador sobre fracciones, resulta en una representación única para una fracción. Entonces se puede comparar el resultado de dos por un cuarto con un medio: el resultado será `True`.

La función `simplificar` divide numerador y denominador de una fracción por su *máximo común divisor*. El máximo común divisor de dos números es el número más grande por el que se pueden dividir ambos. Además, `simplificar` asegura que el símbolo menos (si existe) esté en el numerador de la fracción. La definición es:

```
simplificar (t,n) = ( (signum n * t)/d, abs n/d )
                  where d = mcd t n
```

Una definición simple de `mcd x y` determina el divisor más grande de `x` por el que se puede dividir `y`, usando las funciones `divisores` y `divisible` en el párrafo 2.4.1:

```
mcd x y = last (filter (divisible y') (divisores x'))
          where x' = abs x
                y' = abs y
```

(En el preludio está definida otra función `gcd` (*greatest common divisor*), que es más eficiente:

```
gcd x y = gcd' (abs x) (abs y)
          where gcd' x 0 = x
                gcd' x y = gcd' y (x `rem` y)
```

Este algoritmo se basa en el hecho de que si x y y son divisibles por d , entonces $x \text{ rem } y (=x - (x/y) * y)$ es divisible por d .

Utilizando la función `simplificar` se pueden definir las funciones aritméticas. Para multiplicar dos fracciones, se tiene que multiplicar el numerador y el denominador ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Después, se puede simplificar el resultado (a $\frac{5}{6}$):

```
qMul (x,y) (p,q) = simplificar (x*p, y*q)
```

Dividir por un número es multiplicar por el inverso, entonces:

```
qDiv (x,y) (p,q) = simplificar (x*q, y*p)
```

Sumar dos fracciones es más trabajoso: primero se tiene que reducir los quebrados al mismo denominador común ($\frac{1}{4} + \frac{3}{10} = \frac{10}{40} + \frac{12}{40} = \frac{22}{40}$). Se puede usar el producto de los denominadores como denominador común. Luego, se tiene que multiplicar los numeradores con los denominadores de la otra fracción. Después de que se puedan sumar los resultados, se simplifica el resultado (a $\frac{11}{20}$).

```
qSum (x,y) (p,q) = simplificar (x*q+y*p, y*q)
qRes (x,y) (p,q) = simplificar (x*q-y*p, y*q)
```

El resultado de los cálculos con números racionales se muestra en la pantalla como tupla. Si eso no es suficientemente bonito, se puede definir una función `ratioString`:

```
ratioString :: Ratio -> String
ratioString (x,y)
  | y'==1    = intString x'
  | otherwise = intString x' ++ "/" ++ intString y'
              where (x',y') = simplificar (x,y)
```

3.3.4 Listas y tuplas

Las tuplas muchas veces son elementos de una lista. A menudo se usa una lista de parejas como lista de búsqueda (diccionario, guía de teléfono, etc.). La función de búsqueda se puede definir fácilmente usando patrones; para la lista se usa un patrón para 'la lista no vacía que tiene como primer elemento una pareja' (los otros elementos también son por tanto parejas).

```
buscar :: Eq a => [(a,b)] -> a -> b
buscar ((x,y):ts) z
  | x == z    = y
  | otherwise = buscar ts z
```

La función es polimórfica, opera sobre cualquier lista de parejas. Pero se deben poder comparar los elementos, por tanto el tipo a debe estar en la clase `Eq`.

El elemento que se busca (del tipo a), está definido como segundo parámetro para que se pueda instanciar parcialmente la función con una lista de búsqueda determinado, por ejemplo:

```
numeroDeTel = buscar guiaDeTel
traduccion  = buscar diccionario
```

(`guiaDeTel` y `diccionario` se pueden definir como constantes).

Otra función que utiliza listas de parejas es la función `zip`. Esta función está definida en el preludio. La función `zip` tiene dos listas como parámetros, que se conectan elemento a elemento en el resultado. Por ejemplo: `zip [1,2,3] "abc"` devuelve la lista `[(1,'a'),(2,'b'),(3,'c')]`. Si las dos listas parámetros no tienen el mismo tamaño, entonces la más corta determina el tamaño del resultado. La definición es:


```

zip      :: [a] -> [b] -> [(a,b)]
zip []   ys      = []
zip xs   []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

```

La función es polimórfica, se la puede aplicar a listas con elementos de cualquier tipo.

Una variante de `zip` es la función `zipWith`. Esta función tiene a dos listas y además una función como parámetro que dice cómo se tienen que combinar los elementos:

```

zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f []   ys      = []
zipWith f xs   []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

```

Esta función aplica una función (con dos parámetros) a todos los elementos de las dos listas. La función `zipWith` se parece mucho a `zip`, pero también a `map`, que aplica una función (con un parámetro) a todos los elementos de una lista.

Dada la función `zipWith`, se puede definir la función `zip` como su instanciación parcial:

```

zip = zipWith hacerPareja
    where hacerPareja x y = (x,y)

```

3.3.5 Tuplas y currificación

Usando tuplas se pueden escribir funciones con más de un parámetro sin usar el mecanismo de currificación. Una función puede tener una tupla como único parámetro, con el que se da más de un valor:

```

mas (x,y) = x+y

```

Esta función parece muy normal. Mucha gente diría que es una función con dos parámetros, y que los parámetros, por supuesto, están entre paréntesis. Pero ya sabemos: esta función tiene un parámetro, una tupla; la definición usa un patrón para una tupla.

Es mejor utilizar funciones currificadas que en forma cartesiana. Con currificación se puede instanciar parcialmente una función, con una función de un parámetro tupla no se puede. Todas las funciones estándar con más de un solo parámetro se definen de forma currificada.

En el preludio esta definida una transformación de función (una función con una función como parámetro y otra función como resultado), que transforma una función currificada en una función con un parámetro de tupla. Esta función se llama `uncurry`:

```

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

```

Existe también una función `curry` que transforma una función con un parámetro de tupla en una función currificada. Por tanto, `curry mas`, con `mas` como está definida arriba, puede ser instanciada parcialmente.

3.4 Árboles

3.4.1 Definiciones de datos

Listas y tuplas son dos maneras primitivas de estructurar datos. Si estas dos formas no ofrecen todo lo que se necesita para representar determinada información, se puede definir un nuevo *tipo de datos*.

Un tipo de datos es un tipo que está determinado por la forma en que se pueden construir los elementos. Una 'lista' es un tipo de datos. Los elementos de una lista se pueden construir de dos maneras:

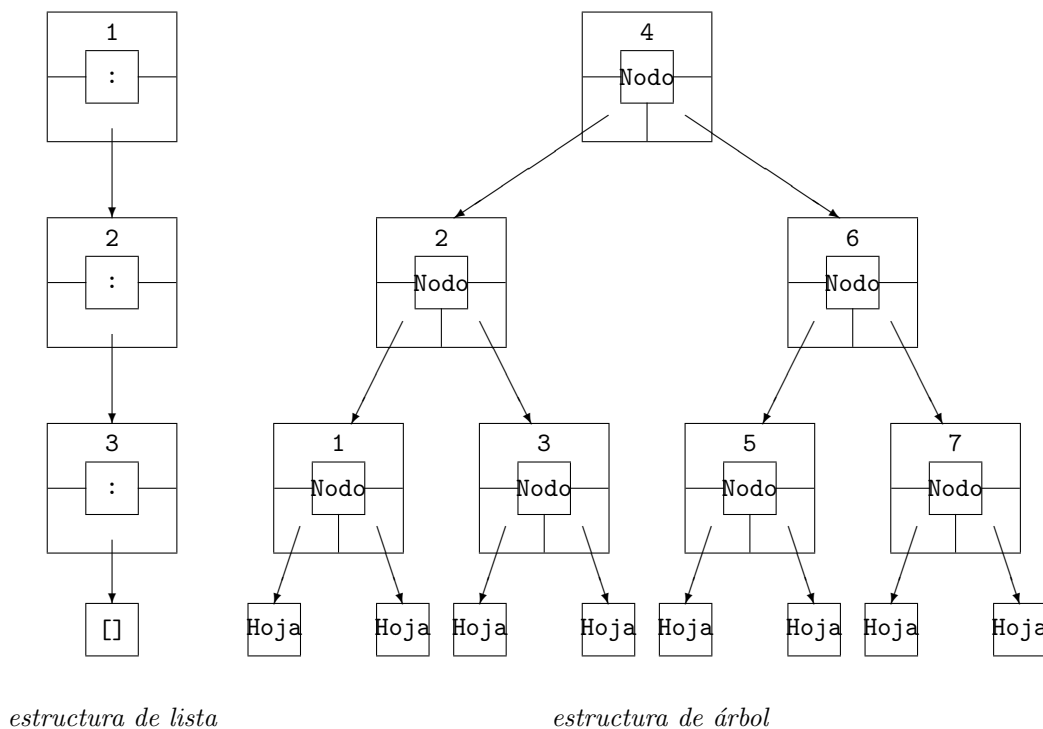


Figura 3.1:

- como la lista vacía;
- aplicando a un elemento y a una lista (más pequeña) el operador `:`.

Estas dos maneras se convierten en los patrones de definiciones de funciones sobre listas, por ejemplo:

```
length []      = 0
length (x:xs) = 1 + length xs
```

Al definir la función para los patrones ‘lista vacía’ y ‘la aplicación del operador `:` a un elemento y una lista’, se definió la función para todos los casos que pueden ocurrir.

Una lista es una estructura lineal; cada vez que se aumenta un elemento, la lista resultante es más larga. Algunas veces, no se quiere una estructura lineal, sino una estructura de *árbol*. Existen diferentes estructuras de árbol. En la figura 3.1 se compara una lista con un árbol que se divide en dos en cada nodo. La estructura de la lista usa el operador `:` con dos parámetros (están en la figura), o con la lista vacía (`[]`), sin parámetros. El árbol no es construido con operadores, sino con funciones: `Nodo` (con tres parámetros) o `Hoja` (sin parámetros).

Las funciones que se usan para construir una estructura de datos se llaman *funciones constructoras*. Las funciones constructoras del árbol son `Nodo` y `Hoja`. Los nombres de funciones constructoras empiezan por una letra mayúscula para distinguirlas de funciones normales. Si se escribe una función constructora como operador, el nombre debe empezar con el símbolo ‘dos puntos’. La función constructora (`:`) de listas es un ejemplo de un operador como función constructora. El constructor `[]` es la única excepción.

Con una *definición de datos* se definen las funciones constructoras que se pueden usar con un nuevo tipo. En esta definición de datos están también los tipos de los parámetros de las funciones constructoras. Por ejemplo, la definición de datos para árboles como están descritos arriba es:

```
data Arbol a = Nodo a (Arbol a) (Arbol a)
             | Hoja
```

Se puede leer esta definición como: ‘Un árbol con elementos del tipo `a` (árbol sobre `a`) puede ser construido de dos formas: (1) aplicando la función `Nodo` a tres parámetros (uno del tipo `a` y dos del tipo árbol sobre `a`), o (2) usando la constante `Hoja`.’

Los árboles pueden formarse usando las funciones constructoras en una expresión. El árbol que está descrito en la figura, se representa en la siguiente expresión:

```
Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
           (Nodo 3 Hoja Hoja)
        )
      (Nodo 6 (Nodo 5 Hoja Hoja)
           (Nodo 7 Hoja Hoja)
      )
    )
```

No hace falta escribir esto de una forma tan bonita; se puede escribir también:

```
Nodo 4(Nodo 2(Nodo 1 Hoja Hoja)(Nodo 3 Hoja Hoja))
      (Nodo 6(Nodo 5 Hoja Hoja)(Nodo 7 Hoja Hoja))
    )
```

La primera construcción es más clara. Se tiene que usar también la regla descrita en el párrafo 1.4.5.

pág. 16

Las funciones sobre árboles pueden definirse mediante análisis por patrones con las funciones constructoras. La siguiente función por ejemplo, determina el número de elementos en un árbol:

```
tamano      :: Arbol a -> Int
tamano Hoja      = 0
tamano (Nodo x p q) = 1 + tamano p + tamano q
```

Compárese esta función con la función `length` sobre listas.

Existen otras formas de árboles. Algunos ejemplos:

- Árboles cuyos elementos no están en los nodos (como en `Arbol`), sino que están solamente en las hojas:

```
data Arbol2 a = Nodo2 (Arbol2 a) (Arbol2 a)
              | Fin2 a
```

- Árboles con información del tipo `a` en los nodos, e información del tipo `b` en las hojas:

```
data Arbol3 a b = Nodo3 a (Arbol3 a b) (Arbol3 a b) | Fin3 b
```

- Árboles que se dividen en tres en los nodos, y no en dos:

```
data Arbol4 a = Nodo4 a (Arbol4 a) (Arbol4 a) (Arbol4 a)
              | Fin4
```

- Árboles cuyo número de ramas bifurcadas de un nodo son variables:

```
data Arbol5 a = Nodo5 a [Arbol5 a]
```

En este árbol no se necesita un constructor para ‘hoja’, porque se representa esto con un nodo con cero ramas bifurcadas.

- Árboles cuyos nodos solamente tienen una rama bifurcada:

```
data Arbol6 a = Nodo6 a (Arbol6 a)
              | Fin6
```

Un árbol de este tipo es en realidad una lista, tiene una estructura lineal.

- Árboles con diferentes tipos de nodos:

```
data Arbol7 a b = Nodo7a Int a (Arbol7 a b) (Arbol7 a b)
                | Nodo7b Char (Arbol7 a b)
                | Fin7a b
                | Fin7b Int
```

3.4.2 Árboles de búsqueda

Un buen ejemplo de una situación en la que el uso de árboles es mejor que el uso de listas es cuando se busca un valor en un gran conjunto de datos. Para esto se pueden usar *árboles de búsqueda*.

pág. 44

En el párrafo 3.1.2 se definió la función `elem`, que devuelve `True` si un elemento está en una lista. No importa mucho para la eficiencia si esta función se define con las funciones estándar `map` y `or`

```
elem      :: Eq a => a -> [a] -> Bool
elem e xs = or (map (==e) xs)
```

o directamente mediante recursión

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

En los dos casos se inspeccionan los elementos de la lista uno por uno. Si se encuentra el elemento, la función da directamente un resultado (gracias a la evaluación perezosa), pero si el elemento no está, la función debe analizar todos los elementos para descubrirlo.

Un poco más eficiente es la situación en la que la función supone que la lista en que se busca está ordenada. Si los elementos están en un orden ascendente, se puede detener el proceso de búsqueda cuando se encuentre un valor que sea mayor al valor buscado. El precio es que no es suficiente poder comparar los elementos (la clase `Eq`), también debe poderse ordenarlos (la clase `Ord`):

```
elem'      :: Ord a => a -> [a] -> Bool
elem' e []      = False
elem' e (x:xs) | e>x = False
               | e==x = True
               | e<x = elem' e xs
```

Una gran mejora es colocar los elementos en un *árbol de búsqueda* en vez de colocarlos en una lista. Un árbol de búsqueda es más o menos un 'árbol ordenado'. Es un árbol construido según la definición de `Arbol` en la sección anterior:

```
data Arbol a = Nodo a (Arbol a) (Arbol a)
              | Hoja
```

En cada nodo hay un elemento, y existen dos subárboles: uno a la izquierda, uno a la derecha (ver la figura en pag. 62). En un árbol de búsqueda se exige que todos los valores en el subárbol izquierdo sean *menores* que el valor en el nodo mismo, y que todos los valores en el subárbol derecho sean *mayores*. Se eligieron los valores en el ejemplo de tal forma que el árbol representado sea un árbol de búsqueda.

Buscar un valor en un árbol de búsqueda es muy simple. Si el valor buscado es igual que el valor en el nodo actual, el valor existe y se ha encontrado. Si el valor buscado es menor que el valor en el nodo actual, entonces se tiene que buscar en el subárbol izquierdo. Sin embargo, si el valor buscado es mayor que el valor en el nodo actual, se tiene que buscar en el subárbol de la derecha. La función `elemArbol` es entonces como sigue:

```

elemArbol  :: Ord a => a -> Arbol a -> Bool
elemArbol e Hoja                = False
elemArbol e (Nodo x izq der)    | e==x = True
                                | e<x  = elemArbol e izq
                                | e>x  = elemArbol e der

```

Si el árbol está construido equilibradamente (*árbol balanceado*), a cada paso de la búsqueda se divide por la mitad el número de elementos que se deben registrar. De este modo, se encuentra rápidamente el elemento buscado o (si no existe), una hoja **Hoja**: un conjunto de mil elementos se debe dividir por la mitad solamente 10 veces, y un conjunto de un millón de elementos solamente veinte veces. Compárese esto con el promedio de medio millón de pasos que usa la función `elem` con un conjunto de un millón de elementos.

En el caso peor, buscar en un conjunto de n elementos requiere n pasos con `elem`, pero con `elemBoom` solamente requiere $\log_2 n$ pasos.

Es muy útil usar árboles de búsqueda si se tiene una gran cantidad de datos entre los que se tiene que buscar muchas veces. También se puede buscar más rápido en combinación con (por ejemplo) la función `buscar` en el párrafo 3.3.4.

pág. 60

Construcción de un árbol de búsqueda

La forma de un árbol de búsqueda puede determinarse ‘a mano’ para una colección de datos determinada. Se puede teclear como una gran expresión con muchas funciones constructoras. Sin embargo, es mucho trabajo que sencillamente no le gusta a nadie y que se puede automatizar.

Tal como la función `insert` que coloca un elemento en su lugar en una lista ordenada (ver el párrafo 3.1.4), así coloca la función `insertArbol` un elemento en un árbol de búsqueda. El resultado también es un árbol de búsqueda. Es decir, el elemento se inserta en el lugar que le corresponde:

pág. 47

```

insertArbol :: Ord a => a -> Arbol a -> Arbol a
insertArbol e Hoja                = Nodo e Hoja Hoja
insertArbol e (Nodo x izq der)    | e<=x = Nodo x (insertArbol e izq) der
                                | e>x  = Nodo x izq (insertArbol e der)

```

En el caso en que se añada un elemento a un árbol vacío (**Hoja**), se construye un árbol pequeño con `e` y dos subárboles vacíos. En el otro caso, si no es un árbol vacío, entonces contiene un valor `x`. Este valor se usa para decidir si se tiene que insertar `e` en el subárbol izquierdo o en el subárbol derecho.

Usando la función `insertArbol` repetidamente, se pueden insertar todos los elementos de una lista en un árbol de búsqueda:

```

listaArbol :: Ord a => [a] -> Arbol a
listaArbol = foldr insertArbol Hoja

```

Compárese esta función con la función `isort` en el párrafo 3.1.4.

pág. 47

El uso de `listaArbol` tiene la desventaja de que el resultado (un árbol de búsqueda) no siempre está balanceado. Si los datos que se insertan están desordenados, normalmente no hay problema. Pero, si la lista que se transforma a un árbol de búsqueda está ordenada, el resultado será un árbol extraño:

```

? listaArbol [1..7]
Nodo 7 (Nodo 6 (Nodo 5 (Nodo 4 (Nodo 3 (Nodo 2 (Nodo 1 Hoja Hoja)
Hoja) Hoja) Hoja) Hoja) Hoja) Hoja

```

Éste es un árbol de búsqueda, pero ha crecido torcido, de modo que el resultado es casi una estructura alineada. Esto empeora la eficiencia (ahora es igual a la eficiencia de buscar en una lista ordenada). Un árbol mejor sería:

```

Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)

```

```

(Nodo 3 Hoja Hoja)
(Nodo 6 (Nodo 5 Hoja Hoja)
(Nodo 7 Hoja Hoja))

```

Se puede escribir un algoritmo alternativo de `insertArbol`, que devuelva árboles equilibrados.

Ordenamiento en árboles de búsqueda

Las funciones que se desarrollaron arriba pueden ser usadas en un nuevo algoritmo de ordenamiento. Necesitamos una función más: una función que coloque ordenadamente los elementos de un árbol de búsqueda en una lista. Esta función es la siguiente:

```

labels          :: Arbol a -> [a]
labels Hoja     = []
labels (Nodo x izq der) = labels izq ++ [x] ++ labels der

```

Contrariamente a `insertArbol`, esta función llama recursivamente al subárbol izquierdo y (después) al subárbol derecho. De ese modo, se recorre cada elemento en el árbol. Como el valor `x` se coloca en su lugar equivalente en la lista, el resultado es una lista ordenada (a condición de que el parámetro sea un árbol de búsqueda).

Con las funciones sobre árboles de búsqueda; podemos escribir un algoritmo para ordenar una lista. Primero, hacemos de la lista un árbol de búsqueda (con `listaArbol`), y después transformamos este árbol en una lista ordenada (con `labels`):

```

ordenar :: Ord a => [a] -> [a]
ordenar = labels . listaArbol

```

3.4.3 Usos especiales de definiciones de datos

Además de las construcciones de árboles, las definiciones de datos se puede usar también de otras formas destacables. El mecanismo de tipos de datos es tan universal que se pueden construir cosas que, en otros lenguajes, necesitan muchas veces construcciones especiales.

Tres ejemplos de esto son: tipos finitos, unión de tipos y tipos abstractos.

Tipos finitos

Las funciones constructoras en una definición de tipos de datos pueden no tener ningún parámetro, como vimos con la función constructora `Hoja` del tipo `Arbol`. También se permite que *ninguna* función constructora tenga parámetros. El resultado es un tipo que contiene exactamente tantos elementos como funciones constructoras: un tipo finito. Las funciones constructoras son constantes que indican los elementos. Un ejemplo:

```

data direccion = Norte | Este | Sur | Oeste

```

Se pueden escribir funciones para estos tipos con uso de patrones, por ejemplo:

```

move          :: direccion -> (Int,Int) -> (Int,Int)
move Norte (x,y) = (x,y+1)
move Este  (x,y) = (x+1,y)
move Sur   (x,y) = (x,y-1)
move Oeste (x,y) = (x-1,y)

```

Las ventajas del tipo finito en relación a un código con enteros o caracteres son:

- las definiciones de funciones son más claras porque se puede usar los nombres de los elementos en vez de códigos oscuros;
- el sistema de tipos se queja si se quieren sumar (por ejemplo) las direcciones (si las direcciones están representadas por enteros, no daría un mensaje de error, con todos los efectos negativos).

Los tipos finitos no son nuevos, también el tipo `Bool` puede ser definido de esta manera:

```
data Bool = False | True
```

Esta es la razón por la que se tiene que escribir `False` y `True` con mayúscula. Son funciones constructoras de `Bool`. (Esta definición no está en el prelude. Los booleanos son primitivos. Esto es así porque otras construcciones primitivas ya tienen que conocer los booleanos, por ejemplo, aquéllas con distinción de casos (1).)

Unión de tipos

Todos los elementos de una lista deben ser del mismo tipo. En una tupla se pueden representar valores de diferentes tipos, pero en las tuplas no es variable el número de elementos. Algunas veces se quiere hacer una lista que tenga elementos del tipo entero y del tipo carácter.

Con una definición de datos se puede construir un tipo `IntOrChar` que tenga como elementos enteros y caracteres:

```
data IntOrChar = Ent Int
              | Car Char
```

Con esto se puede hacer una lista mezclada:

```
xs :: [IntOrChar]
xs = [ Ent 1, Car 'a', Ent 2, Ent 3 ]
```

El único precio es que se tiene que marcar cada elemento con una función constructora `Ent` o `Car`. Estas funciones se pueden interpretar como funciones de conversión;

```
Ent :: Int -> IntOrChar
Car :: Char -> IntOrChar
```

cuyo uso se puede comparar con el uso de las funciones primitivas

```
round :: Float -> Int
chr   :: Int   -> Char
```

Tipos abstractos

En el párrafo 3.3.2 se describió una desventaja de las definiciones de tipos. Si se definen dos tipos del mismo modo, por ejemplo:

```
type Fecha = (Int,Int)
type Ratio = (Int,Int)
```

Entonces se pueden usar los dos para la misma cosa. Se pueden usar 'fechas' como 'números racionales', sin que el comprobador de tipos (en inglés, *type-checker*) devuelva errores.

Con las definiciones de datos realmente se pueden hacer nuevos tipos, de modo que no se pueda intercambiar un `Ratio` con cualquier otro tipo `(Int,Int)`. En vez de la definición de tipos podemos escribir la siguiente definición de datos:

```
data Ratio = Rat (Int,Int)
```

Existe entonces solamente una función constructora. Para hacer una fracción con el numerador 3 y el denominador 5, no es suficiente escribir `(3,5)`, sino se tiene que escribir `Rat (3,5)`. Como tipos de unión se puede interpretar a la función `Rat` como función de conversión de `(Int,Int)` a `Ratio`.

Este método se usa mucho para hacer *tipos abstractos*. Un tipo abstracto consiste en una definición de datos y una serie de funciones que se pueden aplicar al tipo definido (en el caso de `Ratio` por ejemplo `qSum`, `qRes`, `qMul` en `qDiv`).

Como nombre de una función constructora, muchas veces se elige el mismo nombre que el tipo. Por ejemplo:

```
data Ratio = Ratio (Int,Int)
```

No es ningún problema; el intérprete no puede equivocarse (`Ratio` significa el tipo si está en una definición de un tipo, por ejemplo detras `::`, en una expresión es la función constructora).

Ejercicios

Ejercicio 3.1

Escriba una función `aprox seno` que, dados dos números `eps` y `x` (el primero mayor que 0, el segundo cualquiera), dé como resultado el número `y` con la propiedad de que

$$|\sin x - y| < \text{eps}$$

Use la siguiente regla matemática:

$$\sin x = \sum_{n \geq 0} [(-1)^n x^{2n+1} / (2n+1)!]$$

Escriba dos veces una definición para `aprox seno`: una vez usando la función `iterate` y otra con `until`.

Ejercicio 3.2

Escriba una función `aprox log` que, dados dos números `eps` y `x` (`eps > 0`, $-1 < x \leq 1$), devuelva un número `y` con:

$$|\log(1+x) - y| < \text{eps}$$

Use en la definición la función `iterate` o `until`, y use también la siguiente regla matemática:

$$\log(1+x) = \sum_{n=1}^{\infty} ((-1)^{n+1} x^n / n)$$

Ejercicio 3.3

Escriba una función `aprox Exp`, que, dados dos números reales `eps` y `x` (`eps > 0`), devuelva un número `y` con $|e^x - y| < \text{eps}$. Use la función `iterate` o la función `until`. Los matemáticos dicen que:

$$e^x = \sum_{n=0}^{\infty} (x^n / n!)$$

Ejercicio 3.4

a. ¿Qué función `f` y qué lista `a` cumplen la siguiente regla?

```
map (+1) . reverse = foldl f a
```

Además de la definición de la función `f` debe dar su tipo.

b. ¿Qué funciones `f` y `g` cumplen la siguiente regla?

```
all . map (>3) = f . filter g
```

Dé también los tipos de las funciones `f` y `g`.

Ejercicio 3.5

Defina una función `esta` que controle si existe cierto elemento en una lista de elementos. Defina la función de las siguientes maneras:

- Tome todos los elementos iguales al elemento buscado y coloque éstos en una lista. Compruebe después si esta lista está vacía o no.
- Haga una nueva lista en la que todos los elementos iguales al elemento buscado sean reemplazados por 1 y los otros elementos por 0. Sume los elementos de la lista resultante y compruebe si el resultado es igual a 0 o no.
- Compruebe para cada elemento si es igual al elemento buscado o no. Después compruebe si uno de estos tests devolvió `True`.

Ejercicio 3.6

Escriba una función `posiciones` que devuelva una lista de índices de las posiciones de un elemento determinado en una lista de elementos.

Por ejemplo:

```
? posiciones 4 [1,4,3,7,4,2]
[2,5]
? posiciones [3,5] [[3,6],[2,5]]
[]
```

Ejercicio 3.7

Escriba una función `ndedc` (*número de elementos distintos creciente*), que dada una lista no decreciente de números, decida cuántos números distintos hay en la lista.

Use el dato de que la lista está ordenada.

Ejercicio 3.8

Escriba una función `nded` (*número de elementos distintos*), que, dada una lista cualquiera de números, devuelva cuántos números distintos existen en la lista.

Una posibilidad de resolver este problema es contar solamente la primera ocurrencia de cada número en la lista.

Ejercicio 3.9

Escriba una función `segmento`, que, dados una lista `xs` y dos números `i` y `j`, devuelva una sublista de `xs` desde el índice `i+1` hasta el índice `j`.

No se puede usar el operador `!!`.

Antes de contestar esta pregunta, se debe especificar que pasa si $j \leq i$, $j > \#xs$ y si $i > \#s$.

Ejercicio 3.10

Escriba una función `esSegmento`, que, dadas dos listas `xs` y `ys` devuelva `True` si `xs` es segmento de `ys`, y `False` si no.

Una lista `xs` es sublista de `ys` cuando `ys = hs ++ xs ++ ts`, con `hs`, `ts` listas de cero o más elementos. Se puede usar la función `segmento` del ejercicio anterior.

Ejercicio 3.11

Escriba una función `scdosa` (sigue concatenando los dos anteriores), que, dadas dos listas `xs` y `ys` del mismo tipo, devuelva una lista infinita de listas, con las siguientes propiedades:

- Los primeros dos elementos son respectivamente `xs` y `ys`.
- Para cada $n > 0$ el $n+2$ -ésimo elemento es la concatenación del n -ésimo elemento con el $n+1$ -ésimo elemento.

Use la función `iterate`.

Ejercicio 3.12

Escriba una función `sssp` (sigue sumando el segmento previo), que, dada una lista finita de números `ns` con un tamaño $k > 0$, devuelva una lista infinita `ms` que cumpla con las siguientes propiedades:

- `ns = take k ms`
- $\forall n \geq k: ms!!(n+1) = (\text{sum} \cdot \text{drop} (n-k) \cdot \text{take } n) ms$

Por ejemplo:

```
sssp [0,0,1] = [0,0,1,1,2,4,7,13,24,44..
```

Use la función `iterate`.

Ejercicio 3.13

Escriba una función `elimDobles`, que, dada una lista (que puede ser infinita), devuelva una nueva lista, con solamente una ocurrencia de cada elemento de la lista original. El problema en este ejercicio es que la lista puede ser infinita. Por eso, no puede usar las funciones `foldr` y `foldl`.

Ejercicio 3.14

Un valor `x` se denomina extremo interno con índice `i` en la lista `xs`, si `i` es un índice con las siguientes propiedades:

- $1 < i < \text{length } xs$
- `xs!!i = x`
- existen una j y una k , con $j < i$ y $k > i$ con `xs!!j \neq x` y `xs!!k \neq x`
- la mayor j ($j < i$) y la menor k ($k > i$) con `xs!!j \neq x` y `xs!!k \neq x` cumplen con la condición que
 - o `xs!!j > x` y `xs!!k > x`
 - o `xs!!j < x` y `xs!!k < x`

Dos extremos internos con índices `i` y `j` en una lista son vecinos si no existe otro extremo con índice `k` y $i < k < j$ o $j < k < i$.

Escriba una función `extremos`, que calcule los extremos internos de una lista.

Use la función `foldl`.

Ejercicio 3.15

Escriba una función `distanciaExtr` que calcule la máxima distancia entre dos extremos vecinos. (Ver el ejercicio 3.14 para la definición de extremos.) Si no existen dos extremos vecinos en la lista, entonces el resultado será 0.

Ejercicio 3.16

Escriba una función recursiva `sc` (*sublistas crecientes*), que, dada una lista, devuelva una lista de listas que existan en todas las sublistas no decrecientes de la lista. Escriba también una definición de `sc` usando `foldr`.

Por ejemplo:

```
? sc [6,1,4,8] = [[], [6], [1], [1,4], [4],
                  [1,4,8], [4,8], [1,8], [6,8], [8]]
```

Ejercicio 3.17

Escriba una función `dividir`, que, dados una lista no decreciente `xs` y un elemento `x`, devuelva una tupla de dos listas (`ys,zs`), con `xs = ys ++ zs`, donde todos los elementos de `ys` sean menores o iguales que `x`, y todos los elementos de `zs` sean mayores que `x`.

Escriba una función `insertar`, que, dados una lista no decreciente `ys` y un elemento `y`, devuelva una lista no decreciente igual a `ys` más el elemento `y` insertado en el lugar correspondiente.

```
dividir :: a -> [a] -> ([a],[a])
```

Ejercicio 3.18

Escriba una función `unico`, que, dada una lista devuelva una lista que contenga exactamente los elementos que se encuentran solamente una vez en la lista dada. Por ejemplo:

```
? unico "Cuales son las letras unicas en esta frase?"
"oicf?"
```

Ejercicio 3.19

a. Escriba una función `segrec` (*segmentos crecientes*), que dada una lista, devuelva una lista de listas que cumpla con las siguientes condiciones:

- la concatenación de los elementos en el resultado devuelve la lista original
- todos los elementos del resultado son listas no decrecientes y tampoco son vacías
- por cada segmento no decreciente `ys` de la lista dada, existe un elemento en el resultado del cual `ys` es un segmento

La definición de `segrec` debe ser en base a `foldl` o `foldr`.

b. Dé también una definición recursiva de `segrec`.

Ejemplo:

```
? segrec [1,2,3,4,2,3,5,6,4,8,3,2]
[[1,2,3,4], [2,3,5,6], [4,8], [3], [2]]
```

Ejercicio 3.20

Escriba una función recursiva `esSubLista`, que, dadas dos listas, devuelva `True` si la segunda lista es una sublista de la primera, y `False` si no. Decimos que `ys` es una sublista de la lista `xs` si existe una lista creciente de números positivos `is`, con `ys = [xs!!i—i ← is]`. Ejemplos:

```
? esSubLista "muchisimo" "uso"
True
? esSubLista [1,4,2,5,7] [4,7]
True
? esSubLista [1,4,2,5,7] [2,1]
True
```

Ejercicio 3.21

Escriba una función `partir`, que, dados un predicado `p` y una lista `xs`, devuelva una tupla de listas (`ys, zs`) en tal forma que `ys` contenga todos los elementos de `xs`, que cumplan con la condición `p`, y en `zs` el resto de los elementos de `xs`. Por ejemplo:

```
? partir digit "a1!,bc4"
("14", "a!,bc")
```

Es posible escribir una definición simple y correcta:

```
partir p xs = (filter p xs, filter (not.p) xs)
```

Pero, en este ejercicio queremos practicar el uso de las funciones `fold`. Entonces, debes dar una definición en base a `foldr` o `foldl`.

Ejercicio 3.22

Escriba las funciones sumar, multiplicar, restar y dividir para números complejos. Un número complejo es de la forma $a + bi$, con a, b números reales, y i un ‘número’ con la propiedad: $i^2 = -1$. Para la función `dividirCompl` puede ser útil primero derivar una formula para $\frac{1}{a+bi}$. Para esto, puedes calcular los valores de x y y en la ecuación $(a + bi) * (x + yi) = (1 + 0i)$.

Ejercicio 3.23

En sistemas de numeración en base k (con k un número entero y $k > 1$), un número puede ser representado por una lista de números, todos menores que k y mayores o iguales a cero.

En el sistema de numeración en base 10 ($k = 10$), la lista `[9,8,4]` representa el número 984 ($9*100+8*10+4*1$).

En el sistema de numeración en base tres ($k = 3$), la lista `[2,0,1]` representa el número 19 ($2*9+0*3+1*1$).

Escriba una función `listaAnumero`, que, dados un número k y una lista `ms` de números m ($0 \leq m < k$), devuelva el número representado por la lista en el sistema de numeración en base 10.

Defina la función en base a `foldl`.

Ejercicio 3.24

Podemos cambiar la representación de números en un sistema de numeración en base k que está descrita en el ejercicio 3.23 por una representación en que está el número al revés. Entonces, en este caso, el número 984 en el sistema de numeración es representado por la lista `[4,8,9]`.

Escriba una función `listaAnumeroR` que haga lo mismo que la función `listaAnumero`, pero ahora con la representación al revés.

Defina la función en base a `foldr`.

Ejercicio 3.25

Escriba una función `multiplicar`, que, dados un número positivo menor que 10 m y una lista de números `ns`, que representa un número n como esta descrito en el ejercicio 3.24, devuelva una lista que represente la multiplicación $n*m$, también según la representación descrita en el ejercicio anterior. Puede suponer que trabajamos en un sistema de numeración en base 10. Ejemplos:

```
? multiplicar 3 [4,8,9]
[2,5,9,2]
? multiplicar 5 [9,9,9,1,4,6]
[5,9,9,9,0,2,3]
```

Una solución podría ser: cambiar el número representado en la lista por un número entero y después multiplicar. Esta solución no se permite, porque, en este caso, no se pueden multiplicar números que sean muy grandes (la

máquina acepta solamente números enteros hasta cierto límite). Por eso, debe aplicar otro sistema de multiplicar, por ejemplo el sistema que consiste en multiplicar número por número y guardar cada vez el resto. En este caso, trabaja con un par de valores: los números del resultado ya calculados y el resto de la última multiplicación. Use `foldr` o `foldl`.

Ejercicio 3.26

Escriba una función `multip` que haga lo mismo que la función `multiplicar` descrita en el ejercicio 3.25, pero ahora multiplique dos listas en la representación especial, y no (como en el ejercicio anterior), un número entero menor que 10 con una lista en la representación especial. Por ejemplo:

```
? multip [1,3] [4,8,9]
[6,3,9,3]
```

Es útil usar la función `multip` y escribir una función `sumar` más. La función `sumar` debe sumar dos números representados en listas como en los anteriores ejercicios.

Ejercicio 3.27

En el párrafo 3.3.4 del texto está definida una función `buscar` que busca un elemento en base a un índice. Escriba una función `buscarArbol` que busque en un árbol con nodos que contengan parejas (índice y valor) un valor en base a un elemento. Elija por sí mismo una estructura de datos para el árbol. Compruebe el tipo de la función `buscarArbol`.

pág. 60

Ejercicio 3.28

Sea el tipo abstracto de datos `Arbol` definido como sigue:

```
Arbol a = Nodo a [ Arbol a ]
```

Entonces, cada nodo tiene un valor de un tipo `a` y una lista de subárboles (del tipo `a`). Esta lista puede ser vacía (en este caso el nodo es una hoja). Escriba una función `construirArbol` que, dado un número `n`, devuelva un árbol del tipo `Arbol Int` con las siguientes propiedades:

- la profundidad del árbol resultante es `n`
- cada nodo que no es hoja tiene tres hijos (subárboles)
- los nodos en el resultado están numerados en preorden, empezando con 0

Por ejemplo:

```
? construirArbol 0
Nodo 0 []

? construirArbol 1
Nodo 0 [Nodo 1 [], Nodo 2 [], Nodo 3 []]

? construirArbol 2
Nodo 0
[ Nodo 1 [ Nodo 2 [], Nodo 3 [], Nodo 4 []],
  Nodo 5 [ Nodo 6 [], Nodo 7 [], Nodo 8 []],
  Nodo 9 [ Nodo 10 [], Nodo 11 [], Nodo 12 []]
]
```

Ejercicio 3.29

Podemos definir un tipo `Arbol`:

```
data Arbol a = Nodo a [Arbol a]
```

Usando este tipo, escribimos la siguiente función:

```
foldt :: (a -> [b] -> b) -> Arbol a -> b
foldt g (Nodo x ts) = g x (map (foldt g) ts)
```

Defina una función g de tal manera que $foldt\ g$ sea una función con el tipo $Arbol\ a \rightarrow [a]$, que, dado un árbol del tipo $Arbol\ a$ devuelva la rama más larga de ese árbol.

Una rama de un árbol $Nodo\ x\ ts$ es una lista de la forma $x:xs$, con xs una rama del elemento ts , y $xs = []$ si $ts = []$.

Ejercicio 3.30

Dé la definición de un tipo algebraico $expr$ cuyas instancias representen expresiones aritméticas, formadas por los operadores $+$ y $*$ y por variables.

Escriba una función $formar$ en la que la representación árbol de una expresión se transforme en otra representación árbol de una expresión equivalente. La nueva representación no permite multiplicaciones de un argumento que sea una suma. Por ejemplo:

```
formar '(a+b)*(a+c)' = '(a*a + b*a)+(a*c + b*c)'
```

Para el cálculo del resultado, se pueden usar las dos leyes distributivas de multiplicación sobre suma:

```
(a+b)*c = (a*c) + (b*c)
a*(b+c) = (a*b) + (a*c)
```

Ejercicio 3.31

Una matriz es una lista de listas del tipo $[[a]]$. Podemos declarar un tipo abstracto de datos para matrices de enteros:

```
data Mat = Mat [[Int]]
```

Todos los elementos de la lista deben tener el mismo tamaño. Una matriz cuadrada con todos sus elementos, cero o uno, puede ser interpretada como una matriz de conexiones. Si el j -ésimo elemento en la i -ésima lista es uno, significa que existe una conexión del nodo i hacia el nodo j . Un cero indica que no existe una conexión.

- Escriba una función `tieneSalidas`, que, dados una matriz de conexiones m y un nodo i ($1 \leq i \leq \text{length } m$), devuelva `True` si existen caminos desde i a uno o más nodos, y `False` si no.
- Escriba una función `borrar`, que, dados una matriz m y un nodo i , devuelva una matriz igual a m menos las conexiones desde y hacia el nodo i .
- Escriba una función `tieneCiclo`, que, dado un nodo m , devuelva `True`, si existe un ciclo en la matrix m , y `False` si no. Existe un ciclo en una matrix si se puede llegar, desde un nodo i , a través otros nodos, al nodo i .

Ejercicio 3.32

Escriba una función recursiva $numDifAsc$ que, dada una lista de números ascendente, calcule cuántos diferentes elementos están en la lista. Por ejemplo:

```
numDifAsc [1,2,2,5,5,5,5,7,8] = 5
```

Se debe calcular el resultado directamente, no se debe calcular primero la lista sin elementos dobles y después calcular su tamaño.

Ejercicio 3.33

Escriba una función $numDif$ que, dada una lista de números, calcule cuántos diferentes elementos están en la lista. No se debe ordenar la lista y después usar la función $numDifAsc$ o algo así.

Capítulo 4

Algoritmos sobre listas

4.1 Funciones combinatorias

4.1.1 Segmentos y sublistas

En esta sección se presentan algunas funciones combinatorias sobre listas. El resultado es una lista de listas. No se usan las propiedades específicas de los elementos de la lista. Lo único que las funciones combinatorias hacen es eliminar elementos, intercambiar elementos, o contar elementos.

En este y en el siguiente párrafo se definen algunas funciones combinatorias. Dado que no usan las propiedades de los elementos de su lista parámetro, son funciones polimórficas:

```
inits, tails, segs ::      [a] -> [[a]]
subs, perms      ::      [a] -> [[a]]
combs            :: Int -> [a] -> [[a]]
```

Para explicar el funcionamiento de estas funciones, se muestran los resultados de su aplicación a la lista [1,2,3,4]

inits	tails	segs	subs	perms	combs 2	combs 3
[]	[1,2,3,4]	[]	[]	[1,2,3,4]	[1,2]	[1,2,3]
[1]	[2,3,4]	[4]	[4]	[2,1,3,4]	[1,3]	[1,2,4]
[1,2]	[3,4]	[3]	[3]	[2,3,1,4]	[1,4]	[1,3,4]
[1,2,3]	[4]	[3,4]	[3,4]	[2,3,4,1]	[2,3]	[2,3,4]
[1,2,3,4]	[]	[2]	[2]	[1,3,2,4]	[2,4]	
		[2,3]	[2,4]	[3,1,2,4]	[3,4]	
		[2,3,4]	[2,3]	[3,2,1,4]		
		[1]	[2,3,4]	[3,2,4,1]		
		[1,2]	[1]	[1,3,4,2]		
		[1,2,3]	[1,4]	[3,1,4,2]		
		[1,2,3,4]	[1,3]	[3,4,1,2]		
			[1,3,4]	[3,4,2,1]		
			[1,2]	[1,2,4,3]		
			[1,2,4]	[2,1,4,3]		
			[1,2,3]	[2,4,1,3]		
			[1,2,3,4]	[2,4,3,1]		
				[1,4,2,3]		
				(7 otros)		

Por tanto, el significado de las seis funciones es:

- **inits** devuelve todas las partes iniciales (*segmentos iniciales*) de una lista. La lista vacía también se considera parte inicial.
- **tails** devuelve todas las partes finales (*segmentos finales*) de una lista. También la lista vacía es parte final.
- **segs** devuelve todos los segmentos de una lista: los iniciales, los intermedios y los finales.
- **subs** devuelve todas las sublistas de una lista. No hace falta que los elementos sean consecutivos, como en los segmentos, sólo que se mantenga su orden relativo. Existen, por tanto, más sublistas que segmentos.

- `perms` devuelve todas las *permutaciones* de una lista. Una permutación de una lista contiene los mismos elementos, pero posiblemente en otro orden.
- `combs n` devuelve todas las *combinaciones de n elementos*, es decir, todas las maneras de seleccionar `n` elementos de una lista. El orden es el mismo que en la lista original.

Estas funciones se pueden definir de forma recursiva. En la definición se distingue entre el caso con la lista vacía (`[]`) y las listas con uno o más elementos (`(x:xs)`). En el caso `(x:xs)`, la función se llama recursivamente sobre la lista `xs`.

Existe una manera sencilla para encontrar una definición recursiva de una función `f`: observar en una lista ejemplo `(x:xs)` cuál es el resultado de la llamada `f xs`, y tratar de aumentar el resultado hasta el resultado de `f (x:xs)`.

inits

En la descripción de la función `inits` se decidió que la lista vacía también es un segmento de comienzo. Por tanto, la lista vacía sólo tiene un elemento inicial: la propia lista vacía. La definición de `inits` en el caso 'lista vacía' es:

```
inits [] = [ [] ]
```

Para el caso `(x:xs)` vemos a un ejemplo con la lista `[1,2,3,4]`.

```
inits [1,2,3,4] = [ [], [1], [1,2], [1,2,3], [1,2,3,4] ]
inits [2,3,4]   = [ [], [2], [2,3], [2,3,4] ]
```

Este ejemplo muestra que el segundo hasta el quinto elemento de `inits [1,2,3,4]` coinciden con los elementos de `inits [2,3,4]`, pero con el elemento 1 como cabeza. Se tiene que añadir a este conjunto de listas la lista vacía.

Una definición es:

```
inits []      = [ [] ]
inits (x:xs) = [] : map (x:) (inits xs)
```

tails

La lista vacía sólo tiene un segmento de fin: la propia lista vacía. Por tanto, el resultado de `tails []` es una lista con la lista vacía como único elemento.

Para tener una idea de una definición de `tails (x:xs)`, veamos un ejemplo con el parámetro `[1,2,3,4]`:

```
tails [1,2,3,4] = [ [1,2,3,4], [2,3,4], [3,4], [4], [] ]
tails [2,3,4]   = [ [2,3,4], [3,4], [4], [] ]
```

El segundo hasta el quinto elemento son iguales al resultado de la llamada recursiva. Lo único que se debe hacer es añadir como primer elemento la lista pasada como parámetro `([1,2,3,4])`.

Usando esta idea, se puede escribir una definición:

```
tails []      = [ [] ]
tails (x:xs) = (x:xs) : tails xs
```

segs

El único segmento de la lista vacía es la lista vacía. Por tanto, el resultado de `segs []` es, como en los casos de `inits` y `tails`, la lista con la lista vacía como único elemento.

Para encontrar una definición de `segs (x:xs)` usamos otra vez un ejemplo con la lista `[1,2,3,4]`:


```

segs [1,2,3,4] = [ [], [4], [3], [3,4], [2], [2,3], [2,3,4], [1], [1,2], [1,2,3], [1,2,3,4] ]
segs [2,3,4]  = [ [], [4], [3], [3,4], [2], [2,3], [2,3,4] ]

```

Si se colocan los elementos siguiendo el mismo orden, se puede ver que los primeros siete elementos coinciden con la llamada recursiva. En la segunda parte del resultado (las listas que empiezan con un 1), se pueden reconocer los segmentos iniciales de [1,2,3,4] (se ha excluido la lista vacía, porque ya estaba en el resultado).

Por tanto, se puede tomar como definición de `segs`:

```

segs []      = [ [] ]
segs (x:xs) = segs xs ++ tail (inits (x:xs))

```

Otra manera para quitar la lista vacía de `inits` es:

```

segs []      = [ [] ]
segs (x:xs) = segs xs ++ map (x:) (inits xs)

```

subs

La lista vacía es la única sublistas de la lista vacía. Para la definición de `subs (x:xs)` veamos el ejemplo:

```

subs [1,2,3,4] = [ [1,2,3,4], [1,2,3], [1,2,4], [1,2], [1,3,4], [1,3], [1,4], [1],
                  [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], [] ]
subs [2,3,4]  = [ [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], [] ]

```

El número de elementos de `subs (x:xs)` (en el ejemplo 16) es exactamente el doble que el número de elementos de la llamada recursiva `subs xs`. La segunda parte del resultado completo es exactamente igual al resultado de la llamada. También en la primera parte se pueden reconocer las 8 listas, pero allí con un 1 como cabeza.

Por tanto, la definición puede ser:

```

subs []      = [ [] ]
subs (x:xs) = map (x:) (subs xs) ++ subs xs

```

Se llama recursivamente a la función dos veces con el mismo parámetro. No es eficiente: mejor es hacer la llamada solamente una vez, y usar el resultado dos veces. Eso es realmente mejor, porque para calcular `subs xs` se llama otra vez dos veces recursivamente, y en estas llamadas, otra vez, ... Una definición más eficiente es:

```

subs []      = [ [] ]
subs (x:xs) = map (x:) subsxs ++ subsxs
              where subsxs = subs xs

```

4.1.2 Permutaciones y combinaciones

perms

Una *permutación* de una lista es una lista con los mismos elementos, pero posiblemente en otro orden. Se puede definir muy bien la lista de todas las permutaciones de una lista con una función recursiva.

La lista vacía tiene una permutación: la lista vacía. Más interesante es el caso de la lista con uno o más elementos (`x:xs`). Veamos un ejemplo:

```
perms [1,2,3,4] = [ [1,2,3,4] , [2,1,3,4] , [2,3,1,4] , [2,3,4,1]
                  , [1,3,2,4] , [3,1,2,4] , [3,2,1,4] , [3,2,4,1]
                  , [1,3,4,2] , [3,1,4,2] , [3,4,1,2] , [3,4,2,1]
                  , [1,2,4,3] , [2,1,4,3] , [2,4,1,3] , [2,4,3,1]
                  , [1,4,2,3] , [4,1,2,3] , [4,2,1,3] , [4,2,3,1]
                  , [1,4,3,2] , [4,1,3,2] , [4,3,1,2] , [4,3,2,1] ]
perms [2,3,4]   = [ [2,3,4] , [3,2,4] , [3,4,2] , [2,4,3] , [4,2,3] , [4,3,2] ]
```

El número de permutaciones aumenta rápidamente: una lista de cuatro elementos tiene cuatro veces más permutaciones que una lista de tres elementos. En este ejemplo, es un poco más difícil reconocer el resultado de la llamada recursiva. Pero, si se agrupan los 24 elementos en 6 grupos de 4 elementos, se puede ver la relación con la llamada recursiva. En cada grupo hay listas con los mismos valores que los de un resultado recursivo. Los nuevos elementos se intercalan en todos los lugares posibles.

Por ejemplo, las listas del tercer grupo $[[1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1]]$ contienen todos los elementos $[3,4,2]$, a los cuales se añadió el elemento 1 en las cuatro diferentes posiciones (primera, segunda, tercera y cuarta posición).

Para colocar un elemento en todas las posiciones posibles en una lista, se puede escribir una función. También esta función puede ser definida recursivamente:

```
entre      :: a -> [a] -> [[a]]
entre e [] = [ [e] ]
entre e (y:ys) = (e:y:ys) : map (y:) (entre e ys)
```

Se instancia esta función parcialmente con x en la definición de `perms (x:xs)`, y se aplica a todos los elementos del resultado de la llamada recursiva. En el ejemplo, resulta esto en una lista con seis listas de cuatro sublistas. Sin embargo, la intención es que el resultado sea una lista grande con 24 listas pequeñas. Por tanto, a la lista de listas, se tiene que aplicar la función `concat`.

Todo eso resulta en la siguiente definición de la función `perms`:

```
perms []      = [ [] ]
perms (x:xs) = concat (map (entre x) (perms xs))
              where entre e []      = [ [e] ]
                    entre e (y:ys) = (e:y:ys) : map (y:) (entre e ys)
```

combs

El último ejemplo de una función combinatoria es la función `combs`. Esta función tiene, además de una lista, también un número como parámetro:

```
combs :: Int -> [a] -> [[a]]
```

La idea es que en el resultado de `combs n xs` estén todas las sublistas de `xs` de longitud `n`. Por tanto, esta función puede ser definida simplemente por:

```
combs n xs = filter bueno (subs xs)
              where bueno xs = length xs == n
```

Esta definición no es muy eficiente. El número de sublistas es muchas veces muy grande, entonces calcular `subs xs` tarda mucho tiempo, cuando muchas sublistas son desechadas por `filter`. Es mejor definir `combs` directamente, sin usar `subs`.

En la definición de `combs` se distinguen dos casos para el parámetro entero: 0 y $n+1$. En el caso $n+1$ se distinguen también dos casos para el parámetro lista. Por tanto, la definición tiene la siguiente forma:

```
combs 0      xs      = ...
combs (n+1) []      = ...
combs (n+1) (x:xs) = ...
```

Se ven estos tres casos caso por caso:

- Al elegir cero elementos de una lista se obtiene siempre el mismo resultado: la lista vacía. Por tanto, el resultado de `combs 0 xs` es la lista con la lista vacía como único elemento. No importa si la lista `xs` está vacía o no.
- El patrón `n+1` significa '1 o más'. Elegir uno o más elementos de la lista vacía es imposible. El resultado de `combs (n+1) []`, es por tanto la lista vacía. Cuidado: aquí tenemos una *lista vacía de soluciones* y no una *lista vacía como única solución* como en el caso anterior. La diferencia es importante.
- Elegir `n+1` elementos de una lista `x:xs` es posible si se pueden elegir `n` elementos de `xs`. Se puede dividir la solución en dos grupos: listas que contienen el elemento `x`, y listas que no contienen `x`.
 - Para las listas con `x`, se tienen que elegir de `xs` `n` elementos más. `x` es la cabeza de los resultados.
 - Para las listas que no contienen `x`, se tienen que elegir los `n+1` elementos de `xs`.

Para los dos casos se puede llamar a `combs` recursivamente. Los resultados pueden ser combinados con `++`.

Con estas observaciones se tiene la siguiente definición de `combs`:

```
combs 0      xs      = [ [] ]
combs (n+1) []      = [ ]
combs (n+1) (x:xs) = map (x:) (combs n xs) ++ combs (n+1) xs
```

No se pueden combinar las dos llamadas recursivas, como en la definición de `subs`, porque aquí tienen las dos llamadas diferentes parámetros.

4.1.3 La notación @

La definición de `tails` es la siguiente:

```
tails []      = [ [] ]
tails (x:xs) = (x:xs) : tails xs
```

En la segunda regla, se divide la lista `x:xs` en una cabeza `x` y una cola `xs`. Se usa la cola en la llamada recursiva, pero por lo demás se junta la cabeza y la cola otra vez (`x:xs`). Esto es demasiado trabajo, ya que (`x:xs`) es el propio parámetro.

Otra definición de `tails` podría ser::

```
tails [] = [ [] ]
tails xs = xs : tails (tail xs)
```

Ahora no es necesario construir de nuevo la lista parámetro porque no se la divide. Pero ahora se tiene que usar `tail` para calcular la cola.

Lo ideal sería combinar lo bueno de estas dos definiciones. Por tanto, se quiere tener el parámetro completo, y con división entre la cabeza y la cola. Para esta situación existe una notación especial. Delante de un patrón se puede escribir un nombre que representa el parámetro completo. Entre los dos se intercala el símbolo `@`.

Usando esta construcción resulta la siguiente definición de `tails`:

```
tails [] = [ [] ]
tails lista@(x:xs) = lista : tails xs
```

Aunque se puede usar el símbolo @ en símbolos de operadores, se ha reservado un solo @ para esta construcción.

También podemos usar el patrón @ en definiciones de funciones que ya han aparecido. Por ejemplo, en la función `dropWhile`:

```
dropWhile p [] = []
dropWhile p ys@(x:xs)
  | p x = dropWhile p xs
  | otherwise = ys
```

Ésta es la manera en que `dropWhile` está definida en el preludio.

Ejercicios

Ejercicio 4.1

Defina una función recursiva `lcs` (*listas con suma*), que dado un número natural positivo `s`, devuelva una lista de listas que cumpla con las siguientes condiciones:

- todos los elementos son crecientes y ninguno de los elementos contiene un número repetido
- la suma de los números en cada elemento es exactamente el número dado `s`

Por ejemplo:

```
lcs 6 = [[1,2,3], [1,5], [2,4], [6]]
lcs 8 = [[1,2,5], [1,3,4], [1,7], [2,6], [3,5], [8]]
```

Ejercicio 4.2

Escriba una función `bins` que calcula todos los números binarios de longitud `n`, para cualquier número entero `n`. Por ejemplo:

```
? bins 3
["111", "110", "101", "100"]
```

```
bins :: Int -> [[Char]]
```

Use recursión. Los números que empiezan con un 0 no sirven. (Sugerencia: primero calcule todos los números, también aquéllos que empiezan por 0.)

Ejercicio 4.3

Escriba una función recursiva `menosUno`, que calcule todas las sublistas de una lista que se forman a partir de una lista original eliminando un elemento. Por ejemplo:

```
? menosUno [1,2,3,4,5]
[[2,3,4,5], [1,3,4,5], [1,2,4,5],
 [1,2,3,5], [1,2,3,4]]
```

Apéndice A

Literatura relacionada con la Programación Funcional

- Richard Bird y Philip Wadler: *Introduction to Functional Programming*. Prentice-Hall, 1988.
- A.J.T. Davie: *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.
- Ian Hoyer: *Functional Programming with Miranda*. Pitman, 1991.
- Hudak y Fasel: 'A Gentle Introduction to Haskell'. *ACM Sigplan Notices* **27**, 5 (may 1992) pp.T1–T53.
- Hudak, Peyton-Jones, Wadler et al.: 'Report on the Programming Language Haskell: a non-strict purely functional language, version 1.2.' *ACM Sigplan Notices* **27**, 5 (may 1992) pp.R1–R164.
- Mark P. Jones: *Introduction to Gofer 2.20*. Oxford Programming Research Group, 1992.
- Field y Harrison: *Functional Programming*. Addison-Wesley, 1989.
- Simon Peyton-Jones: *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- Simon Peyton-Jones y David Lester: *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992.
- Rinus Plasmeijer y Marko van Eekelen: *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

Índice de Materias

- ++, 42
- ., 31
- /=, 42
- <, 42
- <=, 41
- ==, 41
- >, 42
- >=, 42
- &&, 53

- abcFormule, 12, 20
- abs, 8, 12, 60
 - definición, 12
- and, 10, 28, 29, 37, 42, 44
 - definición, 28, 29
- apóstrofo, 23
- aproxlog, 68
- aproxseno, 68
- Arbol (tipo), 63, 64, 66
- arccos
 - definición, 36, 37
- arcsin
 - definición, 36, 37
- 'as'-patrón, 79
- asociación
 - de :, 40

- bastanteAcept, 35
- bins, 80
- bisiesto, 33
 - definición, 33
- Bool (tipo), 17–20, 49, 67
- borrar, 74
- buscar, 60, 65
 - definición, 60

- cadena, 48
- Car, 67
- Char (tipo), 18, 49, 50, 54
- chr, 50, 51, 67
- Church, Alonzo, 1
- class
 - Eq, 21, 41, 60, 64
 - Num, 20, 21
 - Ord, 20, 41, 47, 50, 64
- comb, 5, 6, 20
 - definición, 5, 11
- combinación, 76, 78–79
- combs, 78, 79
 - definición, 78, 79
- combs n, 76
- comentarios, 16
- comilla inversa, 23
- comp, 5
- comparación
 - de listas, 42
- Complejo (tipo), 58
- concat, 42, 78
 - definición, 42
- concatenación, 42
- constante, 6
- copy, 53, 54
 - definición, 53
- cos, 3, 19, 37
- cuadrado, 11, 21, 27
 - definición, 11, 27
- curry, 61
- Curry, Haskell, 1
 - definición de datos, 62
 - definición de tipo, 58
 - definición inductiva, 15
 - definición recursiva, 14
 - definiciones locales, 12
- desde, 52, 53
 - definición, 52
- después, 30
- despues, 30
 - definición, 30
- dia, 32, 33
 - definición, 32, 33
- diag, 21
- diaSemana, 32, 44
 - definición, 44
- diff, 34, 36, 39
 - definición, 34
- digitChar, 51
 - definición, 51
- digitValue, 50
 - definición, 50
- distancia, 58

- distanciaExtr, 70
- div, 25
- dividir, 22, 71
- dividirCompl, 72
- divisible, 31–33, 55, 59
 - definición, 31
- divisores, 31–33, 53, 59
 - definición, 31
- drop, 43, 46, 48, 57
 - definición, 43
- dropWhile, 46, 80
 - definición, 46, 80
- e
 - definición, 11
- elem, 44, 46, 64, 65
 - definición, 44, 46
- elem'
 - definición, 64
- elemArbol, 64
 - definición, 64
- elemBoom, 65
- elimDobles, 70
- Ent, 67
- entre, 78
 - definición, 78
- enumFrom, 53
- enumFromTo, 41
 - definición, 41
- Eq (class), 21, 41, 60, 64
- esCero
 - definición, 11, 27
- esSegmento, 69
- esSubLista, 71
- esta, 69
- estructura de control, 29
- evaluación perezosa, 52
- evaluación voraz, 53
- even, 10, 11, 19
- exp, 9, 11, 37
- extremos, 70
- f, 68, 69
- fac, 4–7, 11, 25
 - definición, 4, 7, 11, 14
- filter, 28, 32, 41, 45, 46, 55, 56, 59, 78
 - definición, 28, 45
- flexDiff, 34
 - definición, 34
- Float (tipo), 18, 20, 31, 33, 34, 37, 49, 57, 59
- fold, 72
- foldl, 29, 46, 47, 70–72
 - definición, 46
- foldl', 7
- foldr, 29, 37, 41, 44–48, 70–72
 - definición, 29, 45
- for (palabra res.), 29
- formulaABC, 12, 15, 20
 - definición, 11
- formulaABC', 12, 35
 - definición, 12
- fromInteger, 9
- fst
 - definición, 56
- función
 - combinatoria, 75–80
 - como operador, 23
 - sobre listas, 41
- función combinatoria, 75–80
- función de orden superior, 29
- funciones constructoras, 62
- funciones polimórficas, 19
- funciones primitivas, 7
- g, 69
- gcd, 8, 59, 60
 - definición, 59
- Gofer, 2
- guardas, 12
- Haskell, 2
- head, 14, 15, 19, 22, 41, 43, 56
 - definición, 14, 42
- Hoja, 62, 63, 66
- Hope, 2
- id, 20
 - definición, 19
- igualdad
 - sobre listas, 41
- impar, 30
 - definición, 11
- inc, 26
 - definición, 27
- infix (palabra res.), 25
- infixl (palabra res.), 25
- init, 43
 - definición, 43
- inits, 75–77
 - definición, 76
- insert, 47, 48, 65
 - definición, 47
- insertar, 71
- insertArbol, 65, 66
 - definición, 65
- Int (tipo), 17–20, 26, 31, 49, 54
- intérprete, 2
- intervalos
 - notación, 41
- IntOrChar (tipo), 67

- intString, 54, 55, 60
 - definición, 55
- inverso, 37, 39
 - definición, 37
- is...
 - definición, 50
- isort, 48, 65
 - definición, 47
- isspace, 51
- iterate, 54–56, 68, 70
 - definición, 54
- keyword
 - for, 29
 - infix, 25
 - infixl, 25
 - type, 58
 - where, 12, 16, 17, 30, 34
 - while, 29
- labels, 66
 - definición, 66
- last, 43, 59
 - definición, 43
- layn, 51, 52
- lcs, 80
- length, 4, 10, 15, 17, 19, 41, 44, 48, 53, 62, 63, 78
 - definición, 15
- lines, 51
- Lisp, 2
- lista, 39–56
 - comparar, 42
 - concatenación, 42
 - enumeración, 39, 40
 - igualdad, 41
 - notación de intervalos, 41
 - singleton, 40
 - tipo, 39
 - vacía, 40
- lista vacía, 40
- listaAarbol, 65, 66
 - definición, 65
- listaAnumero, 72
- listaAnumeroR, 72
- ln
 - definición, 37
- log, 9
- máximo común divisor, 59
- map, 11, 20, 22, 26–29, 41, 44–46, 51, 52, 56, 61, 64, 78
 - definición, 28, 45
- mas, 25, 26
- McCarthy, John, 2
- mcd, 59
 - definición, 59
- mejorar, 35
- merge, 48
 - definición, 48
- meses, 33
 - definición, 33
- Miranda, 2
- ML, 2
- mod, 25
- multip, 72, 73
- multiplicar
 - fracciones, 60
 - números, 8
- multiplicar, 72
- multiplo, 55
- números racionales, 59
- nded, 69
- ndedc, 69
- negativo
 - definición, 11
- Neuman, John von, 1
- Nodo, 62, 63
- noDoble, 22
- not, 10, 11, 30, 44
- notElem, 44
 - definición, 44
- null, 10
- Num (class), 20, 21
- numeroDia, 32, 33
 - definición, 32
- numerosPrimos, 32, 33
 - definición, 32, 56
- operador, 6
 - como función, 23
- operator, 23
 - ++, 42
 - ., 31
 - /=, 42
 - <, 42
 - <=, 41
 - ==, 41
 - >, 42
 - >=, 42
 - &&, 53
- or, 44, 45, 64
 - definición, 45
- Ord (class), 20, 41, 47, 50, 64
- ord, 50, 51
- Ordenar, 47
- ordenar
 - definición, 66

- palabras reservadas, 7
- par, 14, 30
 - definición, 14
- parámetros actuales, 13
- parámetros formales, 13
- pareja, 56
- partir, 71
- perms, 75, 76, 78
 - definición, 78
- permutación, 76–78
- pi
 - definición, 6, 11
- polimórficas
 - funciones, 19
- polimórfico
 - tipo, 40
- polimorfismo, 19
- posiciones, 69
- positivo
 - definición, 11
- precedencia, 23–24
- predefinidas, 7
- primo, 32, 33, 53, 55
 - definición, 32
- primPlusInt, 7
 - definición, 7
- product, 4, 7, 11, 28, 29, 37
 - definición, 28, 29
- Punto (tipo), 58
- puntoCero, 36, 37
 - definición, 36

- qDiv, 68
- qMul, 68
- qRes, 68
- qSum, 68

- raiz, 35–37
 - definición, 35, 36
- Rat, 68
- Ratio (tipo), 67, 68
- ratioString, 60
- rem, 25, 31, 33, 60
- repeat, 53, 54
 - definición, 53
- reverse, 4, 44, 54, 55
- round, 9, 67

- sc, 70
- scdosa, 70
- Scheme, 2
- Schönfinkel, M., 1
- segcrec, 71
- segmento, 75–77
- segmento, 69
 - segmento final, 75, 76
 - segmento inicial, 75, 76
- segs, 75, 77
 - definición, 77
- semanaDia, 33
- signum, 8, 12, 59
 - definición, 12
- simplificar
 - fracción, 59
- simplificar, 59, 60
 - definición, 59
- sin, 3, 9, 19, 27, 34, 37
- singleton list, 40
- snd
 - definición, 56
- sort, 4, 48, 50
- splitAt, 57
 - definición, 57
- sqrt, 3, 9–13, 34, 35, 57
- sssp, 70
- sublista, 75, 77
- subs, 75, 77–79
 - definición, 77
- sum, 3, 6, 10, 11, 18, 19, 28, 29, 33, 37, 53
 - definición, 6, 19, 28, 29
- suma, 15, 18, 41
 - definición, 15
- sumar
 - fracciones, 60
 - números, 8
- sumar, 73
- sums, 4, 10

- t (tipo), 41
- tail, 14, 15, 41, 43, 77, 79
 - definición, 14, 42
- tails, 75, 76, 79, 80
 - definición, 76, 79
- take, 10, 33, 43, 46, 48, 54, 57
 - definición, 33, 43
- takeWhile, 46, 52, 54, 55
 - definición, 46
- tamano
 - definición, 63
- tan, 19
- tieneCiclo, 74
- tieneSalidas, 74
- tipo de datos, 61
- tipo polimórfico, 19, 40
- toUpper
 - definición, 51
- tupla, 56–61
- Turing, Alan, 1
- type

Arbol, 63, 64, 66
Bool, 17–20, 49, 67
Char, 18, 49, 50, 54
Complejo, 58
Float, 18, 20, 31, 33, 34, 37, 49, 57, 59
Int, 17–20, 26, 31, 49, 54
IntOrChar, 67
Punto, 58
Ratio, 67, 68
t, 41
type (palabra res.), 58

uncurry, 61
 definición, 61
unico, 71
unlines, 51
until, 29, 30, 35–37, 48, 54, 68
 definición, 29
unwords, 51

variable de tipo, 19

where (palabra res.), 12, 16, 17, 30, 34
while (palabra res.), 29
words, 51

zip, 60, 61
 definición, 60, 61
zipWith, 61
 definición, 61