Introducción al Lenguaje Funcional Haskell

Apuntes de la Cátedra Principios de Lenguajes de Programación

Departamento de Teoría de la Computación
Facultad de Informática
Universidad Nacional del Comahue

Marcelo Paulo Amaolo

June 3, 2021

Índice

1 Introducción

Haskell es un lenguaje de programación que permite el desarrollo de sistemas para ejecutar en una computadora. En particular es un lenguaje polimórfico, currificado, fuertemente tipado, con tipos estáticos, perezoso, puramente funcional, y en general, un poco diferente a los lenguajes que están acostumbrados los estudiantes de la carrera.

El lenguaje se llama Haskell, en honor a Haskell Brooks Curry, cuyo trabajo en el área de la lógica matemática sirvió como base para la construcción del paradigma funcional. Haskell está basado en la teoría lógica de las funciones computables llamada Cálculo Lambda por ello utiliza esa letra griega como su logo.

Figure 1: Haskell Brooks Curry

El Cálculo Lambda es un lenguaje formal capaz de expresar funciones computables arbitrarias. En combinación con los tipos, establece una manera compacta de programar utilizando funciones y al mismo tiempo permitir pruebas matemáticas. Por esta razón, Haskell podría verse como una implementación elegante del cálculo lambda, como herramienta de uso para la lógica y la matemática, cuyas construcciones se parecen mucho a las especificaciones, y permiten la manipulación elegante de estructuras de datos infinitas.

El Lenguaje Funcional Haskell, entre otras características, ofrece:

- Un enfoque distinto que incrementa considerablemente la productividad de los programadores.
- Código más corto, más claro y más fácil de mantener
- Menos error y mayor confiabilidad
- Tiempos de desarollo más cortos

Haskell puede definirse como un lenguaje de propósito general, apropiado para una gran variedad de aplicaciones. Es particularmente apropiado para los programas con altos requerimientos de modificación y mantenimiento. Es conocido que la mayoría del tiempo de vida de un software se dedica a la especificación, diseño y mantenimiento, y no a la programación.

Los lenguajes funcionales son extremadamente convenientes para escribir especificaciones que pueden ejecutarse rápidamente, lo que permite además una fase de testeo y debugging muy ágil. Estas características permiten incluso, muchas veces, que estas especificaciones se transformen en los primeros prototipos del programa final. Los programas escritos en lenguajes funcionales son fáciles de mantener, porque el código es más corto, y porque el control riguroso de los efectos colaterales elimina muchísimos problemas imprevistos, que son dificultosos de detectar en otros lenguajes.

Haskell se desarrolló en las últimas décadas como un estándar de la programación funcional perezosa, un estilo de programación en el cual los argumentos o parámetros son evaluados sólo cuando es necesario su valor.

2 Características del Lenguaje Haskell

En su página web oficial, presenta como significadas, las siguientes características:

2.1 Tipado Estático Fuerte

Cada expresión escrita en el lenguale Haskell tiene un tipo, que se determina en tiempo de compilación[1]. Cuando se desea invocar a una función, todos los tipos que componen la aplicación de la invocación deben coincidir, es decir, los tipos de las expresiones en la invocación deben coincidir precisamente con los tipos de los argumentos que espera la función, si no coinciden, el compilador "rechaza" el programa. De esta manera en Haskell, los tipos no sólo garantizan que no ocurran errores en la programación, sino que se transforman en una garantía y una manera de expresar cómo se construyen los programas.

2.2 Puramente funcional

Las funciones en Haskell son funciones en un sentido matemático, es decir, puramente funcionales. No existen sentencias o instrucciones, sólo se utilizan expresiones que se evalúan, y que no pueden modificar los valores de los objetos de datos (variables o argumentos) fuera de su alcance léxico, ya que no tienen efectos colaterales. Las funciones, dada la misma entrada, siempre devuelven la misma información.

2.3 Inferencia de Tipo

Si bien es una práctica muy recomendada cuando se escribe código en Haskell, no siempre es necesario describir explícitamente cuáles son los tipos de todos los recursos que se utilizan en una funcion: el lenguaje puede inferir automáticamente los tipos a partir de las operaciones y los operandos del código, que ayudan en la programación y que brindan seguridad en la ejecución.

2.4 Concurrencia

Debido a las características de cómo maneja Haskell los eventos, el lenguaje se adapta muy bien a la programación concurrente. Es más, su implementación más utilizada, GHC[2], ya viene con un recolector de basura paralelo de alto rendimiento, y una librería para procesos livianos con una buena cantidad de primitivas y abstracciones de concurrencia.

2.5 Evaluación perezosa

Las funciones en Haskell no evalúan sus argumentos sino hasta que se necesitan, esto implica que es posible combinar los programas sin dificultad, con la posibilidad de escribir constructores de control (tales como if/else) sólo escribiendo funciones normales. La pureza del código hace sencillo fusionar las cadenas de funciones, logrando con ello mejoras en el rendimiento. Uno de

[1] O antes de la ejecución si estamos trabajando con un intérprete
[2] https://www.haskell.org/ghc/

los beneficios de la evaluación perezosa consiste en la posibilidad de manipular estructuras de datos potencialmente infinitas, que se construyen según las necesidades de evaluación.

2.6 Disponibilidad de paquetes

Finalmente, vale destacar que la comunidad ha hecho y hace un gran contribución de herramientas de código abierto, con un gran número de paquetes disponibles en servidores públicos.

2.7 Algunos usos actuales

Se muestran algunas organizaciones que utilizan este lenguaje y de qué manera lo hacen:

1. Facebook usa Haskell para combatir el *spam*. Los ingenieros de Facebook han elegido a Haskell por su rendimiento, soporte de desarrollo interactivo y otras características que hacen de Haskell la mejor opción para su proyecto Sigma.
2. NVIDIA utiliza Haskell para el desarrollo de backend de sus GPU.
3. Microsoft usa Haskell en su proyecto Bond. Bond es un marco multiplataforma para trabajar con datos con esquemas. Este marco se usa ampliamente en servicios de gran escala.
4. J.P. Morgan, el banco más grande de los Estados Unidos, tiene un grupo Haskell en su equipo de Desarrollo de Nuevos Productos.
5. IBM, AT&T y Bank of America también utilizan Haskell y soluciones de programación funcional para sus proyectos.
6. Proyecto Cardano, que utiliza la tecnología blockchain para la circulación de criptomonedas y una plataforma para el trabajo con los contratos inteligentes.

3 Formatos

Este apunte contiene un gran cantidad de códigos ejemplo, y para ayudar al lector, los códigos y las salidas están adecuadamente indicadas. Por ejemplo, para diferenciar el código fuente del resto del texto del apunte, se distingue de esta manera:

```
El código fuente tiene esta apariencia
y esta también.
```

Usualmente, se distingue además la interacción entre el programador y el sistema operativo y/o el *shell* interactivo.

```
Las interacciones tienen esta apariencia.
```

4 Cómo comenzar

Existen muchas herramientas accesibles y la mayoría de ellas libres y gratis, para trabajar con el lenguaje funcional Haskell. Si bien una alternativa es ejecutar los códigos escritos en el lenguaje en un servicio *online*, se aconseja trabajar con herramientas locales. Entre las más utilizadas se encuentra *GHC*.

GHC - Glasgow Haskell Compiler es tanto un intérprete como un compilador que produce programas ejecutables en código nativo para varios tipos de arquitecturas. También puede compilar generando programas en lenguaje C. Es la herramientas más utilizada para ejecutar Haskell y por ello se ha transformado en su estándar de facto. GHC fue escrito en Haskell (con extensiones), los programas ejecutables que genera son muy rápidos. En su entorno interpretado es un poco más lento para cargar, pero permite definiciones de funciones en el entorno. Además de satisfacer el estándar Haskell 98, actualmente ha implementado el estándar Haskell 2010 y cuenta con una activa comunidad de desarrolladores.

Para instalar GHC en una computadora con el sistema Linux basados en Debian, use: [3]

```
$ sudo apt install ghc
```

Para este curso de Lenguajes de Programación la herramienta utilizada para trabajar con Haskell es indistinta, si bien se sugiere la instalación de la Plataforma Haskell (Haskell Platform, http://www.haskell.org/platform/, básicamente, Haskell con todas las librerías y módulos más habitualmente usados), con el siguiente comando:

```
$ sudo apt install haskell-platform
```

Para otros sistemas operativos puede consultar en "https://www.haskell.org/platform"

También existen otras alternativas para poder ejecutar los programas que pueden trabajarse para el curso:

- *Hugs* es un intérprete, por lo que no es posible utilizarlo para compilar archivos, pero su naturaleza interactiva permite probar y debuggear programas con facilidad y agilidad. Es muy rápido para cargar los archivos pero es algo lento para ejecutarlos. Implementa el estándar Haskell 98 y la mayoría de sus extensiones, está escrito en C y trabaja prácticamente en todas las plataformas y sistemas operativos. Si bien es una excelente herramienta para aprendizaje, y más que suficiente para un curso introductorio del lenguaje, *actualmente está sin mantenimiento.* En Linux Debian es parte de la distribución y se puede instalar directamente de sus repositorios[4], y también existen versiones más antiguas que funcionan en Windows[5].
- *IDEs online*: diversos sitios proveen la provisión de compiladores y editores *online* para el lenguaje Haskell[6].

[3] Puede trabajar con el FI-Debian, sistema operativo desarrollado en la Facultad para uso estudiantil. Similar para otras distribuciones Linux.

[4] Con el comando "`sudo apt install hugs`"

[5] Ver en https://www.haskell.org/hugs/pages/downloading-May2006.htm

[6] A la fecha de edición del presente, se destacan, entre otros, https://rextester.com/l/haskell_online_compiler,

4.1 GHC

GHC toma un *script* escrito en Haskell (normalmente un programa con una extensión *.hs*) y lo compila para generar un ejecutable.

Ejecutar el compilador es muy simple. Si existiera un programa, escrito con un editor, en un archivo fuente llamado "*Principal.hs*", se puede compilar ejecutando la siguiente línea de comandos:

```
$ ghc -make Principal.hs -o principal
```

La opción "-make" le dice al compilador GHC que se trata de un programa, y no una librería y que se indica que debe construir el programa y todos los módulos dependientes. "*Principal.hs*" establece el nombre del archivo a compilar, y la opción "*-o principal*" significa que la salida del compilador se guardará en un archivo llamado "*principal*". Nótese que en el caso de que el compilador se ejecute en Windows, la salida (según el sistema operativo en uso) será "*-o principal.exe*" que le indicará a Windows que se trata de un archivo ejecutable. En todos los casos, se asume que la ejecución del archivo ejecutable se realizará invocando en la línea de comandos a "*principal*".

En el modo interactivo puede cargarse (con la sentencia *:l*) y trabajar interactivamente con el programa, llamando a las funciones definidas en el script, y los resultados se muestra inmediatamente. Para aprenderlo, es mucho más fácil y rápido en forma interactiva que compilar todo el tiempo los programas que serán (casi siempre) modificados. El modo interactivo se invoca con el comando:

```
$ ghci
```

en la línea de comandos del sistema en el que se haya instalado, con un resultado similar al siguiente:

```
$ ghci
GHCi, version 8.6.5:  http://www.haskell.org/ghc/ :?  for help
Prelude>
```

Si ha definido algunas funciones en un archivo (por ejemplo el archivo **misfunciones.hs**), se pueden cargar estas funciones escribiendo **:l misfunciones** y luego practicar con él (siempre que el archivo .hs se encuentra en la misma carpeta en la que ha sido invocado a ghci). Si cambiara el archivo .hs, entonces puede volver a cargar la nueva versión escribiendo **:l misfunciones** o con **:r** (recargar el script actual).

La forma habitual de trabajar es crear algo en un archivo .hs, cargarlo en el modo interactivo, probar la ejecución, cambiar el archivo .hs y volver a cargarlo una y otra vez, hasta alcanzar el trabajo buscado.

Hay varios comandos que pueden ejecutarse, por ejemplo

```
:quit
```

que cerrará la sesión actual de trabajo, mientras que

```
:?
```

devolverá una lista de los comandos disponibles.

https://paiza.io/es/languages/haskell, etc. Consulte en un buscador.

Particularmente, algunos comandos útiles son:

Comando	Descripción
:load <nombrearchivo>	Carga los módulos de los archivos especificados.
:reload	Repite el último comando de carga.
:edit <nombrearchivo>	Edita un archivo
:edit	Edita el ultimo módulo cargado.
:module <nombremodulo>	Selecciona un módulo para evaluar expresiones
:main <argumentos>	Ejecuta el main con argumentos.
:run <nombre> <args>	Similar a :main.
<expresion>	Evalua la expresión
:type <expresion>	Imprime el tipo de una expresión.
:?	Muestra una lista de comandos
:info <nombre>	Describe al objeto nombrado.
:!<comando>	Shell para comandos
:cd <dir>	Cambia de directorio al directorio <dir>.
:quit	Salir del interprete Hugs.

Los comandos siempre empiezan con ':' excepto si se trata de una expresión a evaluar.

Part I

Funciones y Tipos de Datos Básicos

5 Funciones

5.1 Funciones y Cálculo

Los programas funcionan en líneas generales de manera similar independientemente del lenguaje en el que hayan sido escritos. Los programas toman datos de entrada y los transforman en datos de salida. Un programa funcional describe directamente *como* los datos son analizados, manipulados y reemsamblados. La forma más común de escribir una transformación es una función.

Una **función** es algo que nos devolverá una salida dependiendo de una o mas entradas. Se va a usar el término **resultado** para la salida y los términos **argumentos o parámetros** para las entradas.

El cálculo es el proceso mediante el cual se toma una expresión (numérica o no) y se trata de hallar un valor asociado a ella.

5.2 Programación Funcional

La programación funcional esta basada en la idea del cálculo. Nosotros definimos las funciones a ser utilizadas y el intérprete se encargará de buscar el valor asociado a una expresión que haga uso de estas funciones.

Un programa funcional (***script*** en Haskell), consiste de un número de definiciones de funciones

y otros valores. Una implementación de un lenguaje funcional calculará el valor de cualquier expresión usando las funciones definidas.

5.3 Tipos

En Haskell cada objeto tienen un tipo, y el sistema puede chequear por ejemplo que las funciones sean aplicadas solo a objetos del tipo de dato de la entrada. Estas y otras **restricciones de tipo** son aplicadas a las expresiones y definiciones que nosotros escribimos y el sistema Haskell nos garantiza que se cumplan.

5.4 Scripts en Haskell

El siguiente es un ***script*** de ejemplo:

```
1  ------------------------------------------------
2  --                  Ejemplo.hs                  --
3  ------------------------------------------------
4  answer :: Int              -- una constante entera
5  answer = 42
6  newline :: Char
7  newline = '\n'
8  yes :: Bool                -- La respuesta y es es representada
9  yes = True                 -- por el valor Booleano True
10 greater :: Bool
11 greater = (answer > 71)
12
13 ------------------------------------------------
14 --          El cuadrado de un número           --
15 ------------------------------------------------
16 square :: Int -> Int
17 square x = x * x
18
19 ------------------------------------------------
20 --             Son todos iguales               --
21 ------------------------------------------------
22 allEqual :: Int -> Int -> Int -> Bool
23 allEqual n m p = (n == m) && (m == p)
24
25 ------------------------------------------------
26 --           El máximo de dos enteros           --
27 ------------------------------------------------
28 maxi :: Int -> Int -> Int
29 maxi n m
30    | n > m     = n
31    | otherwise = m
```

En el ejemplo se pueden ver varias *definiciones*. Cada definición asociada a un nombre, como ***answer*** o ***square*** con un valor, el cual puede ser un tipo específico. Los comentarios son insertados con el símbolo "−−", y permiten texto hasta fin de la línea. Para los comentarios de longitud arbitraria, puede utilizarse "{−" y "−}".

La notación "::" se debe leer *es de tipo*. El símbolo "=" es usado para armar definiciones.

La declaración de tipos de ***square*** (línea 16) establece que es una función de *enteros* a *enteros* ya que "***Int->Int***" es el tipo de funciones de tipo entero a entero (Int a Int).

La definición misma es una **ecuación**. Esta da un resultado, ***x*x*** (ver línea 17); ***x*** es una **variable**, la cual representa a la entrada de la función. La salida es definida, usando esta entrada, en el lado derecho de la ecuación. Para llamar a una función con los argumentos ***a***, ***b*** y ***c***, escribimos ***f*** seguido de los argumentos, es decir

```
f a b c
```

.

A esta forma de escribir la aplicación de función se la denomina *yuxtaposición.*

En el ejemplo vemos que ***allEqual*** (línea 22) es una función que toma tres números enteros y devuelve un booleano. Formalmente tienen el tipo ***Int -> Int -> Int -> Bool***.

En la definición de la función ***maxi*** (línea 28) se observa que se realiza un control (línea 30). Los **controles** o **centinelas** son expresiones lógicas que se especifican entre la barra **"|"** y el símbolo ***"="*** determinando cual de las opciones será elegida. A toda la definición se le da el nombre de **ecuación condicional**.

6 Cálculo en Haskell

Se ha visto que la programación funcional es similar a la evaluación o el cálculo en la aritmética, excepto que en este caso usamos nuestras propias funciones y otras definiciones. En el ejemplo, la evaluación de ***answer*** es clara, ***answer*** tiene asociado el valor ***42***, por lo tanto en la evaluación podremos reemplazarla por ***42***.

Las definiciones de funciones pueden ser interpretadas de forma similar. Por ejemplo la definición de ***allEqual*** es dada para todos los valores de ***n m p***. Nosotros obtendremos valores particulares de ***allEqual*** reemplazando o substituyendo las variables con valores en la ecuación de definición.

Entonces, calculando la función inicialmente se calcula como:

```
1 allEqual 2 3 3
2 =(2 == 3) && (3 == 3)
```

```
1 allEqual 5 5 5
2 =(5 == 5) && (5 == 5)
```

Se continúan los cálculos trabajando sobre las componentes lógicas combinándolas usando &&, que a continuación se presentan en dos columnas para poder comparar su salida:

```
allEqual  2 3 3                    allEqual 5 5 5
= (2 == 3) && (3 == 3)             = (5 == 5) && (5 == 5)
= False && True                    = True && True
= False                            = True
```

Resumiendo, en el entorno interactivo:

```
ghci> allEqual 2 3 3
False
ghci> allEqual 5 5 5
True
```

En el caso de la definición de ***maxi*** vemos que hay dos casos, dados por las cláusulas de la ecuación condicional. Para saber que valores utilizar verificaremos los centinelas uno por uno desde arriba hacia abajo hasta que uno de ellos nos da ***True***. Cuando escribamos el cálculo se usará *'??'* para señalar el cálculo del centinela.

Por ejemplo:

```
maxi 3 1                           maxi 3 4
?? 3 >= 1 = True                   ?? 3 >= 4 = False
= 3                                ?? otherwise = True
                                   = 4
```

Para casos más complejos, se pueden utilizar ayudas gráficas para realizar la traza, por ejemplo subrayando que partes van a ser evaluadas en el siguiente paso.

Así:

```
allEqual (maxi 1 5) 5 (maxi 4 2)
= ( ( maxi 1 5 ) == 5 ) && ( 5 == ( maxi 4 2 ) )
  ?? 4 >= 2 = True
= ( ( maxi 1 5 ) == 5 ) && ( 5 == 4 ) )
  ?? 1 >= 5 = False
  ?? otherwise = True
= ( 5 == 5 ) && ( 5 == 4 ))
= True && False
= False
```

7 Sintaxis

La sintaxis de un lenguaje describe a los programas bien formados de dicho lenguaje. Se van a describir las características básicas de la sintaxis del Haskell y se va a hacer hincapié en las características poco comunes que pueden resultar extrañas a primera vista.

7.1 Definiciones y esquema

Un ***script*** contiene una serie de definiciones, una después de la otra. ¿Cómo se puede saber donde termina una definición y donde comienza la siguiente?. En Pascal el separador de sentencias es ";". En Haskell, el esquema del programa es usado para decir donde una definición termina y empieza la siguiente.

Una definiciones terminada por la primera pieza de texto que este al mismo nivel o a la izquierda del comienzo de la definición. Cuando escribimos una definición, su primer carácter abre la caja la cual va a contener a la definición. Por ejemplo:

```
square  x = x * x
↓
```

cualquier cosa que sea escrita en la caja forma parte de la definición

```
square  x = x * x
↓  +x
      +2
```

hasta que algo es encontrado en la misma línea que la "s" o bien hacia la izquierda. Esto cierra la caja.

```
square  x = x * x
   +x
      +2
```

```
cube x = ...
```

Debido a esta regla las definiciones del mismo nivel tendrán la misma indentación. En nuestros ***scripts*** siempre vamos a escribir las definiciones de más alto nivel comenzando en el lado izquierdo de la página. Esta regla es llamada regla de *off/side* ya que es una reminiscencia de la idea de estar *off-side* en el fútbol. La regla también se aplica a las ecuaciones condicionales (como ***maxi***) que consistan de más de una cláusula.

Haskell tiene un mecanismo explícito para indicar el fin de una definición. Es el símbolo "**;**" al igual que en Pascal; en realidad, internamente, Haskell utiliza el "**;**".

7.2 Formato Recomendado

La regla off-side permite varios estilos diferentes de formatos. Para las definiciones de cualquier tamaño, utilizamos la forma

```
fun v1 v2 ... vn
  | g1          = e1
  | g2          = e2
 ...
  | otherwise = en     (o gn   = en)
```

para una **ecuación condicional** para un número n de cláusulas. En este formato, cada cláusula comienza con una nueva línea, y los guardianes y resultados están alineados. Es importante distinguir además que por convención especificamos el tipo de la función definida, que debe coincidir con los lados derechos de la cláusula. Si alguna de las expresiones $\mathbf{e}_i$ o guardianes $\mathbf{g}_i$ son particularmente largos, entonces el guardián puede aparece en una o más líneas por si mismo.

```
fun v1 v2 ... vn
  | un guardián muy largo
    que puede ocupar más de una línea
      = una expresión muy larga
        que puede ocupar más de una línea
 ...
  | otherwise = en     (o gn   = en)
```

7.3 Convenciones de nombres

Los nombres o identificadores son usados para nombrar tipos, funciones o variables. Los identificadores en Haskell tienen la misma forma que en Pascal, comienzan con una letra (mayúscula u minúscula) seguidos de letras, dígitos y el símbolo "_" (underscore) además de las letras acentuadas.

Los nombres utilizados en definición de valores comienzan con minúscula al igual que las variables y las variables de tipos que se verán más adelante. Los nombres de tipos comienzan con mayúsculas. Ejemplos:

```
type Pair = (Int,Int)

sumaDos:: Pair -> Int
sumaDos (primero,segundo) = primero + segundo

```

7.4 Operadores

La asociatividad se aplica para resolver la ambigüedad que se produce cuando hay que evaluar una expresión que contiene varias ocurrencias de un mismo operando. La precedencia se aplica para resolver la ambigüedad que se produce cuando hay que evaluar una expresión que contiene varios operadores pero no se sabe cual evaluar primero.

Por ejemplo:

- La expresión ***4−2−1*** ¿cómo se evalúa? Las dos posibles opciones son ***(4−2)−1*** (asociatividad a izquierda) y ***4−(2−1)*** asociatividad por derecha. Obviamente en este caso la opción correcta es la primera.
- La expresión ***2+3×4*** se puede evaluar como ***(2+3)×4*** o como ***2+(3×4)***, en este caso la operación × se dice que tiene una precedencia más alta que +, por lo tanto se evalúa primero.

La tabla siguiente muestra las relaciones de asociatividad y de precedencia entre operadores Haskell:

Precedencia	Asociativa a izq.	Sin asociatividad	Asociativa a der.
9	! !! //		
8			** ^ ^^
7	* / div mod rem quot		
6	+ −	: +	
5		\\	: %% ++
4		/= > < <= > >= elem notElem	
3		&&	
2			||
1		: =	
0			$$

7.5 Operadores y Funciones

Los operadores pueden ser convertidos a funciones, los cuales preceden sus argumentos, encerrando al operador entre paréntesis.

Por ejemplo, ***(+)*** como una función del tipo ***Int -> Int -> Int***, de forma que:

```
(+) 1 2
```

es equivalente a

```
1 + 2
```

De la misma manera se pueden convertir funciones en operadores encerrando el nombre de la función entre comillas simples[7], por ejemplo:

[7] Es la comilla que tienen los teclados asociados al acento grave ‘

```
2 `maxi` 3
```

es equivalente a

```
maxi 2 3
```

7.6 Operadores definidos por el usuario

El lenguaje Haskell permite al usuario definir operadores infijos de la misma manera que las funciones. Los nombres de los operadores sólo pueden definirse a partir del siguiente conjunto de caracteres:

```
¡ # $ % ^ * + . / < = > ¿ @ \ ^ | :
```

además de los símbolos Unicode. El nombre de un operador no puede comenzar con dos puntos (":"). Por ejemplo, para definir el operador "&&&", como la función mínimo de enteros, se puede notar:

```
1 (&&&) :: Int -> Int -> Int
2 x &&& y
3   | x > y        = y
4   | otherwise    = x
```

De hecho la asociatividad y la precedencia también pueden especificarse [8]

7.7 Patrones en las definiciones

Las definiciones que hemos visto hasta ahora han consistido de una ecuación simple o una ecuación condicional conteniendo una o más cláusulas. Al comienzo de la definición tenemos:

fun v_1 v_2 ... v_n

El nombre de función ***fun*** es aplicado a las variables v_1 a v_n.

Es posible utilizar más de una ecuación condicional para definir una función. De esta forma podemos describir el comportamiento de la función cuando es aplicada a un patrón más que a un conjunto de variables. Los patrones simples son variables y constantes. Por ejemplo:

```
1 totalVentas 0 = ventas 0
2 totalVentas n = totalVentas (n-1) + ventas n
```

[8] Ver Bibliografía asociada

Para hallar el valor de una función para una entrada en particular, usamos la primera ecuación para la cual la entrada coincida con el patrón sobre el lado izquierdo. Un argumento a coincide con un patrón p si

- p es una constante y a es igual a p.
- p es una variable

Existe una tercera forma de patrón que se va a introducir ahora. Supongamos que queremos chequear si un entero es cero o no. Podemos escribir :

```
1 esCero :: Int -> Bool
2 esCero 0 = True
3 esCero _ = False
```

En la ecuación final estamos usando un *patrón comodín*, el cual coincide con cualquier valor. No se puede utilizar este valor en el lado derecho de la ecuación. Por lo tanto una nueva cláusula para la definición de coincidencia de patrones sería:

- a coincide con el patrón _ para cualquier valor de a

El siguiente ejemplo demuestra el uso de la coincidencia de patrones (*pattern matching*) para la definición de funciones:

```
1 fib :: Int -> Int
2 fib 0 = 0
3 fib 1 = 1
4 fib n
5    | n > 1      = fib (n-2) + fib (n-1)
6    | otherwise  = 0
```

8 Tipos de Datos Simple

Haskell es un lenguaje con tipos. Esto significa que cada expresión (o término) del lenguaje tiene un tipo asociado. Se puede verificar el tipo de una expresión utilizando el comando "`:type`". Pero también se puede instruir a que el entorno interactivo (tal Hugs o GHCi) imprima el tipo de cada resultado calculado, ingresando el comando "`:set +t`". Intente verificar su comportamiento cuando realice cálculos básicos aritméticos.

Los tipos básicos de Haskell son:

- **Int** e **Integer** se utilizan para representar a los números enteros. Los elementos de **Integer** son enteros sin límite.
- **Float** y **Double** representan los números reales de punto flotante. Los elementos de **Double** tienen mayor precisión

- **Bool** es el tipo para los valores enteros Verdadero (True) y Falso (False).
- **Char** permite representar a los caracteres.

8.1 Enteros

El tipo ***Int*** es similar al tipo entero clásico. Los objetos de este tipo son representados en una cantidad fija de memoria. El tipo ***Integer*** representa a enteros pero en una cantidad variable de espacio de memoria dependiendo del valor que se deba almacenar. Las operaciones son las clásicas: $+$, $*$, $-$, $\hat{}$, ***div***.

div puede ser usada en forma prefija, por ejemplo:

```
div 14 3
```

da como resultado

```
4
```

pero también se puede usar en forma infija, por ejemplo [9]

```
14 `div` 3
```

La función ***mov*** se comporta de la misma manera. Otras operaciones son ***abs*** y ***negate***, esta última cambia el signo de su argumento. En general todas las funciones de dos argumentos pueden ser utilizadas en forma infija teniendo la precaución de poner el nombre de la función entre comillas simples.

Los operadores relacionales son `>`, `>=`, `==`, `/=` (distinto), `<=`, `<` . Una prueba en el intérprete explicará directamente su comportamiento:

```
ghci> 6 == 6
True
ghci> 1 == 0
False
ghci> 6 /= 6
False
ghci> 6 /= 4
True
ghci> 6 >= 4
True
ghci> 6 >= 6
True
ghci> 6 <= 4
False
ghci> "hola" == "hola"
True
```

[9] Las comillas no son las comillas simples, sino las que resultan de un acento grave, notadas `.

8.2 Booleanos

Los tipos booleanos en Haskell son llamados ***Bool***. Los operadores booleanos provistos por el lenguaje son :

Operador	Significado
&&	y (and)
||	o (or)
not	no (not)

Un ejemplo:

```
1 orExcl :: Bool -> Bool -> Bool
2 orExcl x y = ( x || y ) && not ( x && y)
```

Es posible utilizar las constantes ***True*** y ***False*** como argumentos. Por ejemplo:

```
1 orExcl True x  = not x
2 orExcl False x = x
```

8.3 Caracteres y Cadenas

Los caracteres y cadenas son útiles en el momento de comunicarse con las computadoras mediante las entradas por teclado y las salidas por monitor. Los caracteres son representados en Haskell mediante el tipo ***Char***. Los caracteres Individuales son escritos entre comillas simples. Algunos caracteres especiales son:

Caracter Especial	Significado
Tabulado	'\t'
Nueva línea	'\n'
Barra a izquierda	'\\'
Comillas simples	'\"
Comillas dobles	'\"'

Se pueden utilizar los códigos ASCII para escribir caracteres de la siguiente manera: $=$ ***'\nn'*** donde **nn** es el número correspondiente al código ASCII al que se hace referencia (por ejemplo, '\69' representa la letra 'E').

Existen funciones de conversión[10]:

```
1 toEnum :: Int -> Char
2 fromEnum  ::  Char -> Int
```

[10] toEnum 69::Char 'E'

Pero también están definidas las conocidas funciones ***ord*** y ***chr***. Por ejemplo

```
offset = ord 'A' - ord 'a'

mayusculas :: Char -> Char
mayusculas ch = chr ( ord ch + offset)
```

Las cadenas de caracteres pertenecen al tipo ***String*** y están delimitadas por dobles comillas:

```
"esta es una cadena"
""
"\99a\116"
"gorila\nhipopotamo\ngarza"
"1\t23\t456"
```

Para imprimir una cadena se utiliza

```
putStr :: String -> IO()
```

La función ***print*** simplemente imprime la cadena sin intentar interpretarla. Por ejemplo, se muestran las salidas para ambas funciones:

```
ghci> putStr "gorila\nhipopotamo\ngarza"
gorila
hipopotamo
garza
ghci> print "gorila\nhipopotamo\ngarza"
"gorila\nhipopotamo\ngarza"
```

Haskell permite nombres de tipos. Estos nombres de tipos se denominan sinónimos. En el prelude de Haskell está definido el sinónimo ***String*** de la siguiente manera:

```
Type String = [Char]
```

Lo cual expresa el hecho que una cadena (String) es una lista de caracteres[11].

Las operación de concatenación de cadenas se especifica con *++*. Por ejemplo

```
ghci > "123" ++ "abc"
"123abc"
```

8.4 Números de Punto Flotante

Los números de punto flotante son representados en Haskell por el tipo ***Float***. Internamente Haskell representa los números de punto flotante con una cantidad de espacio fija, lo que tiene

[11] El tipo de datos Lista se verá más adelante

como efecto que la representación de las fracciones no siempre es exacta. Es posible mejorar la precisión de la representación con el tipo ***Double***, o para una representación completa de las fracciones, el tipo ***Rational*** definido a partir del tipo ***Integer***.

Algunos números de punto flotante literales en Haskell pueden ser números decimales, tales como

```
0.31426
-23.12
5623.234
2342.0
```

Haskell también permite literales de números de punto flotante en notación científica. Se expresan a continuación algunos ejemplos con su correspondiente valor

```
231.61e7        (231.61 × 10^7 = 2.316.100.000)
231.6e-2        (231.61 × 10^-2 = 2,3161)
-3.412e03       (-3.412 × 10^3 = -3412)
```

Las operaciones definidas sobre estos números son las operaciones clásicas, representadas en la siguiente tabla:

Operación o Función	Signatura	Descripción
$+ - *$	`Float -> Float -> Float`	Suma Resta Multiplicación
`/`	`Float -> Float -> Float`	División fraccional
`^`	`Float -> Int -> Float`	Exponenciación con exponentes naturales.
`**`	`Float -> Float -> Float`	Exponenciación con exponentes reales
`== /= < > >= <=`	`Float -> Float -> Bool`	Operadores de orden e igualdad
`abs, acos, asin, atan, cos, sin, tan, log, negate, exp, sqrt`	`Float -> Float`	Absoluto, Arco/Valor del Coseno/Seno/Tangente, Logaritmo, Cambia Signo, Exponente, Raíz Cuadrada
`logBase`	`Float -> Float -> Float`	Logaritmo de base arbitraria (primer argumento)

8.5 Convertir valores entre tipos

La clase de tipos **Integral**[12], contiene a los tipos de valores completos, es decir, sin sus fracciones. Los tipos integrales más importantes son: "`Int`" e "`Integer`". La clase de tipos **RealFrac** contiene a los tipos de valores reales, y sus tipos más importantes son: "`Float`", "`Rational`" y "`Double`". Se destaca también la clase de tipos **Num** que contiene los tipos numéricos, entre ellos a "`Int`", "`Integer`", "`Float`" y "`Double`".

Es posible convertir los números de diferentes tipos utilizando funciones para ello. Se destacan las siguientes funciones:

[12]Una clase de tipos agrupa un conjunto de tipos, no es un Tipo dado. No tiene nada que ver con el concepto de Clases de la programación orientada a objetos.

Función	Signatura	Descripción
`fromIntegral`	`(Integral a, Num b) =>` `a -> b`	Convierte de Enteros (Int o Integer) a Números (Int, Integer, Float, Double, etc.)
`fromInteger`	`Num a =>` `Integer -> a`	Convierte de Enteros a Números, usado con Reales
`ceiling, floor, round`	`(RealFrac a, Integral b) =>` `a -> b`	Convierte un número Real a un número Entero por redondeo superior, redondeo inferior o al entero más próximo
toInteger	`Integral a =>` `a -> Integer`	Convierte un número entero al tipo Integer
float2Double	`Float -> Double`	Convierte de número real tipo Float al tipo Double.
double2Float	`Double -> Float`	Convierte de número real tipo Double al tipo Float.
`read`	`String -> Float`	Convierte una cadena que representación un número float a su valor numérico

9 Literales y funciones sobrecargadas

De los tipos de datos simples vistos hasta el momento, se pueden realizar comparaciones por igualdad utilizando el mismo símbolo, el "`==`", por ejemplo para enteros y booleanos, si bien se está referenciando a dos operaciones distintas. En realidad, el símbolo "`==`" se utilizará para realizar la comparación de igualdad para cualquier tipo, con lo que se crea la clase de tipos **Eq**[13], que podríamos notar como "`a`", con lo que ese operador responderá a la signatura:

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

en la que "`a`" representa cualquier tipo que soporta comparación por igualdad. La utilización del mismo símbolo o nombre para representar más de una operación diferente, se llama "**sobrecarga**".

De la misma manera, si consideramos los números `4` y `2`, pertenecen ambos a la clase de tipos **Num** que incluye a los tipos **Int**, **Integer**, **Float** y **Double**. De esta manera los literales numéricos están sobrecargados, así como otras operaciones numéricas, tales como la suma y el producto. Haskell es capaz de gestionar la sobrecarga de varias maneras que iremos presentando en este apunte.

La signatura de "`==`" se expresa como dos valores que pertenecen al mismo tipo, cualquiera sea el tipo, siempre que pertenezcan a la clase de tipos Eq. Así, por ejemplo ¿es posible responder a la pregunta si `2` es igual a `True`?:

[13] Que incluye a todos los tipos excepto IO()

```
> 2 == True
<interactive>:1:1:  error:
• No instance for (Num Bool) arising from the literal ‘2’
• In the first argument of ‘(==)’, namely ‘2’
In the expression:  2 == True
In an equation for ‘it’:  it = 2 == True
```

La respuesta correcta es no, porque la pregunta no tiene sentido. Es imposible comparar un número con algo que no sea un número, o un booleano con algo que no sea booleano. Haskell no entiende de esa manera, y al momento de ejecución nos muestra un error que indica exactamente eso. El mensaje informa que como hay un número en la expresión al lado izquierdo del ==, se espera algún tipo de número del lado derecho, pero como el valor que se pasó como argumento no es un número sino un booleano, entonces la prueba de igualdad no puede ejecutarse.

9.1 La función Show

Los valores de varios tipos de datos pueden ser transformados a ***String*** por medio de la función `show`. Por ejemplo:

```
ghci > show 5
"5"
ghci > show 'a'
"'a'"
ghci > show "Hola Mundo"
"\"Hola Mundo\""
```

En la primera llamada a `show` se muestra cómo el número `5` del tipo `Int` se transforma en una cadena, observe las comillas alrededor del número `5`. En la segunda llamada, el argumento de `show` es "comilla simple", la letra `a` y "comilla simple" (la notación de una expresión del tipo `Char`), que transforma integramente, por lo que la salida de la función Show es una cadena con tres elementos, justamente comilla-letra-comilla.

En la última línea del ejemplo, se observa lo mismo para cadena. La expresión es "comilla doble", la cadena `Hola Mundo` y "comilla doble". La barra aparece porque los caracteres comillas forman parte de la cadena y Haskell agrega una barra para identificar caracteres especiales en una cadena.

9.2 La función Read

La función `read` opera como la inversa de la función `show`, generando un tipo a partir de un `String`, según lo esperado. Por ejemplo:

```
ghci > read "5"
5
ghci > read "5" + 3
8
ghci > read "12"::Double
12.0
ghci > read "True"::Bool
True
```

En los primeros dos casos, Haskell infiere el tipo esperado según los operadores. En los otros casos, el programador determina el tipo esperado.

Part II

Notación y métodos para definir funciones

10 Alcance de los identificadores en Haskell

Todas las definiciones de más alto nivel en un código Haskell tienen como alcance a todo el código. Es decir estas pueden ser utilizadas en todas las definiciones que contenga ese código. Es posible, como parte de una ecuación condicional, establecer definiciones que tengan alcance sólo en forma local a la función o a otro objeto que se defina. Estas definiciones son escritas después de la palabra clave ***where***, que presentamos a partir de ejemplos, antes de especificar formalmente la "visibilidad" de las definiciones, cálculos, etc. locales.

Tomando como ejemplo una función que entrega la suma de los cuadrados de dos números, tenemos:

```
1 sumaCuadrados :: Int -> Int -> Int
```

el resultado de la función será la suma de dos valores, que llamamos ***cuadrN*** and ***cuadrM***, de forma tal que

```
1 sumaCuadrados n m
2   = cuadrN + cuadrM
```

donde la definición de estos valores puede darse dentro de la cláusula ***where*** que sigue a la ecuación:

```
1 sumaCuadrados n m
2   = cuadrN + cuadrM
3    where
4     cuadrN = n*n
5     cuadrM = m*m
```

Las definiciones dadas por una cláusula ***where*** no son visibles en todo el código. Sólo son visibles en las ecuaciones (condicionales) en las cuales aparece. Lo mismo ocurre con las variables (argumentos) en la definición de una función donde su alcance es toda la ecuación en la cual ellas aparecen. Veamos el siguiente ejemplo:

```
1  maximo :: Int -> Int -> Int
2  maximo n m
3   | cuadrN > cuadrM = cuadrN
4   | otherwise       = cuadrM
5    where
6      cuadrN = cuad n
7      cuadrM = cuad m
8      cuad :: Int -> Int
9      cuad z = z * z
```

Puede observarse la visibilidad de las variables utilizadas, el alcance se limita a la ecuación en la que aparecen, por ejemplo el alcance de las definiciones de ***cuadrN***, ***cuadrM*** y ***cuad*** se extiende de la línea 3 a la 9 mientras que el alcance de la variable ***z*** sólo a la línea 9.

10.1 La cláusula let

Para las definiciones en Haskell, también se puede utilizar la cláusula `let` que funciona de manera similar a `where`, pero que también puede ser utilizada en forma interactiva, y permite definir declaraciones locales. Por ejemplo, para calcular las raíces de un ecuación cuadrática, podemos definirla como:

```
1  raices Int -> Int -> Int -> (Int,Int)
2  raices a b c =
3    let det = sqrt ( b * b - 4 * a * c )
4    in ( (-b + det ) / (2 * a) ,
5         (-b - det ) / (2 * a) )
```

La principal diferencia entre `let` y `where` radica en que `let` puede ser utilizado en forma local en cualquier expresión, mientras que `where` tiene alcance sólo para la definición de funciones. Obsérvese que mientras que esta expresión es inválida:

```
ghci > print (1 + (2 * i + 1 where i = 10))
ERROR - Syntax error in expression (unexpected keyword "where")
```

sin embargo es válida la expresión

```
ghci > print (1 + (let i = 10 in 2 * i + 1))
22
```

10.2 Para tener en cuenta

- Las variables que aparecen el lado izquierdo de la definición de función (***n*** y ***m*** en este caso) pueden ser usadas en las definiciones locales
- Las definiciones locales pueden ser usadas antes de que estas sean definidas.
- Las definiciones locales pueden ser utilizadas en resultados y guardas como también en otras definiciones locales.

Es posible tener en un programa dos definiciones o variables con el mismo nombre. En el ejemplo siguiente, la variable ***x*** aparece dos veces. ¿Qué definición tomará en cada caso? La más local en la que es utilizada. Vemos el ejemplo:

```
1  maxCuadr :: Int -> Int -> Int
2  maxCuadr x y
3    | cuad x > cuad y = cuad x
4    | otherwise       = cuad y
5     where
6      cuad :: Int -> Int
7      cuad x = x * x
```

En este ejemplo podemos considerar un "corte" del cuadro interior frente al exterior visto para los alcances, ya que el alcance de ***x*** (externo) será "excluido" (de la línea 7) en la definición de ***cuad***, donde se redefine el ***x*** interno.

Cuando una definición está contenida dentro de otra, el mejor consejo es utilizar variables y nombres diferentes, a menos de que la naturaleza de la solución exija utilizar el nombre más de una vez. Es incluso posible tener múltiples definiciones del mismo nombre en el mismo nivel: una de ellas debe estar escondida si hubiera una colisión debidoa la combinación de más de un módulo.

10.3 Ejecución y Traza

Para analizar la traza en caso de presencia de definiciones locales, tomamos el ejemplo de ***sumaCuadrados***, mostrando la invocación, resolución y traza:

```
1  sumaCuadrados 4 3
2   = cuadrN + cuadrM
3    where
4     cuadrN = 4*4 = 16
5     cuadrM = 3*3 = 9
6   = 16 + 9
7   = 25
```

Se analiza a continuación un ejemplo más complejo, y se comentan los alcances. Sea la función ***maxOcurTres*** que devuelve a partir de tres enteros, el valor máximo junto con el número de ocurrencias. Así

```
1  maxOcurTres :: Int -> Int -> Int -> (Int, Int)
```

Una solución natural sería encontrar el mayor de los tres y ver cuantas veces se repite. Se definen en el código cuatro funciones: ***maxOcurTres*** (línea 1 y resultado del programa), ***cuentaIguales*** (línea 8), ***maxiTres*** (línea 14) y ***maxi*** (línea 17).

```
1  maxOcurTres :: Int -> Int -> Int -> (Int, Int)
2  maxOcurTres n m p
3    = ( max, cantiguales )
4     where
5       max         = maxiTres n m p
6       cantiguales = cuentaIguales max n m p
7
8  cuentaIguales val n m p
9    = esval n + esval m + esval p
10    where
11     esval :: Int -> Int
12     esval x = if x == val then 1 else 0
13
14 maxiTres n m p
15   = maxi (maxi n m) p
16
17 maxi :: Int -> Int-> Int
18 maxi n m
19  | n > m         = n
20  | otherwise = m
```

Una traza para la llamada ***maxOcurTres 2 1 2*** es:

```
maxOcurTres 2 1 2
 = ( max, cantIguales)
  where
    max = maxiTres 2 1 2
        = maxi (maxi 2 1) 2
        ?? 2 >= 1 = True
        = maxi 2 2
        ?? 2 >= 2 = True
        = 2
        = (2, cantIguales)
       where
          cantIguales = cuentaIguales 2 2 1 2
                      = esval 2 + esval 1 + esval 2
               where
                  esval 2 = if 2==2 then 1 else 0
                          = if True then 1 else 0
                          = 1
                      = 1 + esval 1 + esval 2
                  esval 1 = if 1==2 then 1 else 0
                          = if False then 1 else 0
```

```
                        = 0
                    = 1 + 0 + esval 2
                     ...
                    = 1 + 0 + 2
                    = 2
      = (2,2)
```

11 Métodos para definir funciones

11.1 Coincidencia de Patrones

```
1 fact :: Int -> Int
2 fact 0 = 1
3 fact n = n * fact (n-1)
```

11.2 Expresión Condicional

```
1 sign1 :: Int -> String
2 sign1 x =
3   if x < 0 then "Negativo"
4    else if x > 0 then "Positivo"
5    else "Cero"
```

11.3 Centinelas

```
1 absol :: Int -> Int
2 absol n
3   | (n>=0)     =  n
4   | otherwise  = (-n)
```

11.4 Sentencia Case

```
1 caso :: Int -> Int
2 caso x = case x of
3    0 -> 1
4    1 -> 5
5    2 -> 2
```

11.5 Ecuaciones y Definiciones Locales

```
1 sumaCuad :: Int -> Int -> Int
2 sumaCuad x y =  cuadX + cuadY
3    where
```

```
4     cuadX = x * x
5     cuadY = y * y
```

Part III

Tipos de Datos Estructurados

12 Tuplas

Las tuplas representan los registros en Haskell. Las tuplas son construidas utilizando tipos de datos simples o estructurados. La definición formal de tuplas es la siguiente: el tipo de dato

$t_1, t_2, \ldots , t_n$

consiste de tuplas con los valores

$(v_1, v_2, \ldots , v_n)$

tales que v_1::t_1, v_2::t_2, ..., v_n::t_n, en otras palabras, cada componente de la tupla v_i tiene su propio tipo asociado, que se nota como t_i. Cuando se utilizan tuplas, son útiles los tipos sinónimo que le dan un nombre a un tipo. Por ejemplo:

```
1  Type Persona = (String, String, Int)
```

de forma que podemos escribir

```
1  mary :: Persona
2  mary = ("Ana María", "0800-000-8888", 40)
3  parEnteros :: (Int, Int)
4  parEnteros = (32, 33)
```

12.1 Funciones sobre Tuplas

Las funciones sobre tuplas pueden ser según cualquier método utilizado para definir funciones, pero se puede utilizar coincidencia de patrones (*pattern matching*) para ubicar los componentes de la tupla, por ejemplo con el patrón ***(x,y)***:

```
1  sumapar :: (Int, Int) -> Int
2  sumapar (x, y) = x + y
```

una aplicación de la función anterior se evaluaría de la siguiente forma:

```
1  sumapar parEnteros
2   = sumapar (32, 33)
3   = 32 + 33
4   = 65
```

Los siguientes son otros ejemplos del uso de tuplas en la definición de funciones:

```
1  Desplazar :: ((Int, Int), Int) -> (Int, (Int, Int))
2  desplazar ((x, y), z) = (x, (y, z))
3
4  nombre :: Persona -> String
5  telefono :: Persona -> String
6  edad :: Persona -> String
7
8  nombre (a, b, c) = a
9  telefono (a, b, c) = b
10 edad (a, b, c) = c
```

```
ghci> nombre mary
"Ana María"
```

De hecho, las definiciones de `mary` y de `parEnteros` son definiciones de funciones con tuplas, puesto que se utilizan como funciones constantes, igual que en la matemática.

12.1.1 Conversión de tipos con Tuplas

También se pueden aplicar las funciones de conversión de tipos con los tipos estructurados de Tuplas. Por ejemplo, para convertir datos del tipo tupla:

```
ghci > (read "12"::Int, read "True"::Bool)
(12,True)
ghci > show (3,4)
"(3,4)"
```

13 Listas

Las listas son uno de los tipos de datos más importantes en la programación funcional, cuya principal razón es que las listas son inherentemente recursivas. Una lista puede ser vacía o un elemento seguido de una lista. La manipulación de las listas es una de las herramientas fundamentales para muchas de las técnicas de programación funcional.

Una lista es una secuencia de elementos de un mismo tipo, normalmente entre corchetes y cuyos elementos están separados por comas. Si `a` es un tipo cualquiera, entonces `[a]` representa el tipo de listas cuyos elementos son valores de tipo `a`. Por ejemplo, son listas:

```
1  [1, 2, 3, 4, 1, 4]          :: [Int]
2  [True]                      :: [Bool]
3  [False, True, False]        :: [Bool]
4  ['a', 'b', 'c', 'd']        :: [Char]
5  ["One", "Two", "Three"]     :: [String]
```

que pueden ser leído como una "lista de enteros con seis elementos", "una lista de booleanos con un único elemento", "una lista de tres elementos booleanos (Falso, Verdadero y Falso), una lista de caracteres/letras y una lista de tres cadenas (tipo `String`). Se define a **`String`** como un sinónimo para **`[Char]`**, por lo que las dos listas que se escriben a continuación son iguales:

```
['a', 'a', 'b']            :: [Char]
"aab"                      :: [Char]
```

de hecho, la quinta lista anterior también puede ser definida como

```
["One", "Two", "Three"]   :: [[Char]]
```

Por otro lado, la expresión

```
['a', 2, False]
```

no está permitida en Haskell, generaría un error como:

```
ghci> let mezclado = ['a', 2, False]
<interactive>:4:8:  error:
• Couldn't match expected type 'Char' with actual type 'Bool'
• In the expression:  True
In the expression:  ['a', 2, True]
In an equation for 'it':  it = ['a', 2, True]
```

13.1 Notación de Listas en Haskell

Por definición, una lista con los elementos e_1, e_2, *a* e_n, en donde cada e_i pertenece al tipo `t`, se escribe encerrándo los elementos entre corchetes, de la forma:

```
[e1 , e2, ..., en]
```

Hay varias formas de escribir expresiones de listas.

13.1.1 Lista Vacía

La forma más simple es la **lista vacía**, representada mediante `[]`. Es un tipo especial de lista que es un elemento de **cualquier tipo** de lista:

```
[] ::  [Int]
[] ::  [Int -> Int -> Int]
[] ::  [[Bool]]
```

13.1.2 Listas no vacías

Las listas no vacías se construyen enunciando explícitamente sus elementos (por ejemplo, `[2,5,22]`) o añadiendo un elemento al principio de otra lista utilizando el operador de construcción (`:`). Estas notaciones son equivalentes:

```
[2,5,22] = 2:[5,22] = 2:(5:[22]) = 2:(5:(22:[]))
```

El operador (`:`) es asociativo a la derecha, de forma que `2:5:22:[]` es equivalente a `(2:(5:(22:[])))`, una lista cuyo primer elemento es `2`, el segundo `5` y el tercero y último, `22`.

Se pueden construir listas de cualquier tipo incluyendo listas de funciones o listas de listas de números:

```
[mcd,sumaCuad]        :: [Int -> Int-> Int]
[[12,2],[],[2,12]]    :: [[Int]]
```

13.1.3 Lista por rangos

Son tan importantes las listas en Haskell, que hay maneras de generar rápidamente listas que contengan rangos de datos. Por ejemplo se pueden describir los elementos que pertenecen a una lista:

```
[n..m] = [n, n+1,..., m]
```

si n es mayor que m, la lista es vacía. Otros ejemplos:

```
[2..7] = [2, 3, 4, 5, 6, 7]
[3.1 .. 7.0) = [3.1, 4.1, 5.1, 6.1, 7.1]
['a' . . 'm'] = "abcdefghijklm"
[7, 6 .. 3] = [7, 6, 5, 4, 3]
[0.0, 0.3 .. 1.0] = [0.0, 0.3, 0.6, 0.9]
['a', 'd' . . 'o'] = "adgjm"
```

13.2 Listas por comprensión

Una de las características únicas de la programación funcional es la notación de listas por comprensión. Haskell permite generar elementos de una lista, verificando y transformando los elementos a partir de una lista inicial. Es una forma compacta de trabajo, que incluso permite definir directamente funciones complejas. Por ejemplo, suponiendo que la lista dato esta conformada por los elementos 2, 4, 7 y 9, es decir, "`dato = [2,4,7,9]`" entonces por comprensión de lista se define

```
ghci> [ 2 * n | n <- dato ]
[4,8,14,18]
```

que puede leerse como "la lista que se forma calculando el doble de todos los elementos de `dato`".

De la misma manera, si hemos definido la función esImpar, que devuelve verdadero o falso si un número es impar, entonces la definición de la siguiente lista devolverá:

```
ghci> [ esImpar n | n <- dato ]
[False, False, True, True]
```

Se pueden combinar las verificaciones, de forma que poniendo nuevas restricciones a la definición de la lista resulta:

```
ghci> [ 2 * n | n <- dato, esImpar n ]
[14,18]
```

```
[ 2 * n | n <- dato, esImpar n ]
  = [ 2 * n | n <- [2,4,7,9], esImpar n ]
donde
```

```
        n =  2  4  7  9
 esImpar n =  F  F  V  V
    2 * n =       14 18
```

incluso con una nueva restricción

```
ghci> [ 2 * n | n <- dato, esImpar n, n > 8 ]
[18]
```

```
[ 2 * n | n <- dato, esImpar n, n>3 ]
= [ 2 * n | n <- [2,4,7,9], esImpar n, n > 8 ]
donde
        n =  2  4  7  9
 esImpar n =  F  F  V  V
     n > 8 =        F  V
     2 * n =          18
```

```
[ 2 * n | n <- dato, esImpar n, n>3 ]
= [ 2 * n | n <- [2,4,7,9], esImpar n ]
donde
       n =  2  4  7  9
esImpar n =  F  F  V  V
    2 * n =       14 18
```

Para definir una lista por comprensión que permita calcular la suma de los elementos de cada tupla suponiendo la lista "[(2,4,5),(4,6,1),(8,4,2)]":

```
ghci> [ m + n + p | (m, n, p) <- [(2,4,5),(4,6,1),(8,4,2)] ]
[11,11,14]
```

```
[ m + n + p | (m, n, p) <- [(2,4,5),(4,6,1),(8,4,2)] ]
donde
      m =  2  4  8
      n =  4  6  4
      p =  5  1  2
  m+n+p = 11 11 14
```

De la misma manera, se puede considerar más de un recorrido de las listas:

```
ghci> [ x | x <- [ [1,2],[3,4,7] ], x <- x ]
[1,2,3,4,7]
```

```
[ x | x <- [ [1,2],[3,4,7] ], x <- x  ]
donde
    x =  [1,2]   [3,4,7]
    x =  1  2  3  4  7
```

si bien el comportamiento es distinto en caso de requerimientos más complejos, como:

```
ghci> [ x | x <- [1,2], x <- [3,4,7] ]
[3,4,7,3,4,7]
```

```
[ x | x <- [1,2], x <- [3,4,7] ]
donde
    x =  1        2
    x =  3  4  7  3  4  7
```

13.3 Funciones sobre listas

Por ejemplo, se puede escribir la función `longitud`, que calcula la longitud de una lista de números enteros como:

```
longitud :: [Int] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs
```

De hecho, la función `length` está incorporado en el preludio de Haskell, y puede calcular la longitud de las listas de **cualquier tipo**. Se define como

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

Por ejemplo, para calcular la sumatoria de los elementos de una lista de números enteros, se puede definir `sumatoria` como:

```
sumatoria::[Int]->Int
sumatoria [] = 0
sumatoria (x:xs)
  | xs == [] = x
  | otherwise = x + sumatoria xs
```

Para calcular el cuadrado de todos los elementos de una lista, se puede definir como:

```
cuadradoL::[Int]->[Int]
cuadradoL [] = []
cuadradoL (x:xs) = x^2 : cuadradoL xs
```

Para definir la función factorial, utilizando la definición de listas por comprensión y el uso de funciones sobre listas:

```
factorial2 :: Integer -> Integer
factorial2 n = product [1..n]
```

Para definir la función insertar elemento en lista delante del primer elemento mayor o igual, se define como:

```
inserta :: Int -> [Int] -> Int
inserta elem [] = [elem]
inserta elem (x:xs)
  | elem <= x  = elem : (x:xs)
  | otherwise = x : inserta elem xs
```

Una tríada pitagórica es una tripla de enteros (a,b,c) tales que $a^2+b^2=c^2$. Una función que encuentra las tríadas con a, b y c menores que 100 es:

```
triada :: Int -> [(Int,Int,Int)]
triada n = [(a,b,c) | a <- [1..n], b <- [1..n], c <- [1..n], a^2+b^2=c^2 ]
```

13.4 Operadores sobre listas

Si bien hay muchos operadores que trabajan sobre listas, los principales operadores o funciones que deben conocer son los siguientes:

Funciones	Ejemplos	Descripción

`:`	`1:[2,3] = [1,2,3]`
	`'H':"ola" = "Hola"`
	`"H":"ola" = error`[14]
		Anexa elementos a una lista

`++`	`[1,2,3] ++ [1,2] = [1,2,3,1,2]`
	`"muy" ++ "lindo" = "muy lindo"`
	`'h' ++ "ola" = error`[15]
		Concatena listas

`!!`	`[1,2,3] !!  0 = 1`
	`"animal" !!  3 = 'm'`
	`[1..10] !!  11 = error (fuera de rango)`
		Acceso a elemento particular

`head`	`head [1..20] = 1` `head [[1,2],[3,4],[5,6]] = [1,2]`	Cabeza o primer elemento
`tail`	`tail [1,2,3] = [2,3]` `tail [3] = []`	Cola
`length`	`length [1..20] = 20` `length [(10,20),(1,2),(15,16)] = 3`	Longitud
`reverse`	`reverse [1,2,3,4] = [4,3,2,1]`	Invertir
`elem`	`elem 4 [1,2,3,4] = Verdadero` `elem 6 [1,2,3,4] = Falso`	Verifica si el elemento es miembro de la lista
`take`	`take 5 [2,4..100] = [2,4,6,8,10]` `take 5 [1,2,3] = [1,2,3]`	Recupera los primeros n elementos de una lista
`drop`	`drop 2 [1,2,3,4,5] = [3,4,5]` `drop 5 "muy bonito" = "onito"`	Elimina los primeros n elementos de una lista

[14] Tipos incompatibles
[15] Error de tipos incompatibles. El primer operando no es una lista, sino un elemento del tipo `Char`.