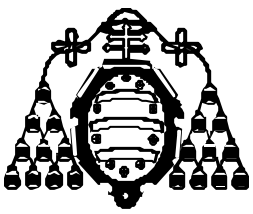
Introducción al lenguaje *Haskell*

Jose E. Labra G.

e-mail: labra@lsi.uniovi.es
http://lsi.uniovi.es/~labra

Octubre 1998



Universidad de Oviedo
Departamento de Informática

1.- Introducción

Haskell es un **lenguaje funcional puro**, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa, tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones, y definiciones de listas por comprehensión.

Incorpora, además, otras características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos.

El contenido de este cuaderno se divide en dos partes, en la primera se introduce el modelo funcional indicando sus ventajas respecto al modelo imperativo sin referirse a ningún lenguaje funcional en concreto. En la segunda parte se describe el lenguaje Haskell y sus principales características.

Se utiliza como referencia el entorno de programación *Hugs* y se supone que el lector tiene unos mínimos conocimientos del modelo de programación imperativo o tradicional.

Referencias: Aunque al final de estos apuntes se incluye una bibliografía consultada para su realización, las principales fuentes han sido el manual *"An Introduction to Gofer"* de Mark P. Jones incluido en la distribución del sistema Gofer y el artículo *"A gentle Introduction to Haskell"* de P. Hudak.

2.- Introducción a la Programación funcional

2.1.-Crisis del *Software*

A principios de la década de los setenta aparecieron los primeros síntomas de lo que se ha denominado ***crisis del software***. Los programadores que se enfrentan a la construcción de grandes sistemas de software observan que sus **productos no** son **fiables**. La alta tasa de errores conocidos (*bugs)* o por conocer pone en peligro la confianza que los usuarios depositan en sus sistemas.

Cuando los programadores quieren corregir los errores detectados se enfrentan a una dura tarea de **mantenimiento**. Cuando se intenta corregir un error detectado, una pequeña modificación trae consigo una serie de efectos no deseados sobre otras partes del sistema que, en la mayoría de las ocasiones, empeora la situación inicial.

La raíz del problema radica en la dificultad de **demostrar que el sistema cumple los requisitos** especificados. La verificación formal de programas es una técnica costosa que en raras ocasiones se aplica.

Por otro lado, el incremento en la potencia de procesamiento lleva consigo un incremento en la **complejidad del software**. Los usuarios exigen cada vez mayores prestaciones cuyo cumplimiento sigue poniendo en evidencia las limitaciones de recursos y la fragilidad del sistema.

Las posibles soluciones podrían englobarse en las siguientes líneas de investigación:

- Ofrecer nuevos desarrollos de la **Ingeniería del Software** que permitan solucionar el problema del Análisis y Diseño de grandes Proyectos Informáticos. Actualmente las Metodologías Orientadas a Objetos han aportado una evolución importante dentro de este campo.
- Proporcionar Sistemas de **Prueba y Verificación** de programas cuya utilización no sea costosa.
- Construir técnicas de **Síntesis de Programas** que permitan obtener, a partir de unas especificaciones formales, código ejecutable.
- Diseñar nuevas Arquitecturas de Computadoras, en particular, técnicas de procesamiento en paralelo.
- Proponer un **modelo de computación diferente** al modelo imperativo tradicional**.**

Esta última solución se basa en la idea de que los problemas mencionados son inherentes al modelo computacional utilizado y su solución no se encontrará a menos que se utilice un modelo diferente. Entre los modelos propuestos se encuentra la **programación funcional** o aplicativa, cuyo objetivo es describir los problemas mediante funciones matemáticas puras sin efectos laterales, y la **programación lógica** o declarativa, que describe los problemas mediante **relaciones** entre objetos.

2.2.-Problemas del Modelo imperativo

Los lenguajes de programación tradicionales como Pascal, C, C++, ADA, etc. forman una abstracción de la máquina de Von-Neumann caracterizada por:

- **Memoria Principal** para almacenamiento de datos y código máquina.
- **Unidad Central de Proceso** con una serie de registros de almacenamiento temporal y un conjunto instrucciones de cálculo aritmético, modificación de registros y acceso a la Memoria Principal.

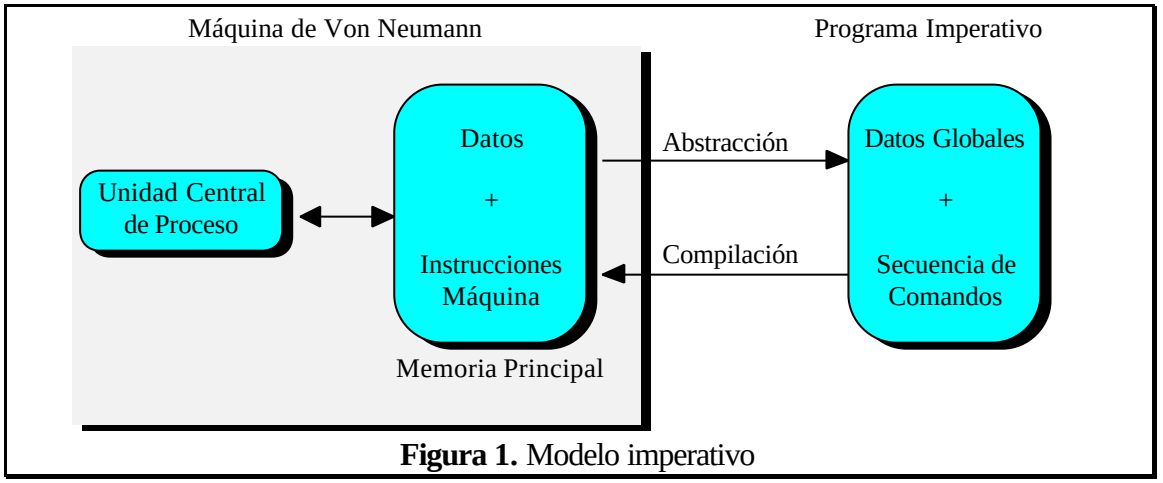


Figura 1. Modelo imperativo

Los programas imperativos poseen una serie de datos globales y una secuencia de comandos ó código. Estos dos elementos forman una abstracción de los datos y código de la memoria principal. Para hacer efectiva dicha abstracción se compila el código fuente para obtener código máquina. El modelo imperativo ha tenido un gran éxito debido a su sencillez y proximidad a la arquitectura de los computadores convencionales.

El programador trabaja en un nivel cercano a la máquina que le permite generar programas eficientes. Con esta proximidad aparece, sin embargo, una dependencia entre el algoritmo y la arquitectura que impide, por ejemplo, utilizar algoritmos programados para arquitecturas secuenciales en arquitecturas paralelas.

Los algoritmos en lenguajes imperativos se expresan mediante una **secuencia de órdenes** que modifican el **estado** de un programa accediendo a los datos globales de la memoria.

Las instrucciones de acceso a los datos globales destruyen el contenido previo de un dato asignando un nuevo valor. Por ejemplo, la instrucción `x:=x+1` incrementa el valor anterior de `x` de forma que la siguiente vez que se acceda a `x`, su valor ha cambiado. Las asignaciones

producen una serie de efectos laterales que oscurecen la semántica del lenguaje. Considérese el siguiente programa en un lenguaje imperativo[1]:

```
Program Prueba;                  --- Programa Principal
VAR                              BEGIN
  flag:BOOLEAN;                   flag:=TRUE;
                                  ...
FUNCTION f (n:INTEGER):           WRITE( f(1) );          <- Escribe 2
INTEGER;                          WRITE( f(1) );          <- Escribe 1
BEGIN                             ...
     flag := NOT flag;            WRITE( f(1) + f(2) );   <- Escribe 4
     IF flag THEN f := n;         WRITE( f(2) + f(1) );   <- Escribe 5
             ELSE f := 2*n        ...
END;                             END.
```

Figura 2: Programa imperativo

La expresión `f(1)` toma diferentes valores dependiendo de la secuencia de ejecución. No se cumplen propiedades matemáticas simples como la propiedad commutativa, en el ejemplo, `f(1) + f(2)` no es igual a `f(2) + f(1)`. Es difícil demostrar la corrección de un programa cuando no se cumplen este tipo de propiedades. Además, al no tener el mismo valor, no es posible evaluar en paralelo `f(1)` con `f(2)` y sumar el resultado.

Las asignaciones dificultan, por tanto, el estudio de la corrección de un programa e impiden que el compilador puede realizar evaluaciones en paralelo. Además, el compilador tiene graves problemas para optimizar código con asignaciones destructivas. Considérese, por ejemplo, la sentencia

```
z := (2*a*y+b)*(2*a*y+c);
```

La expresión a la derecha de la asignación contiene una subexpresión común "`2*a*y`". Muchos compiladores eliminarían la evaluación redundante de ambas expresiones substituyendo la asignación original con

```
t := 2*a*y;
z := (t+b)*(t+c);
```

Sin embargo, no siempre es posible realizar dicha substitución cuando intervienen asignaciones. Considérese, por ejemplo:

```
y := 2*a*y + b;
z := 2*a*y + c;
```

De nuevo aparece una subexpresión común. Sin embargo, si se intenta cambiar el código con

```
t := 2*a*y;
y := t + b;
z := t + c;
```

[1]Por motivos didácticos, se ha elegido un lenguaje similar al Pascal, con la sentencia **`return.`**

el resultado no es equivalente. El problema radica en que el valor de y ha cambiado durante la primera asignación.

Aunque es posible que un compilador analice el código para detectar cuándo es posible realizar este tipo de substituciones, dicho análisis resulta complicado y se realiza en escasas ocasiones.

2.3.-Modelo Funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos laterales y, por tanto, **sin asignaciones destructivas**.

El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre sistema y usuario: el **usuario** introduce una **expresión inicial** y el **sistema** la **evalúa** mediante un proceso de reducción. En este proceso se utilizan las **definiciones de función** realizadas por el **programador** hasta obtener un **valor** no reducible.

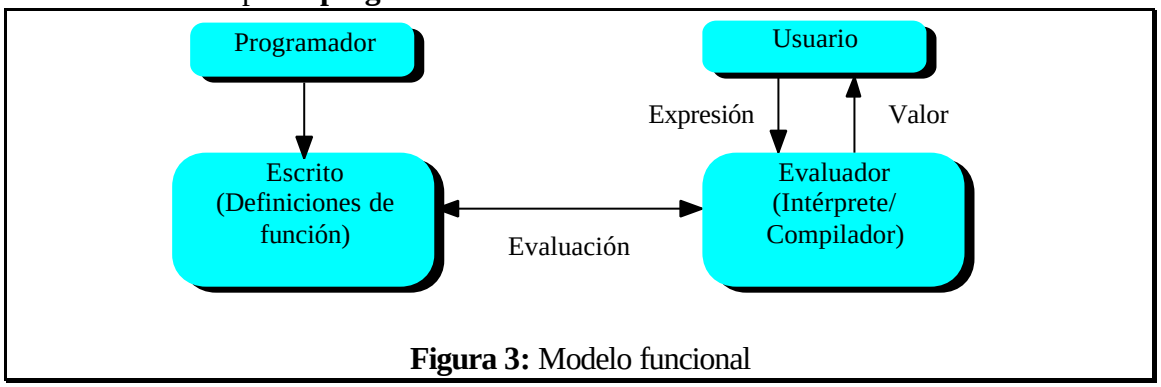


Figura 3: Modelo funcional

El valor que devuelve una función está únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor (esta propiedad se conoce como **transparencia referencial**). Es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad commutativa, asociativa, etc.

El programador se encarga de definir un conjunto de funciones sin preocuparse de los métodos de evaluación que posteriormente utilice el sistema. Este modelo deja mayor libertad al sistema de evaluación para incorporar pasos intermedios de transformación de código y paralelización ya que las funciones no tienen efectos laterales y no dependen de una arquitectura concreta.

La importancia de la programación funcional no radica únicamente en no utilizar asignaciones destructivas. Por el contrario, este modelo promueve la utilización de una serie de características como las funciones de orden superior, los sistemas de inferencia de tipos, el polimorfismo, la evaluación perezosa, etc.

Funciones orden superior

Un lenguaje utiliza funciones de orden superior cuando permite que las funciones sean tratadas como valores de *primera clase*, permitiéndo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados.

```
reaplica (f: Integer -> Integer,
          x: Integer) : Integer
begin
 reaplica := f (f (x));
end

incr (n : Integer) : Integer
begin
 incr := n+1;
end
```

```
--- Programa Principal
...
write(reaplica (incr,0));
...
```

Figura 4: Ejemplo de función de orden superior

En el ejemplo anterior, la función `reaplica` tiene como argumentos una función `f` (que toma elementos de un tipo `Integer` y devuelve elementos de tipo `Integer`) y un valor `x` (de tipo `Integer`). La llamada `reaplica(incr,0)` equivale a aplicar la función `incr` dos veces sobre el valor cero, es decir, equivale a `incr(incr(0))`.

La utilización de funciones de orden superior proporciona una mayor flexibilidad al programador, siendo una de las características más sorbresalientes de los lenguajes funcionales. De hecho, en algunos ámbitos se considera que la propiedad que distingue un lenguaje funcional de otro es la utilización de funciones de orden superior.

Sistemas de Inferencia de Tipos y Polimorfismo

Muchos lenguajes funcionales han adoptado un sistema de inferencia de tipos que consiste en:

- El programador no está obligado a declarar el tipo de las expresiones
- El compilador contiene un algoritmo que infiere el tipo de las expresiones
- Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido.

Los sistemas de inferencia de tipos permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Por ejemplo, si el programador declara la siguiente función:

```
eligeSaludo x = if x then "adios"
                     else "hola"
```

El sistema infiere automáticamente que el tipo es `eligeSaludo::Bool -> String` y, si el programador hubiese declarado que tiene un tipo diferente, el sistema daría un error de tipos.

Los sistemas de inferencia de tipos aumentan su flexibilidad mediante la utilización de **polimorfismo**[2]**.**

El polimorfismo permite que el tipo de una función dependa de un parámetro. Por ejemplo, si se define una función que calcule la longitud de una lista, una posible definición sería:

```
long ls = IF vacia(L) then 0
                      else 1 + long(cola(L));
```

El sistema de inferencia de tipos inferiría el tipo[3]: `long::[x] -> Integer`, indicando que tiene como argumento una lista de elementos de un tipo a cualquiera y que devuelve un entero.

En un lenguaje sin polimorfismo sería necesario definir una función `long` para cada tipo de lista que se necesitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

Evaluación Perezosa

Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si éstos serán utilizados. Por ejemplo:

```
g (x:Integer):Integer
Begin
 (*Bucle infinito*)
 while true do x:=x;
End;

f (x:Integer,
   y:Integer) :Integer
Begin
 return (x+3);
End;
```

```
--- Programa Principal
BEGIN

 write(f(4,g(5)));

END.
```

Figura 5: Evaluación perezosa

Con el sistema de evaluación tradicional, el programa anterior no devolvería nada, puesto que al intentar evaluar `g(5)` el sistema entraría en un bucle infinito. Dicha técnica de evaluación se conoce como evaluación ansiosa (*eager evaluation*) porque evalúa todos los argumentos de una función antes de conocer si son necesarios.

Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa (*lazy evaluation*) que consiste en no evaluar un argumento hasta que no se necesita. En el ejemplo anterior, si se utilizase evaluación perezosa, el sistema escribiría `7`.

[2] En otros contextos se conoce como genericidad o polimorfismo paramétrico

[3] Para ser más exactos, el tipo inferido sería: `long:: ∀x . [x] -> Integer`. Es decir, el tipo x está cuantificado universalmente (puede ser cualquier tipo)

Uno de los beneficios de la evaluación perezosa consiste en la posibilidad de manipular estructuras de datos 'infinitas'. Evidentemente, no es posible construir o almacenar un objeto infinito en su totalidad. Sin embargo, gracias a la evaluación perezosa se puede construir objetos *potencialmente* infinitos pieza a pieza según las necesidades de evaluación.

2.4.-Evolución del modelo funcional

Los orígenes teóricos del modelo funcional se remontan a los años 30 en los cuales Church propuso un nuevo modelo de estudio de la computabilidad mediante el **cálculo lambda**. Este modelo permitía trabajar con funciones como objetos de primera clase. En esa misma época, Shönfinkel y Curry[4] construían los fundamentos de la lógica combinatoria que tendrá gran importancia para la implementación de los lenguajes funcionales.

Hacia 1950, John McCarthy diseñó el lenguaje **LISP** (List Processing) que utilizaba las listas como tipo básico y admitía funciones de orden superior. Este lenguaje se ha convertido en uno de los lenguajes más populares en el campo de la Inteligencia Artificial. Sin embargo, para que el lenguaje fuese práctico, fue necesario incluir características propias de los lenguajes imperativos como la asignación destructiva y los efectos laterales que lo alejaron del paradigma funcional. Actualmente ha surgido una nueva corriente defensora de las características funcionales del lenguaje encabezada por el dialecto **Scheme,** que aunque no es puramente funcional, se acerca a la definición original de McCarthy.

En 1964, **Peter Landin** diseñó la máquina abstracta SECD para mecanizar la evaluación de expresiones, definió un subconjunto no trivial de Algol-60 mediante el cálculo lambda e introdujo la familia de **lenguajes ISWIM** (*If You See What I Mean)* con innovaciones sintácticas (operadores infijos y espaciado) y semánticas importantes.

En 1978 **J. Backus** (uno de los diseñadores de FORTRAN y ALGOL) consiguió que la comunidad informática prestara mayor atención a la programación funcional con su artículo *"Can Programming be liberated from the Von Neumann style?"* en el que criticaba las bases de la programación imperativa tradicional mostrando las ventajas del modelo funcional. Además Backus diseñó el lenguaje funcional **FP** (*Functional Programming)* con la filosofía de definir nuevas funciones combinando otras funciones.

A mediados de los 70, **Gordon** trabajaba en un sistema generador de demostraciones denominado LCF que incluía el lenguaje de programación **ML** (*Metalenguaje).* Aunque el sistema LCF era interesante, se observó que el lenguaje ML podía utilizarse como un lenguaje de propósito general eficiente. ML optaba por una solución de compromiso entre el modelo funcional y el imperativo ya que, aunque contiene asignaciones destructivas y Entrada/Salida con efectos laterales, fomenta un estilo de programación claramente funcional. Esa solución permite que los sistemas ML compitan en eficiencia con los lenguajes imperativos. A mediados de los ochenta se realizó un esfuerzo de estandarización que culminó con la

[4] Como curiosidad, destacar que Haskell B. Curry ha dado nombre al lenguaje haskell y a una técnica conocida como *curryficación* descubierta por Shönfinkel.

definición de **SML** (*Stándar ML*). Este lenguaje es fuertemente tipado con resolución estática de tipos, definición de funciones polimórficas y tipos abstractos. Actualmente, los sistemas en SML compiten en eficiencia con los sistemas en otros lenguajes imperativos y han aparecido proyectos como Fox Project 1994 [HL94] que pretenden desarrollar un nuevo lenguaje ML2000 con subtipos y módulos de orden superior.

Al mismo tiempo que se desarrollaban FP y ML, **David Turner** (primero en la Universidad de St. Andrews y posteriormente en la Universidad de Kent) trabajaba en un nuevo estilo de lenguajes funcionales con evaluación perezosa y definición de funciones mediante encaje de patrones. El desarrollo de los lenguajes SASL (*St. Andrews Static Language)*, KRC (*Kent Recursive* Calculator) y **Miranda**[5] tenía como objetivo facilitar la tarea del programador incorporando facilidades sintácticas como las guardas, el encaje de patrones, las listas por comprehensión y las secciones.

A comienzos de los ochenta surgieron una **gran cantidad de lenguajes funcionales** debido a los avances en las técnicas de implementación. Entre éstos, se podrían destacar Hope, LML, Orwell, Erlang, FEL, Alfl, etc. Esta gran cantidad de lenguajes perjudicaba el desarrollo del paradigma funcional. En septiembre de 1987, se celebró la conferencia FPCA en Portland, Oregon, en la que se discutieron los problemas que creaba esta proliferación. Se decidió formar un comité internacional que diseñase un nuevo lenguaje puramente funcional de propósito general denominado Haskell [Hud92].

Con el lenguaje **Haskell** se pretendía unificar las características más importantes de los lenguajes funcionales. como las funciones de orden superior, evaluación perezosa, inferencia estática de tipos, tipos de datos definidos por el usuario, encaje de patrones y listas por comprehensión. Al diseñar el lenguaje se observó que no existía un tratamiento sistemático de la sobrecarga con lo cual se construyó una nueva solución conocida como las clases de tipos. El lenguaje incorporaba, además, Entrada/Salida puramente funcional y definición de *arrays* por comprehensión.

En Mayo de 1996 aparecía la **versión 1.3** del lenguaje *Haskell* [Has95] que incorporaba, entre otras características, mónadas para Entrada/Salida, registros para nombrar componentes de tipos de datos, clases de constructores de tipos y diversas librerías de propósito general. Posteriormente, surge la versión 1.4 con ligeras modificaciones.

En 1998 se ha decidido proporcionar una versión estable del lenguaje, que se denominará *Haskell98* a la vez que se continúa la investigación de nuevas características.

La principal información sobre el lenguaje Haskell puede consultarse en la dirección:

http://www.haskell.org

[5] Miranda es una marca registrada de Research Software Ltd.

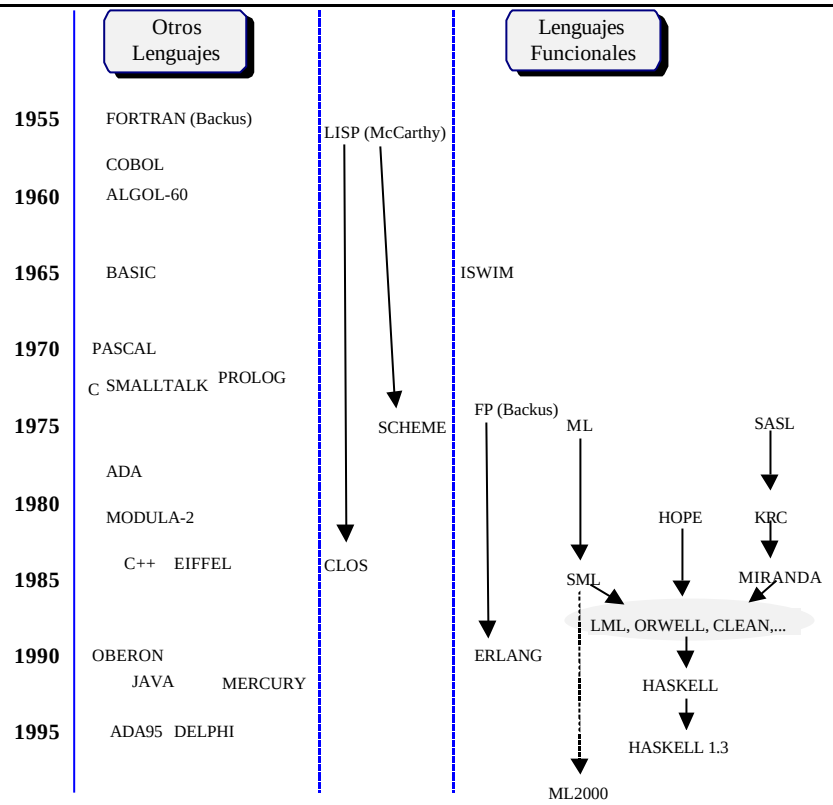


Figura 6: Evolución de los lenguajes funcionales

3.- Lenguaje Haskell

3.1.-Conceptos básicos

El entorno *HUGS* funciona siguiendo el modelo de una calculadora en el que se establece una sesión interactiva entre el ordenador y el usuario. Una vez arrancado, el sistema muestra un *prompt* "?" y espera a que el usuario introduzca una expresión (denominada **expresión inicial** y presione la tecla <RETURN>. Cuando la entrada se ha completado, el sistema evalúa la expresión e imprime su valor antes de volver a mostrar el *prompt* para esperar a que se introduzca la siguiente expresión.

Ejemplo:

```
? (2+3)*8
40

? sum [1..10]
55
```

En el primer ejemplo, el usuario introdujo la expresión `"(2+3)*8"` que fue evaluada por el sistema imprimiendo como resultado el valor `"40".`

En el segundo ejemplo, el usuario tecleó `"sum [1..10]".` La notación `[1..10]` representa la lista de enteros que van de 1 hasta 10, y `sum` es una función estándar que devuelve la suma de una lista de enteros. El resultado obtenido por el sistema es:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

En los ejemplos anteriores, se utilizaron funciones estándar, incluidas junto a una larga colección de funciones en un fichero denominado "*estándar prelude*" que es cargado al arrancar el sistema. Con dichas funciones se pueden realizar una gran cantidad de operaciones útiles. Por otra parte, el usuario puede definir sus propias funciones y almacenarlas en un fichero de forma que el sistema pueda utilizarlas en el proceso de evaluación. Por ejemplo, el usuario podría crear un fichero *fichero.hs* con el contenido:

```
cuadrado::Integer -> Integer
cuadrado x = x * x

menor::(Integer, Integer) -> Integer
menor (x,y) = if x <= y then x else y
```

Para poder utilizar las definiciones anteriores, es necesario cargar las definiciones del fichero en el sistema. La forma más simple consiste en utilizar el comando ":`load`":

```
? :l fichero.hs
Reading script file "fichero.hs"
. . .
?
```

Si el fichero se cargó con éxito, el usuario ya podría utilizar la definición:

```
? cuadrado (3+4)
49

? cuadrado (menor (3,4))
9
```

Es conveniente **distinguir** entre un **valor** como ente abstracto y su **representación**, un expresión formada por un conjunto de símbolos. En general a un mismo valor abstracto le pueden corresponder diferentes representaciones. Por ejemplo, `7+7`, `cuadrado 7`, `49`, `XLIX` (49 en números romanos), `110001` (en binario) representan el mismo valor.

El proceso de **evaluación** consiste en tomar una expresión e ir transformándola aplicando las definiciones de funciones (introducidas por el programador o predefinidas) hasta que no pueda transformarse más. La expresión resultante se denomina representación canónica y es mostrada al usuario.

En el proceso de evaluación pueden seguirse diversas trayectorias, por ejemplo:

```
cuadrado (3+4)
=       { evaluando la suma }
cuadrado 7
=       { utilizando la definición cuadrado x = x * x }
7 * 7
=       { evaluando la multiplicación }
49
```

En el ejemplo anterior se evaluaron primero las expresiones internas antes de la aplicación de la función. Otra posibilidad sería utilizar la definición de la función antes de evaluar los argumentos.

```
cuadrado (3+4)
=       { utilizando la definición cuadrado x = x * x }
(3+4) * (3+4)
=       { evaluando la parte izquierda de la multiplicación }
7 * (3+4)
=       { evaluando la parte derecha de la multiplicación }
7 * 7
=       { evaluando la multiplicación }
49
```

El primer esquema, evaluar los argumentos antes de llamar a la función, se denomina llamada por valor y se asocia con la evaluación ansiosa. El segundo se denomina llamada por nombre y está asociado a la evaluación perezosa. En algunas ocasiones, la llamada por valor puede no terminar, mientras que la llamada por nombre sí. Por ejemplo, si se carga el siguiente programa:

```
infinito:: Integer
infinito = infinito + 1

tres:: Integer -> Integer
tres x = 3
```

Al evaluar la expresión "`tres infinito`" por valor se obtiene:

```
tres infinito
=     { utilizando la definición infinito = infinito + 1}
tres (infinito + 1)
=     { utilizando la definición infinito = infinito + 1}
tres ((infinito + 1) + 1)
=     ...
```

Si se hubiese evaluado por nombre se hubiese obtenido:

```
tres infinito
=     { utilizando la definición tres x = 3}
3
```

Existen valores que no tienen representación canónica (por ejemplo, las funciones) o que tienen una representación canónica infinita (por ejemplo, el número `pi`).

Por el contrario, otras expresiones, no representan ningún valor. Por ejemplo, la expresión (`1/0`) o la expresión (`infinito`). Se dice que estas expresiones representan el valor indefinido.

3.2.- Nombres de función: Identificadores y operadores

Existen dos formas de nombrar una función, mediante un identificador (por ejemplo, *sum*, *product* y *fact*) y mediante un símbolo de operador (por ejemplo, * y +)

El sistema distingue entre los dos tipos según la forma en que estén escritos:

A. Un **identificador** comienza con una letra del alfabeto seguida, opcionalmente, por una secuencia de caracteres, cada uno de los cuales es, una letra, un dígito, un apóstrofe (') o un subrayado (_).

Los identificadores que representan funciones o variables deben comenzar por letra minúscula (los identificadores que comienzan con letra mayúscula se emplearán como funciones constructoras). Los siguientes son ejemplos de posibles identificadores:

```
sum   f     f''   intSum     elemento_dos     do'until'zero
```

Los siguientes identificadores son palabras reservadas y no pueden utilizarse como nombres de funciones o variables:

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

B. Un símbolo de **operador** es escrito utilizando uno o más de los siguientes caracteres:

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

Además, el carácter (~) también se permite, aunque sólo en la primera posición del nombre. Los nombres de operador que comienzan con (:) son utilizados para funciones constructoras

como los identificadores que comienzan por mayúscula mencionados anteriormente. Los siguientes símbolos tienen usos especiales:

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

Todos los otros símbolos de operador se pueden utilizar como variables o nombre de función, incluyendo los siguientes:

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

Se proporcionan dos mecanismos simples para utilizar un identificador como un símbolo de operador o un símbolo de operador como un identificador:

C. Cualquier identificador será tratado como un símbolo de operador si está encerrado entre comillas inversas (`). Así, cualquier expresión de la forma "x `id` y" es equivalente a "id x y"

D. Cualquier símbolo de operador puede ser tratado como un identificador encerrándolo en paréntesis. Por ejemplo, "x + y" podría escribirse como "(+) x y".

Cuando se trabajan con símbolos de operador es necesario tener en cuenta:

A. La **precedencia** La expresión "2 * 3 + 4" podría interpretarse como "(2 * 3) + 4" o como "2 * (3 + 4)". Para resolver la ambigüedad, cada operador tiene asignado un valor de precedencia (un entero entre 0 y 9). En una situación como la anterior, se comparan los valores de precedencia y se utiliza primero el operador con mayor precedencia (en el *standar prelude* el (+) y el (*) tienen asignados 6 y 7, respectivamente, por lo cual se realizaría primero la multiplicación).

B. La **asociatividad**: La regla anterior resolvía ambigüedades cuando los símbolos de operador tienen distintos valores de precedencia, sin embargo, la expresión "1 - 2 - 3" puede ser tratada como "(1 - 2) - 3" resultando -4 o como "1 - (2 - 3)" resultando 2. Para resolverlo, a cada operador se le puede definir una regla de asociatividad. Por ejemplo, el símbolo (-) se puede decir que es:
   Asociativo a la izquierda: si la expresión "x-y-z" se toma como "(x-y)-z"
   Asociativo a la derecha: si la expresión "x-y-z" se toma como "x-(y-z)"
   No asociativo: Si la expresión "x-y-z" se rechaza como un error sintáctico.

En el *standar prelude* el (-) se toma como asociativo a la izquierda, por lo que la expresión "1 - 2 - 3" se tratará como "(1-2)-3".

Por defecto, todo símbolo de operador se toma como no-asociativo y con precedencia 9. Estos valores pueden ser modificados mediante una declaración con los siguientes formatos:

`infixl digito ops`	Para declarar operadores asociativos a la izquierda
`infixr digito ops`	Para declarar operadores asociativos a la derecha

`infix digito ops` Para declarar operadores no asociativos

`ops` representa una lista de uno o más símbolos de operador separados por comas y `digito` es un entero entre 0 y 9 que asigna una precedencia a cada uno de los operadores de la lista. Si el dígito de precedencia se omite se toma 9 por defecto.
Existen ciertas restricciones en la utilización de estas declaraciones:

- Sólo pueden aparecer en ficheros de definición de función que sean cargados en el sistema.
- Para un operador particular, sólo se permite una declaración

En el *standar prelude* se utilizan las siguientes declaraciones:

```
infixl 9 !!
infixr 9 .
infixr 8 ^
infixl 7 *
infix  7 /, `div`, `rem`, `mod`
infixl 6 +, -
infix  5 \\
infixr 5 ++, :
infix  4 ==, /=, <, <=, >=, >
infix  4 `elem`, `notElem`
infixr 3 &&
infixr 2 ||
```

Tabla de precedencia/asociatividad de operadores

A continuación se muestran una serie de expresiones y las formas equivalentes siguiendo las declaraciones del *standar prelude*

Expresión:	Equivalente a:	Motivos
`1+2-3`	`(1 + 2) - 3`	(+) y (-) tienen la misma precedencia y son asociativos a la izquierda.
`x : ys ++ zs`	`x : (ys ++ zs)`	(:) y (++) tienen la misma precedencia y son asociativos a la derecha.
`x == y || z`	`(x == y) || z`	(==) tiene más precedencia que (||)
`3+4*5`	`3+(4*5)`	(*) tiene más precedencia que (+)
`y `elem` z:zs`	`y `elem` (z:zs)`	(:) tiene más precedencia que `elem`
`12 / 6 / 3`	`error sintáctico`	(/) no es asociativo

Obsérvese que la aplicación de funciones tiene más precedencia que cualquier símbolo de operador. Por ejemplo, la expresión "`f x + g y`" equivale a "`(f x) + (g y)`"
Otro ejemplo que a menudo da problemas es la expresión "`f x + 1`", que es tratada como "`(f x)+1`" en lugar de "`f(x+1)`".

3.3.-Tipos

Una parte importante del lenguaje Haskell lo forma el sistema de tipos que es utilizado para detectar errores en expresiones y definiciones de función.

El universo de valores es particionado en colecciones organizadas, denominadas *tipos*. Cada tipo tiene asociadas un conjunto de operaciones que no tienen significado para otros tipos, por ejemplo, se puede aplicar la función (+) entre enteros pero no entre caracteres o funciones.

Una propiedad importante del Haskell es que es posible asociar un único tipo a toda expresión bien formada. Esta propiedad hace que el Haskell sea un lenguaje fuertemente tipado. Como consecuencia, cualquier expresión a la que no se le pueda asociar un tipo es rechazada como incorrecta antes de la evaluación. Por ejemplo:

```
f x = 'A'
g x = x + f x
```

La expresión `'A'` denota el caracter `A`. Para cualquier valor de `x`, el valor de `f x` es igual al caracter `'A'`, por tanto es de tipo `Char`. Puesto que el `(+)` es la operación suma entre números, la parte derecha de la definición de `g` no está bien formada, ya que no es posible aplicar `(+)` sobre un caracter.

El análisis de los escritos puede dividirse en dos fases: Análisis sintáctico, para chequear la corrección sintáctica de las expresiones y análisis de tipo, para chequear que todas las expresiones tienen un tipo correcto.

3.3.1 Información de tipo

Además de las definiciones de función, en los escritos se puede incluir información de tipo mediante una expresión de la forma A::B para indicar al sistema que A es de tipo B. Por ejemplo:

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

La primera línea indica que la función cuadrado es del tipo "*función que toma un entero y devuelve un entero*".

Aunque no sea obligatorio incluir la información de tipo, sí es una buena práctica, ya que el Haskell chequea que el tipo declarado coincide que el tipo inferido por el sistema a partir de la definición, permitiendo detectar errores de tipos.

3.3.2 Tipos predefinidos

En esta sección se indican los principales tipos predefinidos del sistema Haskell, éstos se podrían clasificar en: **tipos *básicos,*** cuyos valores se toman como primitivos, por ejemplo, Enteros, Flotantes, Caracterses y Booleanos; y **tipos *compuestos***, cuyos valores se construyen utilizando otros tipos, por ejemplo, listas, funciones y tuplas.

Booleanos

Se representan por el tipo `"Bool"` y contienen dos valores: `"True"` y `"False"`. El *standar prelude* incluye varias funciones para manipular valores booleanos: `(&&)`, `(||)` y `not`.

`x && y`	es True si y sólo si `x` e `y` son True
`x || y`	es True si y sólo si `x` ó `y` ó ambos son True
`not x`	es el valor opuesto de `x` (`not True = False`, `not False = True`)

También se incluye una forma especial de expresión condicional que permite seleccionar entre dos alternativas dependiendo de un valor booleano:

```
if exp_b then x else y
```

Si la expresión booleana `exp_b` es `True` devuelve `x`, si es `False`, devuelve `y`. Obsérvese que una expresión de ese tipo sólo es aceptable si `exp_b` es de tipo `Bool` y `x` e `y` son del mismo tipo.

Enteros

Representados por el tipo `"Int"`, se incluyen los enteros positivos y negativos tales como el `-273`, el `0` ó el `383`. Como en muchos sistemas, el rango de los enteros utilizables está restringido. También se puede utilizar el tipo `Integer` que denota enteros sin límites superior ni inferior.

En el *standar prelude* se incluye un amplio conjunto de operadores y funciones que manipulan enteros:

`(+)`	suma.
`(*)`	multiplicación.
`(-)`	substracción.
`(^)`	potenciación.
`negate`	menos unario (la expresión "`-x`" se toma como "`negate x`")
`div`	división entera " "
rem	resto de la división entera. Siguiendo la ley: `(x `div` y)*y + (x `rem` y) == x`
`mod`	módulo, como `rem` sólo que el resultado tiene el mismo signo que el divisor.
`odd`	devuelve True si el argumento es impar
`even`	devuelve True si el argumento es par.
`gcd`	máximo común divisor.
`lcm`	mínimo común múltiplo.
`abs`	valor absoluto
`signum`	devuelve `-1`, `0` o `1` si el argumento es negativo, cero ó positivo, respectivamente.

Ejemplos:

```
3^4 == 81,        7 `div` 3 == 2,     even 23 == False
7 `rem` 3 == 1,   -7 `rem` 3 == -1,   7 `rem` -3 == 1
```

```
7 `mod` 3 == 1,  -7 `mod` 3 == 2,   7 `mod` -3 == -2
gcd 32 12 == 4,  abs (-2) == 2,     signum 12 == 1
```

Flotantes

Representados por el tipo `"Float"`, los elementos de este tipo pueden ser utilizados para representar fraccionarios así como cantidades muy largas o muy pequeñas. Sin embargo, tales valores son sólo aproximaciones a un número fijo de dígitos y pueden aparecer errores de redondeo en algunos cálculos que empleen operaciones en punto flotante. Un valor numérico se toma como un flotante cuando incluye un punto en su representación o cuando es demasiado grande para ser representado por un entero. También se puede utilizar notación científica; por ejemplo `1.0e3` equivale a `1000.0`, mientras que `5.0e-2` equivale a `0.05`.

El *standar prelude* incluye también múltiples funciones de manipulación de flotantes: `pi, exp, log, sqrt, sin, cos, tan, asin, acos, atan`, etc.

Caracteres

Representados por el tipo `"Char"`, los elementos de este tipo representan caracteres individuales como los que se pueden introducir por teclado. Los valores de tipo caracter se escriben encerrando el valor entre comillas simples, por ejemplo `'a', '0', '.'` y `'Z'`. Algunos caracteres especiales deben ser introducidos utilizando un código de escape; cada uno de éstos comienza con el caracter de barra invertida `(\)` , seguido de uno o más caracteres que seleccionan el caracter requerido. Algunos de los más comunes códigos de escape son:

`'\\'`	barra invertida
`'\''`	comilla simple
`'\"'`	comilla doble
`'\n'`	salto de línea
`'\b' or '\BS'`	*backspace* (espacio atrás)
`'\DEL'`	borrado
`'\t' or '\HT'`	tabulador
`'\a' or '\BEL'`	alarma (campana)
`'\f' or '\FF'`	alimentación de papel

Otros códigos de escape serían:

`'\^c'` Carácter de control, `c` puede ser:
`"@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"`
Ejemplo, `'\^A'` representa `control-A`

`'\numero'` Caracter ASCII de valor *numero*.

`'\onumero'` Caracter ASCII cuyo valor viene dado por el octal *numero*.

`'\xnumero'` Caracter ASCII de valor dado por el hexadecimal *numero*.

`'\nombre'` Caracter de control con nombre ASCII. *nombre* debe ser uno de los nombres ASCII estándar, por ejemplo `'\DC3'`

En contraste con algunos lenguajes comunes (como el C, por ejemplo), los valores de tipo Char son completamente distintos de los enteros. Sin embargo, el *standar prelude* proporciona las funciones `toEnum` y `fromEnum` que permiten realizar la conversión. Por ejemplo, la siguiente función convierte mayúsculas en minúsculas.

```
mayusc::Char->Char
mayusc c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
```

Funciones

Si *a* y *b* son dos tipos, entonces *a->b* es el tipo de una función que toma como argumento un elemento de tipo *a* y devuelve un valor de tipo *b.*

Las funciones en Haskell son objetos de primera clase. Pueden ser argumentos o resultados de otras funciones o ser componentes de estructuras de datos. Esto permite simular mediante funciones de un único argumento, funciones con múltiples argumentos.

Considérese, por ejemplo, la función de `suma (+)`. En matemáticas se toma la suma como una función que toma una pareja de enteros y devuelve un entero. Sin embargo, en Haskell, la función `suma` tiene el tipo:

```
(+)::Int->(Int->Int)
```
[6]

`(+)` es una función de un argumento de tipo `Int` que devuelve una función de tipo `Int->Int`. De hecho `"(+) 5"` denota una función que toma un entero y devuelve dicho entero más 5. Este proceso se denomina *currificación*[7] y permite reducir el número de paréntesis necesarios para escribir expresiones. De hecho, no es necesario escribir `f(x)` para denotar la aplicación del argumento `x` a la función `x`, sino simplemente `f x.`

Listas

Si `a` es un tipo cualquiera, entonces `[a]` representa el tipo de listas cuyos elementos son valores de tipo `a.`

Hay varias formas de escribir expresiones de listas:

- La forma más simple es la lista vacía, representada mediante `[]`.

[6]Se podría escribir simplemente `(+)::Int->Int->Int,` puesto que el operador `->` es asociativo a la derecha.
[7]En honor de *Haskell B. Curry*

- Las listas no vacías pueden ser construidas enunciando explícitamente sus elementos (por ejemplo, `[1,3,10]`) o añadiendo un elemento al principio de otra lista utilizando el operador de construcción (:). Estas notaciones son equivalentes:

```
[1,3,10] = 1:[3,10] = 1:(3:[10]) = 1:(3:(10:[]))
```

El operador (:) es asociativo a la derecha, de forma que `1:3:10:[]` equivale a `(1:(3:(10:[])))`, una listaa cuyo primer elemento es 1, el segundo 3 y el último 10.

El *standar prelude* incluye un amplio conjunto de funciones de manejo de listas, por ejemplo:

`length xs`	devuelve el número de elementos de xs
`xs ++ ys`	devuelve la lista resultante de concatenar xs e ys
`concat xss`	devuelve la lista resultante de concatenar las listas de xss
`map f xs`	devuelve la lista de valores obtenidos al aplicar la función *f* a cada uno de los elementos de la lista xs.

Ejemplos:

```
? length [1,3,10]
3

? [1,3,10] ++ [2,6,5,7]
[1, 3, 10, 2, 6, 5, 7]

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

? map fromEnum ['H', 'o', 'l', 'a']
[104, 111, 108, 97]

?
```

Obsérvese que todos los elementos de una lista deben ser del mismo tipo. La expresión `['a',2,False]` no está permitida en Haskell

Cadenas

Una cadena es tratada como una lista de caracteres y el tipo `"String"` se toma como una abreviación de `"[Char]"`. Las cadenas pueden ser escritas como secuencias de caracteres encerradas entre comillas dobles. Todos los códigos de escape utilizados para los caracteres, pueden utilizarse para las cadenas.

```
? "hola"
hola

?
```

Además, las cadenas pueden contener la secuencia de escape `"\&"` que puede ser utilizada para separar pares de caracteres dentro una cadena, por ejemplo:

`"\123h"` representa la cadena `['\123', 'h']`
`"\12\&3h"` representa la cadena `['\12', '3', 'h']`

Puesto que las cadenas son representadas como listas de caracteres, todas las funciones del *standar prelude* para listas pueden ser utilizadas también con cadenas:

```
? length "Hola"
4

? "Hola, " ++ "amigo"
Hola, amigo

? concat ["super","cali","fragi","listico"]
supercalifragilistico

? map fromEnum "Hola"
[104, 111, 108, 97]

?
```

Tuplas

Si `t1, t2, ..., tn` son tipos y `n>=2`, entonces hay un tipo de n-tuplas escrito `(t1, t2, ..., tn)` cuyos elementos pueden ser escritos también como `(x1, x2, ..., xn)` donde cada `x1, x2, ..., xn` tiene tipos `t1,t2, ..., tn` respectivamente.

Ejemplo:
```
(1, [2], 3)  :: (Int, [Int], Int)
('a', False)  :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

Obsérvese que, a diferencia de las listas, los elementos de una tupla pueden tener tipos diferentes. Sin embargo, el tamaño de una tupla es fijo.

En determinadas aplicaciones es útil trabajar con una tupla especial con 0 elementos denominada tipo unidad. El tipo unidad se escribe como () y tiene un único elemento que es también ().

3.4.-Definiciones de función

3.4.1 Encaje de patrones simple

La declaración de una función `f` está formada por un conjunto de ecuaciones con el formato:

```
f <pat1> <pat2> . . . <patn> = <expresion>
```
[8]

Donde cada una de las expresiones `<pat1> <pat2> . . . <patn>` representa un argumento de la función y se denominado un **patrón.** El número `n` de argumentos se denomina **aridad.** Si `f` fuese definida por más de una ecuación, entonces éstas deben definirse juntas y cada una debe tener la misma aridad.

Cuando una función está definida mediante más de una ecuación, será necesario evaluar una o más argumentos de la función para determinar cuál de las ecuaciones aplicar. Este proceso se denomina **encaje de patrones.** En los ejemplos anteriores se utilizó el patrón más simple: una variable. Como ejemplo, considérese la definición de factorial:

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

Si se desea evaluar la expresión `"fact 6"` es necesario encajar la expresión `"6"` con el patrón `"n"` y luego evaluar la expresión obtenida a partir de `"product [1..n]"` substituyendo la `"n"` con el `"6"`.

En los patrones se pueden utilizar constantes, como en la siguiente definición de la función `"not"` tomada del *standar prelude:*

```
not True  = False
not False = True
```

Para determinar el valor de una expresión de la forma `"not b"`, se debe evaluar antes la expresión `"b"`. Si el resultado es `"True"` entonces se utiliza la primera ecuación y el valor de `"not b"` será `"False"`. Si el valor de `"b"` es `"False"`, entonces se utiliza la segunda ecuación y `"not b"` será `"True"`.

Otros tipos de patrones útiles serían:

Anónimos: Se representan por el caracter `(_)` y encajan con cualquier valor, pero no es posible referirse posteriormente a dicho valor. Ejemplo:

```
cabeza (x:_) = x
cola   (_:xs) = xs
```

Patrones con nombre: Para poder referirnos al valor que está encajando, por ejemplo, en lugar de definir `f` como:

```
f (x:xs) = x:x:xs
```

Podría darse un nombre a `x:xs` mediante un patrón con nombre:

```
f p@(x:xs) = x:p
```

[8] Si `f` fuese un operador sería `<pat1> f <pat2> = <expresion>`

Patrones n+k: Encajan con un valor entero mayor o igual que `k`. El valor referido por la variable `n` es el valor encajado menos `k`. Ejemplo:

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

En Haskell, el nombre de una variable no puede utilizarse más de una vez en la parte izquierda de cada ecuación en una definición de funció. Así, el siguiente ejemplo:

```
son_iguales x x = True
son_iguales x y = False
```

no será aceptado por el sistema. Podría ser introducido mediante `if`:

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

3.4.2 Ecuaciones con guardas

Cada una de las ecuaciones de una definición de función podría contener **guardas** que requieren que se cumplan ciertas condiciones sobre los valores de los argumentos.

```
minimo x y   | x <= y    = x
             | otherwise = y
```

En general una ecuación con guardas toma la forma:

```
f x1 x2 ... xn | condicion1  =  e1
               | condicion2  =  e2
               .
               .
               | condicionm  =  em
```

Esta ecuación se utiliza para evaluar cada una de las condiciones por orden hasta que alguna de ellas sea `"True"`, en cuyo caso, el valor de la función vendrá dado por la expresión correspondiente en la parte derecha del signo `"="`. En Haskell, la variable `"otherwise"` evalúa a `"True"`. Por lo cual, escribir `"otherwise"` como una condición significa que la expresión correspondiente será siempre utilizada si no se cumplió ninguna condición previa.

3.4.3 Definiciones locales

Las definiciones de función podrían incluir definiciones locales para variables que podrían en guardas o en la parte derecha de una ecuación.

Considérese la siguiente función que calcula el número de raíces diferentes de una ecuación cuadrática de la forma $ax^2 + bx + c = 0$

```
numeroDeRaices a b c | discr>0   =  2
                     | discr==0  =  1
```

```
                | discr<0   =  0
                  where discr = b*b - 4*a*c
```

Las definiciones locales pueden también ser introducidas en un punto arbitrario de una expresión utilizando una expresión de la forma:

```
let <decls> in <expr>
```

Por ejemplo:

```
? let x = 1 + 4 in x*x + 3*x + 1
41

? let p x = x*x + 3*x + 1  in  p (1 + 4)
41

?
```

3.5.-Notaciones especiales

3.5.1 Secuencias aritméticas

La notación de las secuencias aritméticas permite generar una gran cantidad de listas útiles. Existen cuatro formas de expresar secuencias aritméticas:

`[m..]` Produce una lista (potencialmente infinita) de valores que comienzan con `m` y se incrementan en pasos simples.

Ejemplo:

```
[1..] = [1, 2, 3, 4, 5, 6, 7, 8, 9, etc...
```

`[m..n]` Produce la lista de elementos desde `m` hasta `n`, con incrementos en pasos simples. Si `m` es menor que `n` devuelve la lista vacía.

Ejemplos:

```
[-3..3] = [-3, -2, -1, 0, 1, 2, 3]
[1..1]  = [1]
[9..0]  = []
```

`[m,m'..]` Produce la lista (potencialmente infinita) de valores cuyos dos primeros elementos son `m` y `m'`. Si `m` es menor que `m'` entonces los siguientes elementos de la lista se incrementan en los mismos pasos. Si `m` es mayor que `m'` entonces los incrementos serán negativos. Si son iguales se obtendrá una lista *infinita* cuyos elementos serán todos iguales.

Ejemplos:

```
[1,3..] = [1, 3, 5, 7, 9, 11, 13, etc...
[0,0..] = [0, 0, 0, 0, 0, 0, 0, etc...
```

```
[5,4..] = [5, 4, 3, 2, 1, 0, -1, etc...
```

`[m,m'..n]` Produce la lista de elementos `[m,m',..]` hasta el valor límite `n`. Los incrementos vienen marcados por la diferencia entre `m` y `m'`.

Ejemplos:

```
[1,3..12] = [1, 3, 5, 7, 9, 11]
[0,0..10] = [0, 0, 0, 0, 0, 0, 0, etc...
[5,4..1]  = [5, 4, 3, 2, 1]
```

Las secuencias no están únicamente restringidas a enteros, pueden emplearse con elementos enumerados[9] como caracteres, flotantes, etc.

Ejemplo:

```
? ['0'..'9'] ++ ['A'..'Z']
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ

? [1.2, 1.35 .. 2.00]
[1.2, 1.35, 1.5, 1.65, 1.8, 1.95]
```

3.5.2 Listas por comprehensión

La notación de listas por comprehensión permite declarar de forma concisa una gran cantidad de iteraciones sobre listas. Esta notación está adaptada de la teoría de conjuntos de Zermelo-Fraenkel[10]. Sin embargo en Haskell se trabaja con listas, no con conjuntos. El formato básico de la definición de una lista por comprehensión es:

```
[ <expr> | <cualif_1>, <cualif_2> . . . <cualif_n> ]
```

Cada `<cualif_i>` es un cualificador. Existen dos tipos:

- **Generadores**: Un cualificador de la forma `pat<-exp` es utilizado para extraer cada elemento que encaje con el patrón pat de la lista exp en el orden en que aparecen los elementos de la lista. Un ejemplo simple sería la expresión:

```
? [x*x | x <- [1..10]]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- **Filtros**: Una expresión de valor booleano, el significado de una lista por comprehensión con un único filtro podría definirse como:

```
[e |condicion ]  =  if condition then [e] else []
```

Esta forma de lista por comprehensión resulta útil en combinación con generadores, Ejemplo:

[9]Los elementos que se pueden utilizar deben ser de tipos que sean instancias de la clase `Enum`. Véase sección 9.

[10]En otros contextos, se conoce como expresiones-ZF

```
? [x*x |x<-[1..10], even x ]
[4,16,36,64,100]
```

Si aparecen varios cualificadores hay que tener en cuenta que:

- Las variables generadas por los cualificadores posteriores varían más rápidamente que las generadas por los cualificadores anteriores:

```
? [ (x,y) | x<-[1..3], y<-[1..2] ]
[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]

?
```

- Los cualificadores posteriores podrían utilizar los valores generados por los anteriores:

```
? [ (x,y) | x<-[1..3], y<-[1..x]]
[(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

?
```

- Las variables definidas en cualificadores posteriores ocultan las variables definidas por cualificadores anteriores. Las siguientes expresiones son listas definidas por comprehensión de forma válida. Sin embargo, no es una buena costumbre reutilizar los nombres de las variables ya que dificulta la comprensión de los programas.

```
? [ x | x<-[[1,2],[3,4]], x<-x ]
[1, 2, 3, 4]

? [ x | x<-[1,2], x<-[3,4] ]
[3, 4, 3, 4]

?
```

- Un cambio en el orden de los cualificadores puede tener un efecto directo en la eficiencia. Los siguientes ejemplos producen el mismo resultado, pero el primero utiliza más reducciones debido a que repite la evaluación de "`even x`" por cada valor posible de "y".

```
? [ (x,y) | x<-[1..3], y<-[1..2], even x ]
[(2,1), (2,2)]
(110 reductions, 186 cells)

? [ (x,y) | x<-[1..3], even x, y<-[1..2] ]
[(2,1), (2,2)]
(62 reductions, 118 cells)

?
```

3.5.3 Expresiones *lambda*

Además de las definiciones de función con nombre, es posible definir y utilizar funciones sin necesidad de darles un nombre explícitamente mediante expresiones *lambda* de la forma:

```
\ <patrones atómicos> -> <expr>
```
[11]

Esta expresión denota una función que toma un número de parámetros (uno por cada patrón) produciendo el resultado especificado por la expresión `<expr>`. Por ejemplo, la expresión:

```
(\x->x*x)
```

representa la función que toma un único argumento entero 'x' y produce el cuadrado de ese número como resultado. Otro ejemplo sería la expresión

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

que toma dos argumentos enteros y devuelve su suma. Esa expresión es equivalente al operador (+):

```
? (\x y->x+y) 2 3
5
```

3.5.4 Expresiones *case*

Una expresión *case* puede ser utilizada para evaluar una expresión y, dependiendo del resultado, devolver uno de los posibles valores.

```
paridad x = case  (x mod 2) of
                   0 -> "par"
                   1 -> "impar"
```

Combinando expresiones *case* con las expresiones *lambda* de la sección anterior, se puede traducir cualquier declaración de función en una simple ecuación de la forma
`<nombreFunción> = <expr>.`

Por ejemplo, la función stándar `map`, cuya definición se escribe normalmente como:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

podría también definirse mediante una única ecuación:

```
map = \f xs -> case xs of
                 []     -> []
                 (y:ys) -> f y : map f ys
```

Este tipo de traducción es utilizado en la implementación de muchos lenguajes de programación funcional.

3.5.5 Secciones

[11]Esta es una generalización de las expresiones utilizadas en el cálculo de funciones lambda de Church. El caracter `\' se ha elegido por su parecido con la letra griega *lambda* '*l* '

Como ya se ha indicado, la mayoría de las funciones de más de un argumento son tratadas como funciones de un único argumento, cuyo resultado es una función que puede aplicarse al resto de argumentos. De esa forma, "(+) 5" denota una función que toma un argumento entero `"n"` y devuelve el valor entero `"5+n".` Las funciones de este tipo son tan utilizadas que el sistema proporciona una notación especial para ellas. Si `"e"` es una expresión atómica y `"*"` es un operador infijo, entonces también son funciones `(e *)` y `(* e)` conocidas como "*secciones del operador* `*"`. Su definición es:

```
(e *) x  = e * x
(* e) x  = x * e
```

Ejemplos:
- `(1+)` es la función sucesor que devuelve su argumento más 1.
- `(1.0/)` es la función *inverso*.
- `(/2)` es la función *mitad*.
- `(:[])` es la función que convierte un elemento simple en una lista con un único elemento que lo contiene

Las expressiones `"(e *)"` y `"(* e)"` son tratadas como abreviaciones de `"(*) e"` y `"flip (*) e"` respectivamente, donde `"flip"` se define como:

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

Existe un caso especial importante: Una expresión de la forma `(-e)` es interpretada como `"negate e",` no como la sección que resta el valor de `"e"` de su argumento.

3.5.6 Disposición del código

El lector se habrá preguntado cómo es posible evitar la utilización de separadores que marquen el final de una ecuación, una declaración, etc. Por ejemplo, dada la siguiente expresión:

```
ejemplo x y z = a + b
      where a = f x y
            b = g z
```

¿ Cómo sabe el sistema *Haskell* que no debe analizarla como:

```
ejemplo x y z   = a + b
      where a   = f x y
            y b = g z       ?
```

La respuesta es que el *Haskell* utiliza una sintaxis bidimensional denominada ***espaciado** (layout)* que se basa esencialmente en que las declaraciones están alineadas por columnas. En el ejemplo anterior, obsérvese que `a` y `b` comenzaban en la misma columna. Las reglas del espaciado son bastante intuitivas y podrían resumirse en:

1.- El siguiente caracter de cualquiera de las palabras clave `where`, `let`, o `of` es el que determina la columna de comienzo de declaraciones en las expresiones `where`, `let`, o `case` correspondientes. Por tanto podemos comenzar las declaraciones en la misma línea que la palabra clave, en la siguiente o siguientes.

2.- Es necesario asegurarse que la columna de comienzo dentro de una declaración está más a la derecha que la columna de comienzo de la siguiente cláusula. En caso contrario, habría ambigüedad, ya que el final de una declaración ocurre cuando se encuentra algo a la izquierda de la columna de comienzo.

El espaciado es una forma sencilla de agrupamiento que puede resultar bastante útil. Por ejemplo, la declaración anterior sería equivalente a:

```
ejemplo x y z = a + b
      where { a = f x y ;
              b = g z }
```

3.6.-Tipos definidos por el usuario

3.6.2 Sinónimos de tipo

Los sinónimos de tipo se utilizan para proporcionar abreviaciones para expresiones de tipo aumentando la legibilidad de los programas. Un sinónimo de tipo es introducido con una declaración de la forma:

```
type Nombre a1 ... an = expresion_Tipo
```

donde

- `Nombre` es el nombre de un nuevo constructor de tipo de aridad `n>=0`
- `a1,..., an` son variables de tipo diferentes que representan los argumentos de `Nombre`
- `expresion_Tipo` es una expresión de tipo que sólo utiliza como variables de tipo las variables `a1,..., an.`

Ejemplo:

```
type Nombre = String
type Edad   = Integer
type String = [Char]
type Persona = (Nombre, Edad)

tocayos::Persona -> Persona -> Bool
tocayos (nombre,_) (nombre',_) = n == nombre'
```

3.6.1 Definiciones de tipos de datos

Aparte del amplio rango de tipos predefinidos, en Haskell también se permite definir nuevos tipos de datos mediante la sentencia `data`. La definición de nuevos tipos de datos aumenta la seguridad de los programas ya que el sistema de inferencia de tipos distingue entre los tipos definidos por el usuario y los tipos predefinidos.

Tipos Producto

Se utilizan para construir un nuevo tipo de datos formado a partir de otros. Ejemplo:

```
data Persona = Pers Nombre Edad

juan::Persona
juan = Pers "Juan Lopez" 23
```

Se pueden definir funciones que manejen dichos tipos de datos:

```
esJoven:: Persona -> Bool
esJoven (Pers _ edad) = edad < 25

verPersona::Persona -> String
verPersona (Pers nombre edad) = "Persona, nombre " ++ nombre ++ ", edad: " ++
show edad
```

También se puden dar nombres a los campos de un tipo de datos producto:

```
data = Datos { nombre::Nombre, dni::Integer, edad:Edad }
```

Los nombres de dichos campos sirven como funciones selectoras del valor correspondiente. Por ejemplo:

```
tocayos:: Persona -> Persona -> Bool
tocayos p p' = nombre p == nombre p'
```

Obsérvese la diferencia de las tres definiciones de Persona

1.- Como sinónimo de tipos:

```
type Persona = (Nombre, Edad)
```

No es un nuevo tipo de datos. En realidad, si se define

```
type Direccion = (Nombre, Numero)
type Numero = Integer
```

El sistema no daría error al aplicar una función que requiera un valor de tipo persona con un valor de tipo Dirección. La única ventaja (discutible) de la utilización de sinónimos de tipos de datos podría ser una mayor eficiencia (la definición de un nuevo tipo de datos puede requerir un mayor consumo de recursos.

2.- Como Tipo de Datos

```
data Persona = Pers Nombre Edad
```

El valor de tipo Persona es un nuevo tipo de datos y, si se define:

```
type Direccion = Dir Nombre Numero
```

El sistema daría error al utilizar una dirección en lugar de una persona. Sin embargo, en la definición de una función por encaje de patrones, es necesario conocer el número de campos que definen una persona, por ejemplo:

```
esJoven (Pers _ edad) = edad < 25
```

Si se desea ampliar el valor persona añadiendo, por ejemplo, el "dni", todas las definiciones que trabajen con datos de tipo Persona deberían modificarse.

3.- Mediante campos con nombre:

```
data Persona = Pers { nombre::Nombre, edad::Edad }
```

El campo sí es un nuevo tipo de datos y ahora no es necesario modificar las funciones que trabajen con personas si se amplían los campos.

Tipos Enumerados

Se puede introducir un nuevo tipo de datos enumerando los posibles valores.

Ejemplos:

```
data Color       = Rojo | Verde | Azul
data Temperatura = Frio | Caliente
data Estacion    = Primavera | Verano | Otonio | Invierno
```

Se pueden definir funciones simples mediante encaje de patrones:

```
tiempo :: Estacion -> Temperatura
tiempo Primavera = Caliente
tiempo Verano    = Caliente
tiempo _         = Frio
```

Las alternativas pueden contener a su vez productos:

```
data Forma = Circulo        Float
           | Rectangulo Float Float


area :: Forma -> Float
area (Circulo radio)          = pi * r * r
area (Rectangulo base altura) = base * altura
```

Tipos Recursivos

Los tipos de datos pueden autorreferenciarse consiguiendo valores recursivos, por ejemplo:

```
data Expr = Lit Integer
```

```
            | Suma Expr  Expr
            | Resta Expr Expr

eval (Lit n) = n
eval (Suma  e1 e2) = eval e1 + eval e2
eval (Resta e1 e2) = eval e1 * eval e2
```

También pueden definirse tipos de datos polimórficos. El siguiente ejemplo define un tipo que representa árboles binarios:

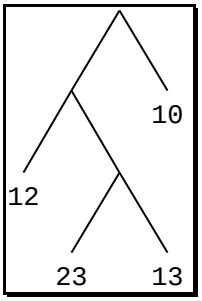
```
data Arbol a = Hoja a | Rama (Arbol a) (Arbol a)
```

Por ejemplo,

```
a1 = Rama (Rama (Hoja 12) (Rama (Hoja 23) (Hoja 13)) (Hoja 10)
```

tiene tipo

```
Arbol Integer
```



y representa el árbol binario de la figura.

Otro ejemplo, sería un árbol, cuyas hojas fuesen listas de enteros o incluso árboles.

```
a2 ::Arbol [Integer]
a2 = Rama (Hoja [1,2,3] Hoja [4,5,6])

a3 :: Arbol (Arbol Int)
a3 = Rama (Hoja a1) (Hoja a1)
```

A continuación se muestra una función que calcula la lista de nodos hoja de un árbol binario:

```
hojas              :: Arbol a -> [a]
hojas (Hoja h)           = [h]
hojas (Rama izq der)     = hojas izq ++ hojas der
```

Utilizando el árbol binario anterior como ejemplo:

```
? hojas a1
[12, 23, 13, 10]

? hojas a2
[[1,2,3], [4,5,6]]
```

Nuevos Tipos de Datos a partir de tipos existentes

En ciertas ocasiones puede desearse utilizar un tipo de datos ya existente (por motivos de eficiencia) pero etiquetarlo para que distinguirlo del tipo de datos original[12]. Para elloe, se puede emplear la declaración `newtype`.

[12] Recuérdese que los sinónimos de tipos no distinguen entre el tipo original y el sinónimo.

```
newtype Persona = Per (Nombre, Edad)

esJoven (Per (_, edad)) = edad < 25
```

Internamente, el tipo persona se representa como una tupla con dos valores, pero un valor de tipo Persona se distingue mediante la etiqueta `"Per"`.

3.7.-Entrada/Salida

Hasta ahora, todas las funciones descritas tomaban sus argumentos y devolvían un valor sin interactuar con el exterior. A la hora de realizar programas *"reales"* es necesario que éstos sean capaces de almacenar resultados y leer datos de ficheros, realizar preguntas y obtener respuestas del usuario, etc.

Una de las principales ventajas del lenguaje *Haskell* es que permite realizar las tareas de Entrada/Salida de una forma puramente funcional, manteniendo la transparencia referencial y sin efectos laterales. Para ello, a partir de la versión 1.3 se utiliza una **mónada** de Entrada/Salida.

El concepto de mónada tiene su origen en una rama de las matemáticas conocida como Teoría de la Categoría. No obstante, desde el punto de vista del programador resulta más sencillo considerar una mónada como un tipo abstracto de datos. En el caso de la mónada de Entrada/Salida, los valores abstractos son las acciones primitivas correspondientes a operaciones de Entrada/Salida convencionales. Ciertas operaciones especiales permiten componer acciones de forma secuencial (de forma similar al punto y coma de los lenguajes imperativos). Esta abstracción permite ocultar el estado del sistema (es decir, el estado del *mundo* externo) al programador que accede a través de funciones de composición o primitivas.

Una expresión de tipo `IO a` denota una computación que puede realizar operaciones de Entrada/Salida y devolver un resultado de tipo `a`.

A continuación se declara una sencilla función que muestra por pantalla la cadena `"Hola Mundo"`:

```
main::IO()
main = print "Hola, mundo!"
```

La función `main` tiene tipo `IO ()` indicando que realiza Entrada/Salida y no devuelve ningún valor. Esta función tiene un significado especial cuando el lenguaje es compilado, puesto que es la primera función evaluada por el sistema. En esta ocasión, se utiliza la función `print` declarada en el *Standar prelude* que se encargará de imprimir su argumento en la salida estándar.

3.7.1 Funciones básicas de Entrada/Salida

A continuación se muestran algunas de las funciones básicas de Entrada/salida predefinidas:

`putChar::Char->IO ()`	Imprime un caracter
`getChar::IO Char`	Lee un caracter
`putStr::String->IO ()`	Imprime una cadena
`putStrLn::String->IO ()`	Imprime una cadena y un salto de línea
`print::Show a => a ->IO ()`	Imprime un valor de cualquier tipo imprimible (perteneciente a la clase `Show`)
`getLine::IO String`	Lee una cadena de caracteres hasta que encuentra el salto de línea
`getContents::IO String`	Lee en una cadena toda la entrada del usuario (esta cadena será potencialmente infinita y se podrá procesar gracias a la evaluación perezosa)
`interact::(String->String)->IO ()`	Toma como argumento una función que procesa una cadena y devuelve otra cadena. A dicha función se le pasa la entrada del usuario como argumento y el resultado devuelto se imprime.
`writeFile::String->String->IO ()`	Toma como argumentos el nombre de un fichero y una cadena; escribe dicha cadena en el fichero correspondiente.
`appendFile::String->String->IO ()`	Toma como argumentos el nombre de un fichero y una cadena; añade dicha cadena al final del fichero correspondiente.
`readFile::String->IO String`	Toma como argumento el nombre de un fichero y devuelve el contenido en una cadena.

A continuación se muestran dos programas sencillos: el primer programa convierte la entrada del usuario a mayúsculas.

```
main = interact (map toUpper)
```

El siguiente programa escribe en el fichero `"tabla.txt"` una tabla con los cuadrados de los 10 primeros números naturales.

```
main = appendFile "tabla.txt" (show [(x,x*x) | x<-[1..10]])
```

3.7.2 Composición de Operaciones de Entrada/Salida

Existen dos funciones de composición de acciones de E/S. La función `>>` se utiliza cuando el resultado de la primera acción no es interesante, normalmente, cuando es `()`. La función `(>>=)` pasa el resultado de la primera acción como un argumento a la segunda acción.

```
(>>=)         ::  IO a    -> (a -> IO b)        -> IO b
(>>)          ::  IO a    -> IO b               -> IO b
```

Por ejemplo

```
main = readFile  "fentrada"                 >>= \cad ->
       writeFile "fsalida" (map toUpper cad) >>
       putStr "Conversion realizada\n"
```

es similar al ejemplo de la sección anterior, salvo que se leen los contenidos del fichero `fentrada` y se escriben, convirtiendo minúsculas en mayúsculas en `fsalida`.

Existe una notación especial que permite una sintaxis con un estilo más imperativo mediante la sentencia `do`. Una versión mejorada del ejemplo anterior, podría reescribirse como:

```
main = do
        putStr "Fichero de Entrada? "
        fentrada <- getLine
        putStr "Fichero de salida? "
        fsalida <- getLine
        cad <- readFile fentrada
        writeFile fsalida (map toUpper cad)
        putStr "Conversion realizada\n"
```

La función `return` se utiliza para definir el resultado de una operación de Entrada/Salida. Por ejemplo, la función `getLine` podría definirse en función de `getChar` utilizando `return` para definir el resultado.

```
getLine :: IO String
getLine = do c <- getChar
          if c == '\n' then return ""
                       else do s <- getLine
                               return (c:s)
```

3.7.3 Control de excepciones

El sistema de incluye un mecanismo simple de control de excepciones. Cualquier operación de Entrada/Salida podría lanzar una excepción en lugar de devolver un resultado. Las excepciones se representan como valores de tipo `IOError`. Mediante la función `userError` el usuario podría lanzar también sus propios errores.
Las excepciones pueden ser lanzadas y capturadas mediante las funciones:

```
fail            ::  IOError -> IO a
catch           ::  IO a    -> (IOError -> IO a) -> IO a
```

La función `fail` lanza una excepción; la función `catch` establece un manejador que recibe cualquier excepción elevada en la acción protegida por él. Una excepción es capturada por el manejador más reciente. Una excepción es capturada por el manejador más reciente. Puesto que los manejadores capturan todas las excepciones (no son selectivos), el programador debe encargarse de propagar las excepciones que no desea manejar. Si una excepción se propaga fuera del sistema, se imprime un error de tipo `IOError`.

3.8.-Sobrecarga

Cuando una función puede utilizarse con diferentes tipos de argumentos se dice que está sobrecargada. La función (+), por ejemplo, puede utilizarse para sumar enteros o para sumar

flotantes. La resolución de la sobrecarga por parte del sistema Haskell se basa en organizar los diferentes tipos en lo que se denominan ***clases de tipos.***

Considérese el operador de comparación `(==)`. Existen muchos tipos cuyos elementos pueden ser comparables, sin embargo, los elementos de otros tipos podrían no ser comparables. Por ejemplo, comparar la igualdad de dos funciones es una tarea computacionalmente intratable, mientras que a menudo se desea comparar si dos listas son iguales. De esa forma, si se toma la definición de la función `elem` que chequea si un elemento pertenece a una lista:

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

Intuitivamente el tipo de la función `elem` debería ser `a->[a]->Bool`. Pero ésto implicaría que la función == tuviese tipo `a->a->Bool`. Sin embargo, como ya se ha indicado, interesaría restringir la aplicación de == a los tipos cuyos elementos son *comparables*.

Además, aunque == estuviese definida sobre todos los tipos, no sería lo mismo comparar la igualdad de dos listas que la de dos enteros.

Las ***clases de tipos*** solucionan ese problema permitiendo declarar qué tipos son instancias de unas clases determinadas y proporcionando definiciones de ciertas operaciones asociadas con cada clase de tipos. Por ejemplo, la clase de tipo que contiene el operador de igualdad se define en el *standar prelude* como:

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

`Eq` es el nombre de la clase que se está definiendo, `(==)` y `(/=)` son dos operaciones simples sobre esa clase. La declaración anterior podría leerse como: *"Un tipo `a` es una instancia de una clase `Eq` si hay una operación `(==)` definida sobre él"*

La restricción de que un tipo `a` debe ser una instancia de una clase `Eq` se escribe `Eq a`. Obsérvese que `Eq a` no es una expresión de tipo sino una restricción sobre el tipo de un objeto `a` (se denomina un **contexto)**. Los contextos son insertados al principio de las expresiones de tipo. Por ejemplo, la operación == sería del tipo:

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

Esa expresión podría leerse como: *"Para cualquier tipo `a` que sea una instancia de la clase `Eq`, `==` tiene el tipo `a->a->Bool`"*. La restricción se propagaría a la definición de `elem` que tendría el tipo:

```
elem:: (Eq a) => a -> [a] -> Bool
```

Las declaraciones de instancias permitirán declarar qué tipos son instancias de una determinada clase. Por ejemplo:

```
instance Eq Int where
```

```
   x == y = intEq x y
```

La definición de == se denomina **método.** `IntEq` es una función primitiva que compara si dos enteros son iguales, aunque podría haberse incluido cualquier otra expresión que definiese la igualdad entre enteros. La declaración se leería como: *"El tipo `Int` es una instancia de la clase `Eq` y el método correspondiente a la operación == se define como ..."*. De la misma forma se podrían crear otras instancias:

```
instance Eq Float where
   x == y = FloatEq x y
```

La declaración anterior utiliza otra función primitiva que compara flotantes para indicar cómo comparar elementos de tipo `Float`. Además, se podrían declarar instancias de la clase `Eq` tipos definidos por el usuario. Por ejemplo, la igualdad entre elementos del tipo `Arbol` definido en la sección 7 :

```
instance (Eq a) => Eq (Arbol a) where
  Hoja a      == Hoja b      = a == b
  Rama i1 d1  == Rama i2 d2  = (i1==i2) && (d1==d2)
  _           == _           = False
```

Obsérvese que el contexto (`Eq a`) de la primera línea es necesario debido a que los elementos de las hojas son comparados en la segunda línea. La restricción adicional está indicando que sólo se podrá comparar si dos árboles son iguales cuando se puede comparar si sus hojas son iguales.

El *standar prelude* incluye un amplio conjunto de clases de tipos. De hecho, la clase `Eq` está definida con una definición ligeramente más larga que la anterior:

```
class Eq a where
 (==), (/=) :: a->a->Bool
  x /= y      = not (x == y)
```

Se incluyen dos operaciones, una para igualdad `(==)` y otra para no igualdad `(/=)`. Se puede observar la utilización de un **método por defecto** para la operación `(/=)`. Si se omite la declaración de un método en una instancia entonces se utiliza la declaración del método por defecto de su clase. Por ejemplo, las tres instancias anteriores podrían utilizar la operación `(/=)` sin problemas utilizando el método por defecto (la negación de la igualdad).

Haskell también permite la *inclusión* de clases. Por ejemplo, podría ser interesante definir una clase `Ord` que *hereda* todas las operaciones de `Eq` pero que, además tuviese un conjunto nuevo de operaciones:

```
class (Eq a) => Ord a where
   (<), (<=), (>=), (>)   :: a->a->Bool
   max, min               :: a->a->a
```

El contexto en la declaración indica que `Eq` es una **superclase** de `Ord` (o que `Ord` es una **subclase** de `Eq`), y que cualquier instancia de `Ord` debe ser también una instancia de `Eq`.

Las inclusiones de clase permiten reducir el tamaño de los contextos: Una expresión de tipo para una función que utiliza operaciones tanto de las clases `Eq` como `Ord` podría utilizar el contexto `(Ord a)` en lugar de `(Eq a,Ord a)`, puesto que `Ord` implica `Eq`. Además, los métodos de las subclases pueden asumir la existencia de los métodos de la superclase. Por ejemplo, la declaración dee `Ord` en el *standar prelude* incluye el siguiente método por defecto:

```
x < y       =     x <=y && x/=y
```

Haskell también permite la **herencia múltiple,** puesto que las clases pueden tener más de una superclase. Los conflictos entre nombres se evitan mediante la restricción de que una operación particular sólo puede ser miembro de una única clase en un ámbito determinado.

En el *Standar Prelude* se definen una serie de clases de tipos de propósito general. En la figura se muestra, a modo de ejemplo, la estructura jerárquica de las clases numéricas predefinidas.

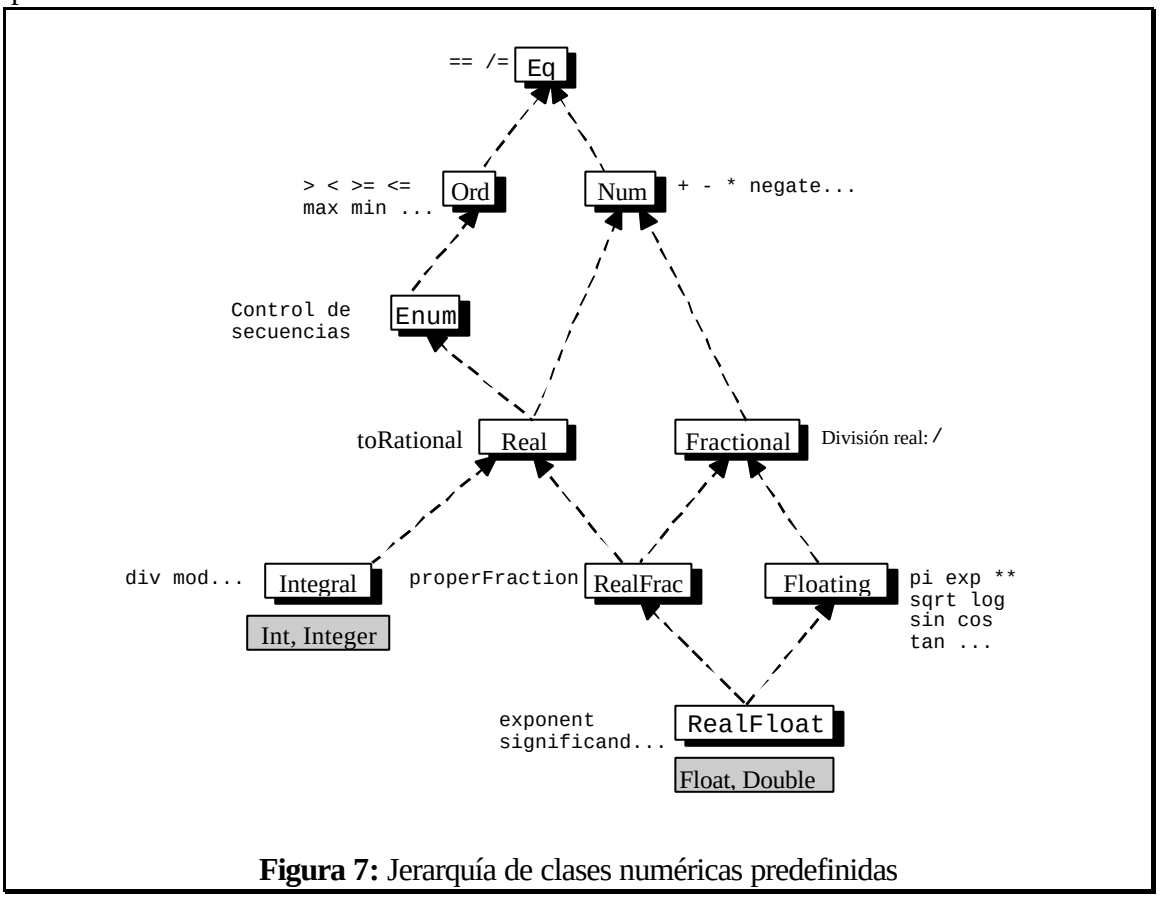


Figura 7: Jerarquía de clases numéricas predefinidas

La clase `Num` proporciona varias operaciones básicas comunes a todos los tipos numéricos; éstos incluyen la suma, resta, negación, multiplicación, y valor absoluto:

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

Obsérvese que `Num` no proporciona un operador de división; se proporcionan dos clases diferentes de operadores de división en dos subclases de `Num` que no se solapan.

La clase `Integral` proporciona la operación de división (`div`) y resto (`rem`), así como los predicados `even` y `odd` (que chequean si un elemento es par o impar, respectivamente). Las instancias estándar de `Integral` son `Integer` (para números enteros no limitados, enteros *grandes*) e `Int` (para enteros limitados, su rango depende de la implementación).

El resto de tipos numéricos están en la clase `Fractional` que proporciona el operador de división (`/`). La subclase `Floating` contiene las funciones trigonométricas, logarítmicas y exponenciales.

La subclase `RealFrac` de `Fractional` y `Real` proporciona una función `properFraction` que descompone un número en su parte real y fraccionaria y una colección de funciones que redondean a valores enteros mediante diferentes reglas:

```
properFraction::(Fractional a, Integral b) => a->(b,a)
truncate, round,
floor,ceiling ::(Fractional a, Integral b) => a -> b
```

La subclase `RealFloat` de `Floating` y `RealFrac` proporcionan algunas funciones especializadas para acceso eficiente a los componentes de un número flotante, el exponente y la mantisa. Los tipos estándar `Float` y `Double` son instancias de esta clase.

Además de las clases numéricas, en el *Standar prelude* se definen otras clases de tipos como las clases `Enum` (cuyos elementos pueden aparecer en secuencias), `Ix` (índices de arrays), `Show` (elementos transformables en una cadena de caracteres y, por tanto, imprimibles), `Read` (elementos que pueden ser leídos de una cadena de caracteres), `Monad` (definición de mónadas), etc.

3.9.-Módulos

A pesar de que en el sistema Hugs no se utiliza compilación separada, se admite la descomposición en módulos de los programas. Un módulo define una colección de valores, tipos de datos, sinónimos de tipo, clases, etc. en un entorno. Cada módulo puede hacer referencia a otros módulos mediante declaraciones `import`, cada una de las cuales especifica el nombre del módulo a importar y las entidades que se importan. Los módulos pueden ser mutuamente recursivos.

A cada módulo se le debe asociar un único nombre (identificadores de *Haskell*) que comienzan por minúscula (ejemplo `Pila`) . Existe un módulo especial, `Prelude`, que es importado por todos los módulos salvo que se indique lo contrario y un conjunto de módulos estándar de librería que pueden ser importados si de desea.

A continuación se muestran dos ejemplos sencillos de la utilización de módulos en *Haskell*. El primer módulo `Pila` declara un tipo abstracto `TPila` con las funciones de acceso `mete` (para meter un elemento en la pila), `saca` (saca un elemento de la pila, devuelve error si está vacía)

, `vacia` (devuelve una pila sin elementos) y `muestra` que representa el contenido de la pila en un `String`

```
module Pila (TPila, mete, saca, vacia, muestra) where

data TPila a = PilaVacia | Apila a (TPila a)

mete::a -> TPila a -> TPila a
mete x p = Apila x p

saca::TPila a ->TPila a
saca (Apila _ p) = p
saca PilaVacia   = error "Sacando elementos de pila vacia"

vacia::TPila a
vacia = PilaVacia

muestra::(Show a)=>TPila a ->String
muestra PilaVacia = " "
muestra (Apila x p) = " " ++ show x ++ muestra p
```

El segundo módulo `Main` importa las entidades exportadas por el módulo `Pila`, declara la función `main` y desarrolla una sencilla aplicación que mete y saca el elemento 1 en una pila a petición del usuario.

```
module Main where
import Pila

menu::IO Char
menu = do
        putStrLn "Programa ejemplo de manejo de pilas"
        putStrLn "   m.- Meter elemento en pila "
        putStrLn "   s.- Sacar elemento de pila "
        putStrLn "   v.- Ver elementos de pila  "
        putStrLn "   f.- Finalizar programa     "
        putStrLn "---------------------------Teclear Opcion: "
        x <- getChar
        if elem x "msfv" then return x
                         else menu

main::IO()
main = bucle vacia

bucle::TPila Int -> IO ()
bucle p = do
           opc <- menu
           putStrLn ("Opcion: " ++ show opc)
           case opc of
            'm' -> bucle (mete 1 p)
            's' -> bucle (saca p)
            'v' -> putStrLn ("Contenido de Pila: " ++ muestra p) >>
                   bucle p
            'f' -> putStrLn "Fin del Programa! "
```

Bibliografía

[Bird97] R. Bird, *Introduction to Functional Programming using Haskell*. Prentice Hall International, 2nd Ed. New York, 1997

[CM95] C. Clack, C. Myers, E. Poon. *Functional Programming with Miranda*. Prentice Hall. 1995

[Cun95] H. C. Cunningham. *Notes on Functional Programming with Gofer*. Technical Report UMCIS-1995-01. Univ. of Mississippi. Junio-1995

[FH88] A.J. Field, P. G. Harrison. *Functional Programming*. Addison Wesley, 1988.

[Has96] J. Peterson, K. Hammond (editors) *Haskell 1.3. A non-strinct, Purely Functional Language*, Mayo 1996

[Hud89] P. Hudak. *Conception, evolution and application of functional programming languages*. ACM Computing Surveys, 21 (3): 359-411, Sept. 1989.

[HF92] P. Hudak, J. H. Fasel. *A Gentle Introduction to Haskell*. ACM SIGPLAN NOTICES, 27 (5), Mayo 1992

[HPW92] P. Hudak, S. Peyton Jones, P. Wadler (editors). *Report on the Programming Language Haskell, a Non-strict Purely Functional Language (version 1.2)* ACM SIGPLAN Notices, 27 (5) Mayo 1992

[Jon91] M.P.Jones. *An Introduction to Gofer*, 1991. Manual distribuido como parte del sistema Gofer a partir de la versión 2.20

[PE93] R. Plasmeijer, Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley. 1993

Indice

Tabla de Contenidos