Universidad Tecnológica Nacional
Facultad Regional Córdoba
Dpto de Ingeniería en Sistemas de Información
Cátedra Paradigmas de Programación

UNIDAD Nº 4:

PARADIGMA FUNCIONAL

2021

EL PRESENTE CAPÍTULO HA SIDO ELABORADO, CORREGIDO Y COMPILADO POR:

DR. ING. CASTILLO, JULIO
ESP. ING. GUZMAN, ANALÍA
ESP. ING. LIGORRIA, LAURA
DR. ING. MARCISZACK, MARCELO

ÚLTIMA REVISIÓN: MAYO 2021

ÍNDICE

OBJETIVOS DE LA UNIDAD

Que el alumno comprenda acabadamente los principios constitutivos y filosóficos que dan origen a este paradigma.

Que el alumno utilice el concepto de funciones para la construcción de programas en la resolución de problemas.

CONTENIDOS ABORDADOS

Concepto de función matemática y su empleo en un lenguaje de programación. La función como bloque de construcción de programas. Concepto de programa en el paradigma funcional. El Cálculo lambda, concepto de funciones de orden superior, manejo y concepto de variables. Funciones recursivas. Manejo de listas. Funciones de orden superior. Sistemas de tipos. Polimorfismo y tipos genéricos. Expresiones lambda.

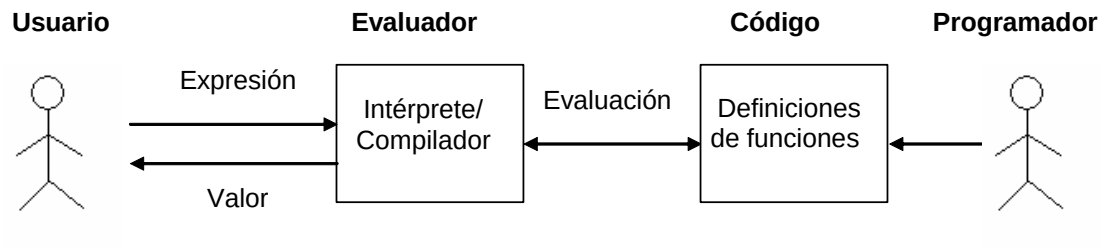
Lenguaje: Haskell

1. PARADIGMA FUNCIONAL

1.1 INTRODUCCIÓN

El paradigma funcional es un paradigma declarativo que se basa en un modelo matemático de composición de funciones. En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el valor deseado, tal como sucede en la composición de funciones matemáticas.

Los programas funcionales expresan mejor qué hay que calcular, y no detallan tanto cómo realizar dicho cálculo a diferencia de lo que ocurre en los programas imperativos.



La característica principal de la programación funcional es que los cálculos se ven como una función matemática que hacen corresponder entradas y salidas.

No existe el concepto de celda de memoria que es asignada o modificada. Más bien, existen valores intermedios que son el resultado de cómputos intermedios, los cuales resultan en valores útiles para los cálculos subsiguientes.

Tampoco existen sentencias imperativas(al menos en un lenguaje de programación funcional puro) y todas las funciones tienen transparencia referencial, es decir, el resultado devuelto por una función sólo depende de los argumentos que se le pasan en esa llamada.

La idea de "repetición" se modela utilizando la recursividad ya que no que no existe el concepto de valor de una variable.

1.2 HISTORIA

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 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 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.

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 la que se decidió formar un comité internacional que diseñase un nuevo lenguaje puramente funcional de propósito general denominado Haskell.

Con el lenguaje **Haskell** se pretendía unificar las características más importantes de los lenguajes funcionales, como las funciones de orden superior, la evaluación perezosa, la inferencia estática de tipos, los tipos de datos definidos por el usuario, el encaje de patrones y las listas por comprensión. 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 comprensión.

Durante casi 10 años aparecieron varias versiones del lenguaje Haskell, hasta que en 1998 se decidió proporcionar una versión estable del lenguaje, que se denominó Haskell98, a la vez que se continuaba la investigación de nuevas características y nuevas extensiones al lenguaje.

A continuación veremos gráficamente la evolución de los más importantes lenguajes de programación:

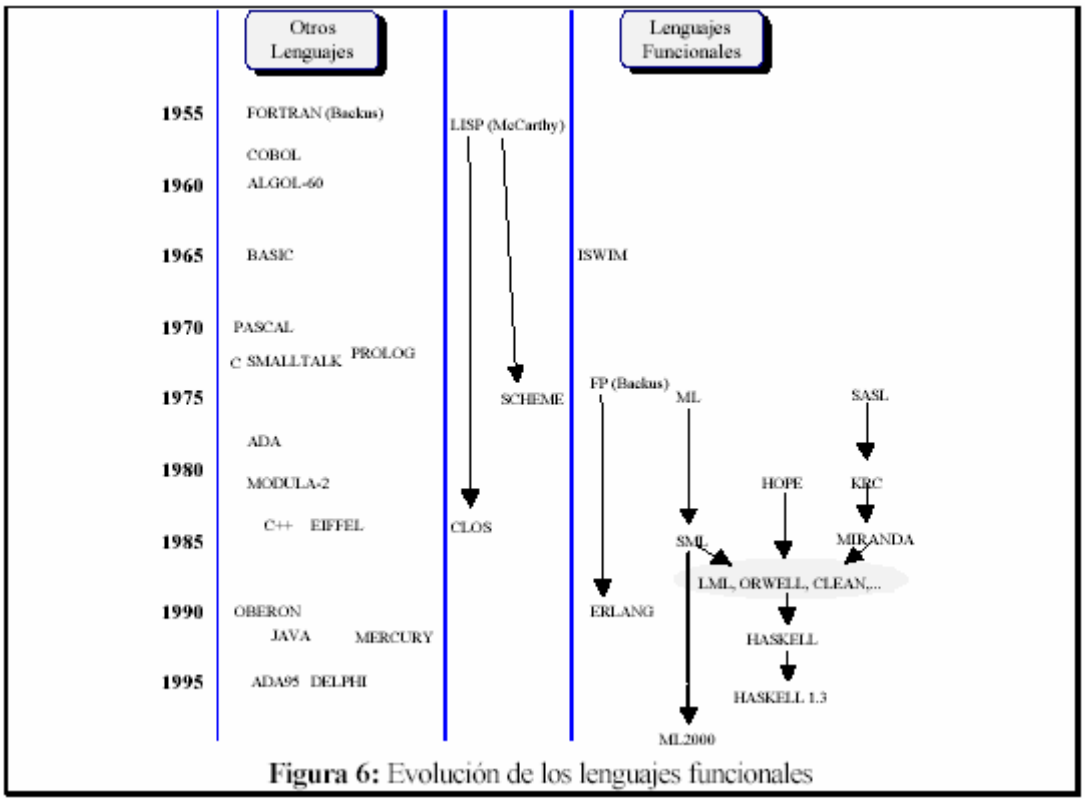


Figura 6: Evolución de los lenguajes funcionales

1.3 CARACTERÍSTICAS

- Transparencia referencial: permite que el valor que devuelve una función esté únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor. De esta manera, los valores resultantes son inmutables. No existe el concepto de cambio de estado.

- Utilización de tipos de datos genéricos: permite aumentar su flexibilidad y realizar unidades de software genéricas, es una forma de implementar el polimorfismo en el paradigma funcional.

- Recursividad: Basada en el principio matemático de inducción, que se ve expresada en el uso de tipos de datos recursivos, como las listas, y funciones recursivas que las operan.

- Posibilidad de tratar a las funciones como datos mediante la definición de funciones de orden superior, que permiten un gran nivel de abstracción y generalidad en las soluciones. El uso correcto de las funciones de orden superior puede mejorar substancialmente la estructura y la modularidad de muchos programas.

- Los lenguajes funcionales relevan al programa de la compleja tarea de gestión de memoria. El almacenamiento se asigna y se inicializa implícitamente, y es recuperado automáticamente por un recolector de la basura (garbage collector). La tecnología para la asignación de memoria y de la recolección de la basura están actualmente bien desarrollados por lo que su costo de funcionamiento es leve.

- La manera de construir abstracciones es a través de funciones, ya que no existen los conceptos de variables o el concepto de cambio de estado.
- Basado en la matemática y en la teoría de funciones.

1.4 VENTAJAS Y LIMITACIONES

Se enumeran a continuación algunas de las ventajas y limitaciones de trabajar con este paradigma.

Ventajas

- Fácil de formular matemáticamente.
- Administración automática de la memoria.
- Simplicidad en el código.
- Rapidez en la codificación de los programas.

Limitaciones

- No es fácilmente escalable.
- Difícil de integrar con otras aplicaciones.
- No es recomendable para modelar lógica de negocios o para realizar tareas de índole transaccionales.

1.5 ÁREAS DE APLICACIÓN

El paradigma funcional tiene diversas áreas de aplicación entre las cuales podemos enumerar a las siguientes:

- Demostraciones de teoremas: Por su naturaleza "funcional" este paradigma es útiles en la demostración automática de teoremas, ya que permite especificar de manera adecuada y precisa problemas matemáticos.
- Creación de compiladores, analizadores sintácticos: Su naturaleza inherentemente recursiva le permite modelar adecuadamente estos problemas.
- Resolver problemas que requieran demostraciones por inducciones.
- En la industria se puede usar para resolver problemas matemáticos complejos.

- Se utiliza en centros de investigaciones y en universidades.

1.6 EJEMPLOS DE IMPLEMENTACIONES

A continuación se detallan algunas implementaciones exitosas en la industria:

- ABN AMRO Amsterdam: Es un banco internacional con sede en Ámsterdam. Para sus actividades de banca de inversión necesita medir el riesgo de ciertos agentes financieros, para ello utiliza Haskell.
- Amgen en Thousand Oaks: Compañía Californiana que se desarrolla en la industria terapéutica y en biotecnología. Amgen fue pionera en el desarrollo de nuevos productos basados en los avances en la recombinación del ADN y en biología molecular. Amgen utiliza Haskell para lograr una rápida creación de productos software que implemente modelos matemáticos complejos y que les permita realizar una validación matemática rigurosa.
- Antíope Fair Haven: Compañía de Nueva Jersey (USA) que se desempeña en la industria de la comunicación inalámbrica y redes. Realizan diseño de sistemas inalámbricos, desde la capa física y las demás capas de protocolo de red. Utilizan Haskell para realizar programas de simulación y modelar los diseños de hardware.
- Ericsson: Ericsson utiliza Haskell para la implementación de algoritmos de procesamiento digital de señales.

1.7 FAMILIA DE LENGUAJES EN EL PARADIGMA FUNCIONAL

Todos los lenguajes funcionales están basados en el Lambda-Calculo. De hecho su influencia es evidente en lenguajes de programación funcionales como Lisp, Hope, Miranda, Haskell, ML y Scheme. El lambda cálculo provee de una sintáxis básica para definir primitivas de programación y usa el concepto de función matemática para transformar argumentos en resultados.

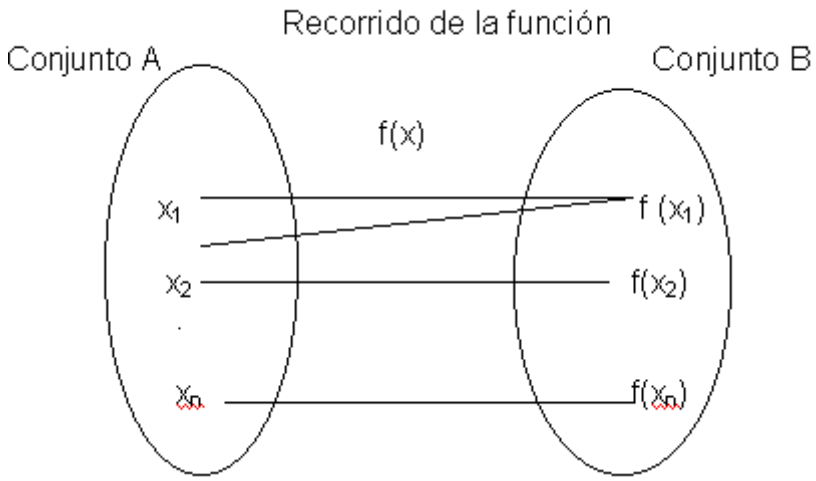
El lenguaje Lisp ideado por John McCarthy fue el primer lenguaje de programación funcional y fue el único por muchos años. Aunque todavía se usa Lisp, no es un lenguaje que reúna las exigencias. Debido a la creciente complejidad de los programas de ordenador, se hizo necesaria una mayor verificación del programa por parte del ordenador. En los años ochenta se crearon un gran número de lenguajes funcionales tipados. Algunos ejemplos son ML, Scheme (una adaptación de Lisp), Hope y Miranda.

Los lenguajes ML y Scheme tienen también muchos seguidores. Estos lenguajes han hecho algunas concesiones en la dirección de los lenguajes imperativos. Un grupo de investigadores notables concibió un lenguaje que incluía todas las mejores cualidades de los diferentes lenguajes. Este lenguaje se llama Haskell y es puramente funcional de propósito general.

1.8 CONCEPTOS GENERALES

1.8.1 REPASO DE FUNCIÓN MATEMÁTICA

Una función es una regla de asociación que relaciona dos o más conjuntos entre si; generalmente cuando tenemos la asociación de dos conjuntos las función se define como una regla de asociación entre un conjunto llamado dominio con uno llamado codominio o imagen.



Expresado de otra manera, una función f entre dos conjuntos A y B, es una correspondencia que a cada elemento de un subconjunto de A, llamado "Dominio de f", le hace corresponder uno y sólo uno de un subconjunto B llamado "Imagen de f".

f : A → B

f es una función de A en B, o f es una función que toma elementos del dominio A y los aplica sobre otro llamado imagen B.

Se dice que el ***dominio de una función*** son todos los valores que puede tomar el conjunto del dominio y que encuentra correspondencia en el conjunto llamado imagen, generalmente cuando se habla del plano, el dominio es el intervalo de valores que están sobre el eje de las X´s y que nos generan una asociación en el eje de las Y´s.

El otro conjunto que interviene en la definición es el conjunto llamado ***codominio o imagen de la función,*** este conjunto es la gama de valores que puede tomar la función; en el caso del plano son todos los valores que puede tomar la función o valores en el eje de las Y´s.

También, cuando se grafica en el plano cartesiano se tiene una relación de dos variables, considerando como variable aquella literal que esta sujeta a los valores que puede tomar la otra.

Usualmente las funciones se expresan con la siguiente sintaxis

f(x) = 2 * x

donde:

x: argumento nominal que representa a los elementos del Dominio

f: nombre de la función

En el lado derecho se representa el elemento del codominio o imagen o la forma de obtenerlo.

Generalmente, para calcular la imagen de x a partir de f, deben usarse otras funciones, las cuales también deberán definirse, por ejemplo.

f(x) = 2 * x

g(x) = f(x) + 4

donde f, g, *, + son funciones

En general:

1. Las funciones establecen relaciones entre dos conjuntos, imponiendo la restricción única de que un elemento no puede tener dos imágenes,
2. La definición de funciones es una estructura jerárquica, en la cual las funciones más simples aparecen en la definición de las funciones más complejas.
3. En las definiciones, las funciones utilizadas a la derecha del signo de igualdad se referencian solo por su nombre. El significado puede obtenerse de reemplazar el nombre por su definición.
4. Esta sucesión se clausura con funciones primitivas, cuyo significado se da por sobreentendido.
5. Básicamente se aplican los conceptos del diseño modular, en el cual un problema complejo se descompone en problemas más simples.
6. Cada uno de ellos es un problema primitivo, o bien, deberá volver a descomponerse.
7. La facilidad de referenciar una función por su nombre, permite que la misma sea empleada sin tener la necesidad de conocer su estructura interna.
8. Esta estrategia permite diseñar realizando refinamientos sucesivos.
9. La estructura de un módulo complejo puede definirse, sin que necesariamente estén definidos los módulos componentes.

1.8.2 FUNCIONES: SINTAXIS EN EL PARADIGMA FUNCIONAL

En el esquema funcional, las funciones se denominan expresiones y se pueden representar con la siguiente sintaxis:

<expresión> ::=

<variable> |

<constante> |

(<variable>, ... ,<variable>) <expresión> |

<expresión> (<expresión>, ... ,<expresión>)

Todas las expresiones denotan un valor, pueden ser expresiones que no admitan ser aplicadas a ningún argumento, ya que están totalmente evaluadas o expresiones que pueden ser aplicadas a un cierto número de argumentos para ser totalmente evaluadas.

Las formas permitidas por la gramática son:

A. expresiones atómicas, que pueden ser <variable> y <constante>
B. expresiones compuestas
   - abstracciones funcionales: (<variable>, . . .,<variable>) <expresión>
   - aplicaciones funcionales: <expresión> (<expresión>, …. ,<expresión>)

1.8.3 ABSTRACCIÓN FUNCIONAL

Una abstracción funcional es una expresión por la cual se define una función. Considérese la siguiente expresión:

3*5

mediante una abstracción funcional puede construirse una expresión de tal forma que el producto anterior sea un caso particular de ésta.

Por ejemplo:

(x) 3 * x

es una forma más general, y mediante una aplicación de la misma puede obtenerse la expresión original. Tambien se podría generalizar más la abstracción funcional, expresando:

(x, y) x * y

donde la definición de la función o abstracción funcional cumple con la construcción general:

(<variable>, ... ,<variable>) <expresión>

También se puede expresar la abstracción funcional de la siguiente forma:

(x) (y) <expresión>

tanto (y) <expresión> como (x) (y) <expresión> son abstracciones funcionales. Esta opción indicaría que la imagen será una función de y.

1.8.4 FUNCIÓN DE ORDEN SUPERIOR

Es una función tal que alguno de sus argumentos es una función o que devuelve una función como resultado.

- Son útiles porque permiten capturar esquemas de cómputo generales (abstracción).
- Son más útiles que las funciones normales (parte del comportamiento se especifica al usarlas).

Por ejemplo:

Definimos una función que invoque a otra función dos veces, por ejemplo el incremento en 1. Para expresar está función colocamos en el primer argumento, en lugar de una variable, una función llamada f, quedando la siguiente expresión:

(f ,x) f (f x) ó (f) (x) f (f (x))

donde:

- f representa la expresión o función.
- x es una variable.
- (x) f (f (x)) es una abstracción funcional (que posee la variable 'x' y la expresión ' f').
- f (f (x)) es una abstracción funcional (define la expresión 'f').
- f (x) es una aplicación funcional (define el cuerpo de la abstracción funcional anterior).

esta expresión indicaría que, la imagen de la función sobre la variable f es una función sobre la variable x que aplica dos veces f a x

Si asociamos un nombre a esta función:

dos_veces = (f) (x) f (f x)

donde:

- (x) f(f(x)) es la expresión
- y (f) (x) f(f(x)) la abstracción funcional

Si consideramos la función inc (que incrementa en 1 un valor), por ejemplo:

inc = (x) x+1

Si asociamos f con la función inc, la imagen de inc de acuerdo a dos_veces sería:

dos_veces (inc) ===> (x) inc(inc(x))

la aplicación de dos_veces a una función da como resultado otra función, la cual obviamente puede aplicarse a su vez a otro valor.

Si probamos con el valor de x = 3:

(dos_veces (inc)) 3 ===> (3) inc(inc(3))

4

5

===> 5

En este ejemplo podemos destacar que los argumentos y resultados de la aplicación de funciones pueden ser a su vez funciones, las cuales pueden ser también aplicadas a otros argumentos.

Este tipo de función es denominado FUNCIÓN DE ORDEN SUPERIOR denotando el hecho que sus argumentos y resultados pueden ser funciones.

1.8.5 LAMBDA-CALCULO

Cálculo lambda: es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión. Se puede considerar al cálculo lambda como el más pequeño lenguaje universal de programación. Consiste de una regla de transformación simple (substitución de variables) y un esquema simple para definir funciones.

El cálculo lambda es universal porque cualquier función computable puede ser expresada y evaluada a través de él. Church redujo todas las nociones del cálculo de sustitución.

Normalmente, un matemático debe definir una función mediante una ecuación.

Por ejemplo, si una función f es definida por la ecuación f(x)=t, donde t es algún término que contiene a x, entonces la aplicación f(u) devuelve el valor t[u/x], donde t[u/x] es el término que resulta de sustituir u en cada aparición de x en t.

Por ejemplo, si f(x)=x*x, entonces f(3)=3*3=9.

- Lambda Expresiones

Church propuso una forma especial (más compacta) de escribir estas funciones. En vez de decir “la función f donde f(x)=t”, él simplemente escribió \x.t.

Para el ejemplo anterior: \x.x*x.

Un término de la forma \x.t se llama “lambda expresión”. La principal característica de lambda cálculo es su simplicidad ya que permite efectuar solo dos operaciones:

• Definir funciones de un solo argumento y con un cuerpo especifico, denotado por la siguiente terminología: x.B, en donde x determina el parámetro o argumento formal y B representa el cuerpo de la función, es decir f(x) = B.

Ejemplo 1:

Para la función

f: A → B

f(x) 2x + 1

Podemos escribirla como una -expresión de la forma

x . 2x+1

Ejemplo 2:

f(x, y) = (x + y)*2

Se expresaría en -Calculo como

x. y. * (+ x y) 2

Aplicar alguna de las funciones definidas sobre un argumento real (A); lo que es conocido también con el nombre de reducción, y que no es otra cosa que sustituir las ocurrencias del argumento formal (x), que aparezcan en el cuerpo (B) de la función, con el argumento real(A), es decir: (x.B) A

Ejemplo 3:

(x. (x+5)) 3

Lo que indica que en la expresión x+5, se debe sustituir el valor de x por 3.

Los dos mecanismos básicos presentados anteriormente se corresponden con los conceptos de abstracción funcional y aplicación de función; si le agregamos un conjunto de identificadores para representar variables se obtiene lo mínimo necesario para tener un lenguaje de programación funcional.

El cálculo Lambda tiene el mismo poder computacional que cualquier lenguaje imperativo tradicional.

1.8.6 EVALUACIÓN PEREZOSA(LAZY)

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

Haskell, Miranda y Clean son perezosos, mientras que LISP, SML, Erlang y Scheme son estrictos.

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. EL LENGUAJE HASKELL

2.1 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 (lazy evaluation), tipado polimórficos, tipos definidos por el usuario, entre otras.

Su nombre proviene de Haskell Brooks Curry, quien trabajó en la lógica matemática que sirve como fundamento de los lenguajes funcionales. Haskell se basa en el cálculo lambda, razón por la cual el signo de lambda (λ) es utilizado como ícono.

Está específicamente diseñado para manejar un ancho rango de aplicaciones, tanto numéricas como simbólicas. Para este fin, Haskell tiene una sintaxis expresiva y una gran variedad de constructores de tipos, a parte de los tipos convencionales (enteros, punto flotante y booleanos).

Hay disponible un gran número de implementaciones. Todas son gratis. Los primeros usuarios, tal vez, deban empezar con Hugs, un intérprete pequeño y portable de Haskell.

2.1.1 ¿POR QUE USAR HASKELL?

Escribir en grandes sistemas software para trabajar es difícil y caro. El mantenimiento de estos sistemas es aún más caro y difícil. Los lenguajes funcionales, como Haskell pueden hacer esto de manera más barata y más fácil.

Haskell, un lenguaje puramente funcional ofrece:

1. Un incremento substancial de la productividad de los programas.
2. Código más claro y más corto
3. Una “semántica de huecos” más pequeña entre el programador y el lenguaje.
4. Tiempos de computación más cortos.

Haskell es un lenguaje de amplio espectro, apropiado para una gran variedad de aplicaciones. Es particularmente apropiado para programas que necesitan ser altamente modificados y mantenidos.

La vida de muchos productos software se basa en la especificación, el diseño y el mantenimiento y no en la programación.

2.1.2 HASKELL EN ACCIÓN

Haskell es un lenguaje de programación fuertemente tipado (en términos anglófilos, a typeful programming language; término atribuido a Luca Cardelli). Para aquellos lectores que estén familiarizados con Java, C, Modula, o incluso ML, el acomodo a este lenguaje será más significativo, puesto que el sistema de tipos de Haskell es diferente y algo más rico.

2.2 TIPOS DE DATOS

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.

2.2.1 SISTEMA DE INFERENCIA DE TIPOS

Haskell cuenta con un sistema de inferencia de tipos que consiste en:

- El programador no está obligado a declarar explícitamente el tipo de las expresiones, sin embargo es aconsejable y una buena practica escribirlas.
- Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincida 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, y de esta manera se evita tener que 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 del polimorfismo.

Otro ejemplo:

Dada la expresión:

```
doble x = 2 * x
```

donde 2 * x implica que el operador (*) se debe poder aplicar sobre 2 y sobre x. Luego, si x fuera de tipo String, el compilador nos informaría un error de tipos.

Haskell nos infomará sobre un conflicto de tipos en las siguientes situaciones:

1) Si se define la función: doble x = “nombre” * x . Entonces el valor “nombre” no es del tipo adecuado para el operador (*).

2) Si se define la función:

doble :: String -> Int

doble x = 2 * x

Entonces tendremos un error al intentar la operación (*) con un argumento de tipo String.

3) Al evaluar la expresión Main> doble “2”, el valor “2” (String) no coincide con el tipo Int.

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

2.2.2 TIPOS PREDEFINIDOS

Existen varios “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.

Describiremos a continuación los tipos básicos:

Booleanos

Se representan por el tipo "Bool" y contienen dos valores: "True" y "False". Las funciones para manipular valores boleanos son: (&&), (||) 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)

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 está restringido. También se puede utilizar el tipo Integer para denotar enteros sin límites superior ni inferior.

A continuación proveemos algunos operadores y funciones que permiten manipular 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.

Algunos operadores y funciones de manipulación de flotantes:

(+), (-), (*), (/)	Suma, resta, multiplicación y división fraccionaria.
(^)	Exponenciación con exponente entero.
(**)	Exponenciación con exponente fraccionario.
sin, cos, tan	Seno, coseno y tangente.
asin, acos, atan	Arcoseno, arcocoseno y arcotangente.
truncate	Convierte un número fraccionario en un entero redondeando hacia el cero (trunca la parte decimal).
round	Convierte un número fraccionario en un entero redondeando hacia el entero más cercano.
fromIntegral	Convierte un entero en un número en punto flotante.
log	Logaritmo en base e.
sqrt	Raíz cuadrada (positiva).

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
'\t' or '\HT'   tabulador
```

2.3 FUNCIONES

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.

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`.

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)
```

`(+)` 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* (en honor a Haskell B.Curry) 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`.

Nota : Se podría escribir simplemente *`(+)::Int->Int->Int`*, puesto que el operador `->` es asociativo a la derecha.

En Haskell las funciones se definen usualmente a través de una colección de ecuaciones. Por ejemplo, la función inc puede definirse por una única ecuación:

```
inc n = n+1
```

Una ecuación es un ejemplo de declaración. Otra forma de declaración es la declaración de tipo de una función o type signature declaration, con la cual podemos dar de forma explícita el tipo de una función; por ejemplo, el tipo de la función inc:

```
inc :: Integer -> Integer
```

A continuación proveemos de otro ejemplo con dos maneras alternativas de definir la función suma:

```
suma::Integer -> Integer -> Integer
suma x y  =  x + y
```

`Entonces, l`a invocación a esta función se hará de la siguiente forma:

```
suma 2 5
7
```

Por otra parte, la función suma también podría haberse definido como una función no parcializada (uncurried):

```
suma::(Integer, Integer) -> Integer

suma (x, y)  =  x + y
```

Luego, la invocación a esta función se debe hacer de la siguiente manera:

```
suma (2,5)
7
```

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:

Los operadores se utilizan usando por defecto la notación infija, lo que facilita la lectura matemática:

```
4 + 3

x + y
```

Sin embargo, también es posible usar el operador en notación prefija como se muestra en el siguiente ejemplo:

```
(+) 2 3
```

2.4 EL ENTORNO DE HASKELL - HUGS

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`":

```
Hugs>:load fichero.hs ó

Hugs>:l fichero.hs
```

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

```
Main> cuadrado 2
4 :: Integer

Main> cuadrado(3+5)
64 :: Integer

Main> menor(2,4)
2 :: Integer

Main> cuadrado (menor (3,4))
9 :: Integer
```

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.

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.

En resumen, para escribir un programa en Haskell hay que cumplir los siguientes pasos:

1. Escribir un script con la definición de todas las funciones que necesitemos para resolver el problema. Se puede utilizar cualquier editor de texto, aunque es recomendable usar alguno que resalte automáticamente la sintaxis de Haskell. Guardamos el archivo con una extensión .hs.

2. Cargar el script en Hugs. Para ello utilizamos el comando: load seguido del nombre del archivo. Ahora en el prompt de Hugs ya podemos evaluar cualquier expresión que haga referencia a funciones definidas en el script.

3. Ingresar en el prompt la expresión a evaluar y presionar la tecla <RETURN>, para que el sistema evalúe la expresión e imprima su valor antes de volver a mostrar el *prompt*.

2.4.1 ELEMENTOS DE HUGS

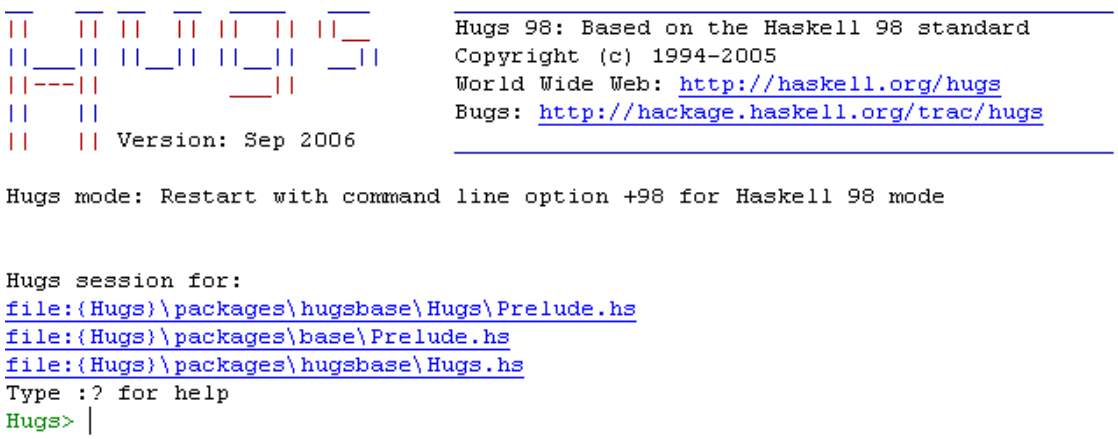
Prelude.hs

El prelude es un fichero de nombre Prelude.hs que es cargado automáticamente al arrancar Hugs y contiene la definición de un conjunto de funciones que podemos usar cuando las necesitemos.

Algunos ejemplos: div, mod, sqrt, id, fst, snd.

Ventana de evaluación

Al iniciar una sesión en Hugs, nos aparecerá una ventana como ésta:



Hugs evaluará cualquier expresión, sintácticamente correcta, que se ingrese en el prompt

```
Hugs>  <expresion>

<resultado>
```

Por ejemplo, si ingresemos (7 + 4) ∗ 3, la evaluará y nos devolverá 33

```
Hugs> (7+4)*3
```

```
33 :: Integer
```

Si ingresamos sqrt 2, nos devolverá una aproximación de√2

```
Hugs> sqrt 2

1.4142135623731 :: Double
```

2.4.2 COMANDOS DE HUGS

Si uno escribe :? y presiona ENTER:

```
Hugs> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names exported by <modules>
:main <aruments>    run the main function with the given arguments
:find <name>        edit module containing definition of name
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Hugs>
```

La primera línea dice que se puede usar como abreviatura sólo la primera letra de cada comando.

Cargar un archivo

Para cargar un archivo hay que utilizar el comando :load <nombre>.

```
Hugs> :l nuevo.hs
```

Editar un archivo

Para editar un archivo, hay que utilizar el comando : edit.

Ejemplo 1:

```
--edita el último archivo cargado
Hugs> :edit
```

Ejemplo 2:

```
--edita el archivo especificado
Hugs> :edit archivo.hs
```

Crear un archivo

Para crear un nuevo archivo hay que usar el comando :edit <nombre>, el sistema intentará abrirlo, al no encontrarlo informará que no existe y preguntará si quiere crear uno nuevo, si se acepta la edición y se comienza a escribir el nuevo código.

```
Hugs> :e nuevo.hs
```

2.4.3 MÓDULOS

Haskell proporciona un conjunto de definiciones globales que pueden ser usadas por el programador sin necesidad de definirlas. Estas definiciones aparecen agrupadas en módulos de biblioteca. Una biblioteca es un conjunto de definiciones relacionadas.

La importación de los módulos se consigue escribiendo la palabra import al inicio del programa.

En Hugs existen librerías llenas de módulos muy útiles, por ejemplo, el módulo de biblioteca Prelude, visto anteriormente, es automáticamente importado por cualquier programa.

Cada programa o script que se crea va a ser un módulo, y un conjunto de módulos formarán un proyecto.

Una vez que se comienza a escribir un script es recomendable definir el nombre de ese módulo. Supongamos que vas a crear un programa que quieras llamar "Test1", debes hacer lo siguiente (en tu editor de texto):

Definir el módulo de la siguiente forma:

```
module Test1 where
```

Donde la primera letra del nombre del módulo debe ir en mayúsculas.

Escribir el programa, grabar y cargarlo en Hugs y ahora en vez de aparecer

```
Hugs>
```

Aparecerá esto:

```
Test1>
```

La ventaja de los módulos es que pueden importarse, y cargar muchos programas a la vez, usando el comando import.

Ejemplo de Hola Mundo:

```
--MiPrimerPrograma.hs
module Main where
```

```
main = putStrLn "¡Hola mundo!"
```

Si lo evaluamos:

```
Main> main
¡Hola mundo!
 :: IO ()
```

2.5 SINTAXIS DE HASKELL

En esta sección se muestra lo esencial de la sintaxis de Haskell.

2.5.1 CASE SENSITIVE

La sintaxis de Haskell diferencia entre mayúsculas y minúsculas. Se dice que es Case sensitive.

2.5.2 COMENTARIOS

Una buena práctica de programación es documentar siempre el código fuente. Un comentario en un script es información de valor para el lector humano, más que para el computador (no interviene para nada al momento de evaluar una expresión). Hay dos formas de expresar los comentarios:

En una línea: utiliza el símbolo "--" que inicia un comentario y ocupa la parte de la línea hacia la derecha del símbolo. Por ejemplo:

```
-- Este es un comentario, cuando empieza con "--"
```

MultiLinea: utiliza "{-" para inicia el comentario y -} para finalizarlo. Por ejemplo:

```
{- Este es un comentario,

en dos líneas-}
```

2.5.3 VALORES

Los valores son entidades abstractas que podemos considerar como la respuesta a un cálculo

```
5     -1     8
```

Cada valor tiene asociado un tipo (<valor> :: <tipo>)

```
2 :: Int

Int   ≡     {...-3,-2,-1,0,1,2,3...}
```

2.5.4 EXPRESIONES

Las expresiones son términos construidos a partir de valores

```
(2*3)+(4-5)
```

La reducción o evaluación de una expresión consiste en aplicar una serie de reglas que transforman la expresión en un valor

```
(2*3)+(4-5) -→ 6+(4-5) -→ 6+(-1) -→ 6-1 -→ 5
```

Un valor es una expresión irreducible

Toda expresión tiene un tipo: el tipo del valor que resulta de reducir la expresión (Sistema Fuertemente Tipado).

```
(2*3)+(4-5) :: Int
```

Las funciones también son valores (y por lo tanto tendrán tipo)

2.5.5 IDENTIFICADORES

Un identificador Haskell consta de una letra seguida por cero o más letras, dígitos, subrayados y comillas simples. Los identificadores son *case-sensitive* (el uso de minúsculas o mayúsculas importa).

Los siguientes son ejemplos posibles de identificadores:

sum f f'' fintSum nombre_con_guiones

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 class instance primitive

La letra inicial del identificador distingue familias de identificadores: empiezan por

- *Mayúscula* los tipos y constructores de datos
- *Minúscula* los nombres de función y variables

2.5.6 OPERADORES

En Haskell los operadores son funciones que se escriben entre sus (dos) argumentos en lugar de precederlos. A una función escrita usando la notación infija se le llama un operador. Por ejemplo,

3 <= 4 en lugar de menorIgual 3 4

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

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

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

- **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).

- **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.

2.5.7 EXPRESIONES IF-THEN-ELSE

En Haskell, no hay estructuras de control, pero si existe la función especial if – then – else, que implica otro modo de escribir expresiones cuyo resultado depende de cierta condición, la sintaxis es:

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

Si la expresión booleana `exprBool` es True devuelve `exprSi`, si es False, devuelve `exprNo`.

Obsérvese que una expresión de ese tipo sólo es aceptable si `exprBool` es de tipo Bool y `exprSi` y `exprNo` son del mismo tipo.

Por ejemplo:

```
Hugs> if 5 > 2 then 10.0 else 20.0
10.0 :: Double
Hugs> 2 * if 'a'<'z' then 10 else 4
20 :: Integer
```

Ejemplos, definiendo funciones:

El mayor de dos números:

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

La evaluación sería:

```
Main> maxEnt 3 7
7 :: Integer
```

El mayor de tres números:

```
mayor:: Int->Int->Int->Int
mayor x y z =     if x >= y && x>=z then x
                        else if y >= x && y>=z  then y
                              else z
```

La evaluación sería:

```
Main> mayor 2 3 4
4 :: Int
```

2.5.8 EXPRESIONES CASE

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

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

Al evaluar la función:

```
Hugs> paridad 3
"impar" :: [Char]
```

2.5.9 EXPRESIONES 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. En general una ecuación con guardas toma la forma:

```
f x1 x2 ... xn | condicion1 = e1
```

```
                | condicion2 = e2
                  .
                  .
                | condicionm = em
                | otherwise = exp
```

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, "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.

Ejemplos

1.
```
minimo x y  | x <= y = x
            | otherwise = y
```

Al evaluar la función:

```
Main> minimo 3 6
3 :: Integer
```

ó

```
Main> minimo 7 5
5 :: Integer
```

2.
```
absoluto:: Integer -> Integer

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

Al evaluar la función:

```
Main> absoluto 3
3 :: Integer
ó
Main> absoluto (-4)
4 :: Integer
```

3.
```
signo :: Integer -> Integer

signo x   | x > 0 = 1
          | x == 0 = 0
          | otherwise = -1
```

Al evaluar la función:

```
Main> signo 3
1 :: Integer

Main> signo 0
0 :: Integer

Main> signo (-10)
-1 :: Integer
```

2.5.10 DEFINICIONES LOCALES

Las definiciones de funciones pueden incluir definiciones locales para variables que pueden ser usadas en guardas o en la parte derecha de una ecuación, en Haskell hay dos formas:

EXPRESIONES LET.

Las *expresiones let* de Haskell son útiles cuando se requiere un conjunto de declaraciones locales. Se introducen en un punto arbitrario de una expresión utilizando una expresión de la forma:

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

Por ejemplo:

```
Hugs> let x = 1 + 4 in x*x + 3*x + 1
41 :: Integer
```

ó:

```
Hugs> let p x = x*x + 3*x + 1 in p (1 + 4)
41 :: Integer
```

EXPRESIONES WHERE

A veces es conveniente establecer declaraciones locales en una ecuación con varias con guardas, lo que podemos obtener a través de una *cláusula where*:

```
f x y  |  y>z          =  ...
       |  y==z         =  ...
       |  y<z          =  ...
     where z = x*x
```

Nótese que ésto no puede obtenerse con una expresión `let`, la cual únicamente abarca a la expresión que contiene. Solamente se permite una cláusula `where` en el nivel externo de un grupo de ecuaciones o expresión case. Por otro lado, las cláusulas `where` tienen las mismas propiedades y restricciones sobre las ligaduras que las expresiones `let`.

Estas dos formas de declaraciones locales son muy similares, pero recordemos que una expresión `let` es una *expresión*, mientras que una cláusula `where` no ---forma parte de la sintaxis de la declaración de funciones y de expresiones case.

Por ejemplo:

```
comparaCuadrado::Integer -> Integer -> String
comparaCuadrado x y   | y>z   = " el segundo argumento es mayor
que el cuadrado del primero"
                      | y==z =   "el segundo argumento es igual
que el cuadrado del primero"
                      | y<z   =   "el segundo argumento es menor
que el cuadrado del primero"
                       where z = x*x
```

Si evaluamos:

```
Main> comparaCuadrado 5 3
"el segundo argumento es menor que el cuadrado del primero" ::
[Char]
Main> comparaCuadrado 5 45
" el segundo argumento es mayor que el cuadrado del primero" ::
[Char]
Main> comparaCuadrado 5 25
"el segundo argumento es igual que el cuadrado del primero" ::
[Char]
```

2.5.11 Disposición del código: Espaciado (layout)

En Haskell no existe el concepto de separadores que marquen el final de una ecuación, una declaración, etc. Para determinar esto, Haskell, utiliza una sintaxis bidimensional denominada ***espaciado*** *(layout)* que se basa esencialmente en que las declaraciones están alineadas por columnas.

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 uso del layout reduce en gran medida el desorden sintáctico asociado con los grupos de declaraciones, mejorando la legibilidad. Es fácil de aprender y recomendamos su uso.

Por ejemplo, dada la siguiente expresión:

```
--usando layout
ejemplo1::Integer
ejemplo1  = (a + a)
            where a = 6

ejemplo2::Integer
ejemplo2  = (a + b)
            where
                 a = 6
                 b = 5
```

2.5.12 EXPRESIONES RECURSIVAS

Una expresión es recursiva cuando su evaluación (en ciertos argumentos) involucra el llamado a la misma expresión que se esta definiendo. La recursión es una herramienta poderosa y usada muy frecuentemente en los programas en Haskell.

La recursión es una técnica que se basa en definir expresiones en términos de ellas mismas. Si del lado derecho de una expresión aparece en algún punto la misma expresión que figura del lado izquierdo, se dice que la expresión tiene llamado recursivo, es decir **"se llama a sí misma"**.

En la definición recursiva, es necesario considerar dos momentos de evaluación:

- **Evaluación del caso Básico:** Momento en que se detiene el proceso de evaluación, se obtiene una valorización de la expresión.
- **Evaluación Recursiva:** Se evalúa la expresión actual, efectuando evaluaciones a si misma, hasta obtener la valorización de la expresión.

Por ejemplo, recordemos la definición (matemática) de la función factorial:

factorial : N → N

factorial(0) = 1

factorial(n) = n × factorial(n − 1)

si n > 0

Podemos definir esta función en Haskell con una correspondencia directa:

1. Ejemplo usando una sola expresión, en donde la evaluación del caso básico y la evaluación recursiva están en una expresión if-then-else :

```
factorial:: Integer -> Integer
factorial n = if n==0  then 1
                  else n * fact(n-1)
```

2. Ejemplo usando dos expresiones, una para evaluar el caso básico y la otra para hacer la evaluación recursiva:

```
factorial::Integer -> Integer
factorial 0 = 1
```

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

2.5.13 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>
```

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 (+):

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

2.6 TIPOS DE DATOS COMPUESTOS

Los tipos compuestos, son aquellos cuyos valores se construyen utilizando otros tipos.

2.6.1 TUPLAS

Una tupla es un dato compuesto, es una sucesión de elementos, donde el tipo de cada elemento puede ser distinto. El tamaño de una tupla es fijo.

Definición general: 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 como (x1,x2,...,xn) donde cada x1,x2,...,xn tiene tipos T1,T2,...,Tn.

Aridad:

- La aridad de una tupla es el número de elementos.
- La tupla de aridad 0, (), es la tupla vacía, denominada tipo unidad.
- No están permitidas las tuplas de longitud 1.

Ejemplos:

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

Obsérvese que los elementos de una tupla pueden tener tipos diferentes y el tamaño de una tupla es fijo.

Las tuplas son útiles cuando una función tiene que devolver más de un valor, por ejemplo:

```
--devuelve el anterior y posterior de un número
antPos:: Integer -> (Integer,Integer)
antPos x = (x - 1, x + 1)

Main> antPos 2
(1,3) :: (Integer,Integer)
```

2.6.2 LISTAS

Una lista es una colección de cero o más elementos todos del mismo tipo. Las expresiones de tipo lista se construyen con [] (que representa la lista vacía) y : (a:as es la lista que empieza con el elemento a y sigue con la lista as). También pueden escribirse entre corchetes, con los elementos separados por comas.

Sintaxis para listas

x1: (x2: (...(xn−1: (xn: [])))) ó [x1,x2,...xn−1,xn]

Ejemplo:

```
1:(2:(3:([])))          ó      [1,2,3]
```

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

Definición general: Si v1,v2,...,vn son valores con tipo t, entonces v1: (v2: (...(vn−1: (vn: [])))) es una lista con tipo [t].

Por ejemplo: [Int] representa el tipo lista de enteros, [Bool] es una lista de booleanos, etc.

Definición recursiva: Una lista está compuesta por una cabeza y una cola que es una lista compuesta por los elementos restantes.

La lista que no tiene elementos no puede dividirse en cabeza y cola. Se denota [ ] y se dice lista vacía.

En una lista el operador (:) permite dividir cabeza y cola: (x:xs)

Longitudes:

- La longitud de una lista es el número de elementos.
- La lista de longitud 0, [], es la lista vacía.
- Las listas de longitud 1 se llaman listas unitarias.

Ejemplos

```
[False,True] :: [Bool]          --contiene elementos booleanos
[1,2,3]:: [Int]                 --contiene elementos enteros
['a', 'a', 'a'] :: [Char]       --contiene elementos char,
                                --se visualiza como un String "aaa"
[(True,1), (False,2)]:: [(Bool,Int)]     --contiene
tuplas(Bool,Int)
[[1,2,3], [4], [5,6]]:: [[Int]]     --contiene listas de enteros
["uno","dos"] :: [[Char]]       --contiene elementos listas de char
                                -- o String
```

El tipo String es sinónimo de [Char], y las listas de este tipo se pueden escribir entre comillas: "uno" es lo mismo que [’u’, ’n’, ’o’].

Algunas funciones útiles sobre listas

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.
take n xs	devuelve los n primeros elementos de xs
drop n xs	devuelve el resultado de sacarle a xs los primeros n elementos
head xs	devuelve el primer elemento de la lista
tail xs	devuelve toda la lista menos el primer elemento
last xs	devuelve el último elemento de la lista
init xs	devuelve toda la lista menos el último elemento
xs ++ys	concatena ambas listas
xs !! n	devuelve el n-ésimo elemento de xs
elem x xs	dice si x es un elemento de xs

Ejemplos:

```
Hugs>length [1,3,10]
3
Hugs>[1,3,10] ++ [2,6,5,7]
```

```
[1, 3, 10, 2, 6, 5, 7]
Hugs>concat [[1], [2,3], [], [4,5,6]]
[1, 2, 3, 4, 5, 6]
Hugs>map fromEnum ['H', 'o', 'l', 'a']
[104, 111, 108, 97]
```

Ejemplos usando funciones propias:

1. Retorna una lista de enteros:

```
lista:: [Int]
lista = [1,2,3]

Main> lista
[1,2,3] :: [Int]
```

2. Dada una lista de enteros retornar la suma de todos los elementos:

```
sumar:: [Integer] -> Integer
sumar []= 0
sumar (x : xs) = x + sumar xs


Main> sumar [1,2,3]
6 :: Integer
```

3. Dada una lista de enteros retornar la cantidad de elementos:

```
contar:: [Integer] -> Integer
contar []= 0
contar (x : xs) = 1 + contar xs

Main> contar [1,2,3]
3 :: Integer
```

4. Dada una lista de enteros mostrar sus elementos:

```
mostrar:: [Int] -> String
mostrar [] = []
mostrar (x : xs) = show(x) ++ " " ++ mostrar xs

Main> mostrar [1,1,1]
"111" :: [Char]
```

2.7 TIPOS DEFINIDOS POR EL USUARIO

2.7.1 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.`

Ejemplos:

```
type Nombre = String
type Edad = Integer
type String = [Char]
type Persona = (Nombre, Edad)
tocayos::Persona -> Persona -> Bool
tocayos (nombre,_) (nombre',_) = n == nombre'
```

El tipo String es un sinónimo de tipo, posee la siguiente defición:

type String = [Char]

Sintaxis especial para las cadenas de caracteres:

[’h’, ’o’, ’l’, ’a’] ≡ "hola"

Ejemplos:

```
"hola "++ [’m’, ’u’, ’n’, ’d’, ’o’] => "hola mundo"
```

2.7.2 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 pueden 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.

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.

2.8 TIPOS POLIMÓRFICOS

Haskell proporciona tipos *polimórficos* (tipos cuantificados universalmente sobre todos los tipos). Tales tipos describen esencialmente familias de tipos.

Por ejemplo, (para_todo a)[a] es la familia de las listas de tipo base a, para cualquier tipo a. Las listas de enteros (e.g. [1,2,3]), de caracteres (['a','b','c']), e incluso las listas de listas de interos, etc., son miembros de esta familia. (Nótese que [2,'b'] *no* es un ejemplo válido, puesto que no existe un tipo que contenga tanto a 2 como a 'b'.)

Ya que Haskell solo permite el cuantificador universal, no es necesario escribir el símbolo correspondiente a la cuantificación universal, y simplemente escribimos [a] como en el ejemplo anterior. En otras palabras, todas las variables de tipos son cuantificadas universalmente de forma implícita. Las listas constituyen una estructura de datos comunmente utilizada en lenguajes funcionales, y constituyen una buena herramienta para mostrar los principios del polimorfismo.

En Haskell, la lista [1,2,3] es realmente una abreviatura de la lista 1:(2:(3:[])), donde [] denota la lista vacía y : es el operador infijo que añade su primer argumento en la cabeza del segundo argumento (una lista). (: y [] son, respectivamente, *los operadores cons y nil del lenguaje Lisp*) Ya que : es asociativo a la derecha, también podemos escribir simplemente 1:2:3:[].

Ejemplo : “el problema de contar el número de elementos de una lista”

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

Esta definición es auto-explicativa. Podemos leer las ecuaciones como sigue: "La longitud de la lista vacía es 0, y la longitud de una lista cuyo primer elemento es x y su resto es xs viene dada por 1 más la longitud de xs." (Nótese el convenio en el nombrado: xs es el plural de x, y x:xs debe leerse: "una x seguida de varias x).

2.9 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
```

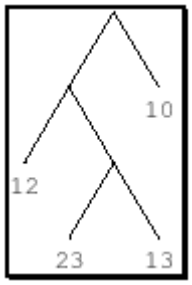
También pueden definirse tipos de datos polimórficos. El siguiente ejemplo define un tipo que representa árboles binarios:

```
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
Hugs> hojas a1
 [12, 23, 13, 10]
```

```
? hojas a2
Hugs> hojas a2
[[1,2,3], [4,5,6]]
10
12
23 13
```

2.10 SOBRECARGA Y CLASES EN HASKELL

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 esto 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
      x == y                  =  not (x /= y)
```

`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 laclase `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 anteriomente :

```
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.

3. ANEXO I – LA ENSEÑANZA DE HASKELL -

Haskell en la enseñanza de la Programación Declarativa (ProDec).

Desde el punto de vista educativo, un aspecto muy importante para la elección de un lenguaje es la existencia de intérpretes y compiladores eficientes y de libre disposición.

Casi simultáneamente, junto a la aparición del primer compilador eficiente y completo para Haskell desarrollado ´unicamente para plataformas UNIX, el primer sistema disponible para PCs surge a partir de 1992 con el desarrollo de Gofer50 por Mark Jones en las universidades de Oxford, Yale y Nottingham.

El éxito de un lenguaje es el disponer de un entorno de desarrollo adecuado. Muchos lenguajes no se

usan por carecer de tal entorno (p.e. Prolog).

Hugs 98 proporciona características adicionales que Haskell 98 no presenta en su definición original, como la posibilidad de utilizar una librería de funciones gráficas, o ampliaciones del sistema de tipos para describir datos vía registros extensibles.

Se puede afirmar existen motivos que aconsejan el uso de Haskell en favor de otro lenguaje funcional muy utilizado en las universidades, como LisP, o también su derivado, Scheme.

Quizás se podría argumentar que LisP, además de ser considerado el primer lenguaje funcional, es uno de los lenguajes más desarrollados, e incluso es aún un lenguaje *vivo* desde el punto de vista educacional y sigue utilizándose en muchas universidades pero esencialmente para aplicaciones dentro de la *Inteligencia Artificial*.

Sin embargo, ni LisP ni Scheme presentan la pureza ni algunas de las ventajas de Haskell. Entre éstas se podrían destacar la descripción de ecuaciones vía patrones a la izquierda, el polimorfismo restringido y el sistema de clases de tipos, que obligan a hacer un uso del lenguaje en forma disciplinada.

Haskell es el lenguaje con más posibilidades entre los actualmente existentes, es de amplia aceptación en muchas universidades, y está siendo utilizado en la industria cada vez con más profusión. Cabe resaltar que la Universidad de Málaga es pionera en su uso para la docencia y la investigación, desde la extensa divulgación realizada por la ACM

Los motivos citados justifican la elección del lenguaje Haskell, además justifican el uso de Haskell por sobre la elección de otros lenguajes funcionales. El lenguaje utilizado en el paradigma funcional queda reflejado en el título de muchos libros de programación funcional. Citemos por ejemplo [Bird, 1998]: *Introduction to Functional Programming using* Haskell , una revisión y adaptación a Haskell del clásico [Bird y Wadler, 1988]. O también, [Thompson, 1999], *Haskell: The Craft of Functional Programming*, al igual que [Ruiz Jim´enez et al., 2000], *Razonando con* Haskell. O el último e interesante libro de [Hudak, 2000], *The Haskell School of Expression. Learning Functional Programming through Multimedia*.

4. BIBLIOGRAFÍA

- Richard Bird. “Introduction to Functional Programming using Haskell”. Prentice Hall International, 2nd Ed. New York, 1997
- Luca Cardelli. “Basic Polymorphic Typechecking”. AT&T Bell Laboratories, Murray Hill, NJ 07974. 1998.
- Paul Hudak. “A Gentle Introduction to Haskell 98”. Yale University. Department of Computer Science. John Peterson. 1999.
- Paul Hudak. “The Haskell School of Expression: Learning Functional Programming through Multimedia”. Yale University. Cambridge University Press, New York, 2000.
- Simon Thompson. “Haskell: The Craft of Functional Programming”.3rd Edition. ISBN: 9780201882957. 1999.
- Jose E. Labra G. “Introducción al lenguaje Haskell”. Universidad de Oviedo, Departamento de Informática, Octubre 1998.
- Kurt Normark. “Funcional Programming in Scheme”. Department of Computer Science, Aalborg University, Denmark. 2003.
- Richard Kelsey, William Clinger and Jonathan Rees. "Revised Report on the Algorithmic Language Scheme". Higher-Order and Symbolic Computation, Vol. 11, No. 1, August 1998, pp. 7--105.
- Barendregt, Hendrik Pieter. “The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics”. 103 (Revised ed.), North Holland, Amsterdam. Corrections. 1984.
- Ken Slonneger y Barry L. Kurtz. “Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach”. Addison-Wesley, Reading, Massachusetts. 1995.
- Carlos Varela. “Programming Languages”. Rennselaer Polytechnic Institute. USA. 2011.
- Lucas Spigariol. “Paradigma Funcional” . Paradigmas de Programación - FRBA – UTN. 2007 .