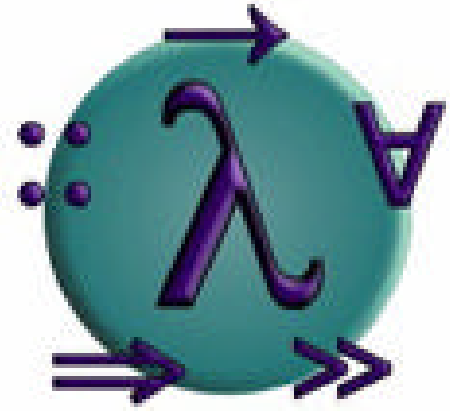
Marzo de 2008

Paradigmas de programación



Haskell un lenguaje de programación funcional

Elaborado por :

- ***Castillo, Julio Javier***
  Analista en Computación - Famaf
  Depto. Ingeniería en Sistemas de Información – UTN - FRC

INDICE

Un poco de 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, evaluación perezosa, inferencia estática de tipos, tipos de datos definidos por el usuario, encaje de patrones y 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 evolucion de los más importantes lenguajes de programación:

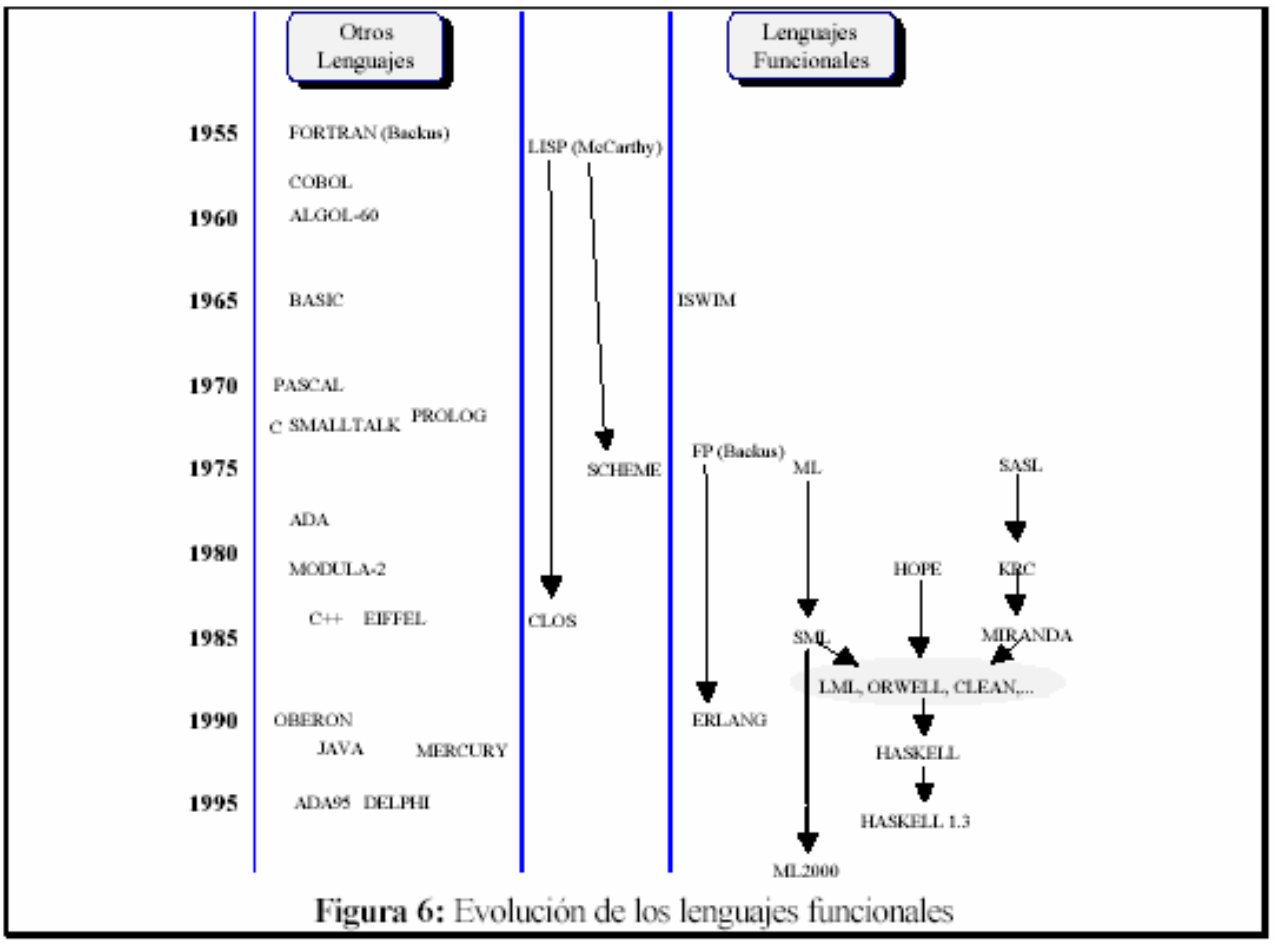


Figura 6: Evolución de los lenguajes funcionales

El lenguaje Haskell:

A. INTRODUCCIÓN

Haskell es un lenguaje de programación. En particular, es un lenguaje de tipos polimórficos, de evaluación perezosa, puramente funcional, muy diferente de la mayoría de los otros lenguajes de programación.
El nombre del lenguaje se debe a Haskell Brooks Curry.
Haskell se basa en el lambda cálculo, por eso se usa lambda como un logo.

B. ¿QUE ES HASKELL?

Haskell es un lenguaje de programación moderno, estándar, no estricto, puramente funcional.
Posee todas las características avanzadas, incluyendo polimorfismo de tipos, evaluación perezosa y funciones de alto orden. También es un tipo de sistema que soporta una forma sistemática de sobrecarga y un sistema modular.
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.

C. ¿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 substacial de la productividad de los programas.
2. Código más claro y más corto y con un mantenimiento mejor.
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.

Los lenguajes funcionales son idóneos para escribir especificaciones que actualmente son ejecutadas (y, por lo tanto, probadas y depuradas).Tal especificación es, por tanto, el primer prototipo del programa final.
Los programas funcionales son también relativamente fáciles de mantener porque el código es más corto, más claro y el control riguroso de los efectos laterales elimina gran cantidad de interacciones imprevistas.

D. DESARROLLO FUTURO DE HASKELL

Haskell 98 está completo. Es la definición oficial y actual de Haskell. Se espera que este lenguaje pueda seguir funcionando aunque se añadan nuevas extensiones y los compiladores seguirán apoyándose en Haskell 98.
No obstante, han sido propuestas muchas extensiones que han sido implementadas en algunos sistemas Haskell; por ejemplo el diseño de guardas, clases de tipos con multiparámetros, cuantificaciones locales y universales, etc.
La lista de correo de Haskell es un forum de discusión sobre las características de los nuevos lenguajes. La gente propone una nueva característica de algún lenguaje y trata de convencer a los investigadores sobre la conveniencia de desarrollarla en algún sistema Haskell. Al final, la gente que está interesada de verdad define esta nueva característica en Haskell 98 y lo presenta en un documento que es añadido a esta lista de correo.
Un buen ejemplo de la utilidad de este proceso es la creación de FFI (Foreing Function Interface) en Haskell.
Haskell II ha sido desarrollado durante mucho tiempo por un comité liderado por John Launchbury. Obviamente las extensiones que hayan sido bien descritas ycomprobadas tendrán más probabilidad de ser aceptadas.

E. HASKELL EN ACCION

Haskell es un lenguaje de programación fuertemente tipado (en términos anglófilos, a *typeful* programming language; término atribuido a Luca Cardelli.) Los tipos son penetrantes (pervasive), y presenta además un potente y complejo sistema de tipos. Para aquellos lectores que esten 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.

1-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.

1.1 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                    //no es necesario, pero es buena práctica
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.

1.2 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.

1.3 Funciones

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
```

2.-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`":

```
? :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.
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. 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)
```

`(+)` 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.

4. 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 `[]`.
- 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 lista 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.

5.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 ().

6. 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.

7.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
?
```

8. 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 aloperador (+):

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

9. 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, 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.
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 }
```

10. Tipos definidos por el usuario

10.1 *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.`

Ejemplo:

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

```
tocayos (nombre,_) (nombre',_) = n == nombre'
```

10.2 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.

10.3 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.

10.4 Tipos Polimórficos

Haskell proporciona tipos *polimóficos* (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).

pattern matching: (matcheo de patrones) En el ejemplo anterior, los miembros izquierdos de las ecuaciones contienen patrones tales como [] y x:xs. En una aplicación o llamada a la función, estos patrones son comparados con los argumentos de la llamada de forma intuitiva ([] solo "concuerda" (matches) o puede emparejarse con la lista vacia, y x:xs se podrá emparejar con una lista de al menos un elemento, instanciándose x a este primer elemento y xs al resto de la lista). Si la comparación tiene éxito, el miembro izquierdo es evaluado y devuelto como resultado de la aplicación. Si falla, se intenta la siguiente ecuación, y si todas fallan, el resultado es un error.

11. 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
```

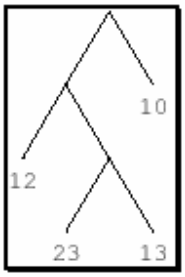
También pueden definirse tipos de datos polimórficos. El siguiente ejemplo define un tipo que representa árboles binarios:

```
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 ho jas 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]]
10
12
23 13
```

12.-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.

12.1 *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.

12.2 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"
```

en este ejemplo 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)
```

12.3 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. 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`.

13. 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)
```

I
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.

14. 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.

15. Ejercicios Prácticos

```
--Función recursiva para calcular el factorial de un número
  factorial :: Integer -> Integer
  factorial 0 = 1
  factorial n = n * factorial (n - 1)


--Función para calcular las raíces de una ecuación de segundo grado a partir de
sus coeficientes
  raíces :: Float -> Float -> Float -> (Float, Float)
  raíces a b c
   | disc >= 0 = ((-b + raizDisc) / denom,
```

```
                (-b - raizDisc) / denom)
   | otherwise = error "La ecuación tiene raíces complejas"
     where
        disc = b*b - 4*a*c
        raizDisc = sqrt disc
        denom = 2*a


--Función para calcular el valor de e (2.71828182845905)
   euler :: Double -> Double
   euler 0.0 = 1.0
   euler n  = 1.0 / product [1..n] + euler (n - 1.0)


--Algoritmo de ordenación quicksort
  qs::Ord a=>[a]->[a]
  qs [] = []
  qs (p:xs) = qs [x|x<-xs,x<=p] ++ [p] ++ qs [x|x<-xs,x>p]
```

ANEXO – 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*.

Bibliografia & Papers :

- *Introduction to Functional Programming using Haskell*- Richard Bird, Prentice Hall International, 2nd Ed. New York, 1997
- Basic polimorphics typecheking – Luca Cardelli
- *A gentle introduction to Haskell*
- The craft of Functional Programming – *Simon Thompson*
- Introducción al lenguaje Haskell - *Jose E. Labra G.*
- http://www.haskell.org