HASKELL & HUGS

```
apaz@characatux:~> hugs
__   __ __  __  ____   ___      ________________________________________
||   || ||  ||  ||  ||  ||__     Hugs 98: Based on the Haskell 98 standard
||___|| ||__||  ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: 20041024       ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs.Base> _
```

Haskell
A Purely Functional Language

Mientras que Haskell es un lenguaje de programación funcional de alto nivel, Hugs es un interprete que entiende el lenguaje Haskell.

Haciendo clik encima del número del índice se dirige  directamente al tema.

Propósito General:

El propósito de realizar ésta presentación, no es la exposición dogmática y docta del tema Haskell & Hugs, sino todo lo contrario, acercar al estudiante que por primera vez se ve enfrentado a un lenguaje de programación, sin tener formación previa y requiere de una terminología específica acompañada de las definiciones lo más allegadas a su entendimiento posible para acercarse a la abstracción del lenguaje computacional.

LENGUAJES:

Lenguaje Natural = lenguaje humano, es un conjunto arbitrario de símbolos, socialmente aceptados, no instintivo de comunicar ideas, es un instrumento de expresión y significación social, construido socialmente.

Lenguaje Computacional = Los lenguajes de programación son un conjunto acotado (finito) de palabras y signos , interrelacionados por medio de reglas. Son lenguajes inventados para controlar las respuestas de una máquina dada.

Clasificación de Lenguajes Computacionales

Los lenguaje de computación se pueden clasificar de diversas maneras. A los efectos del tema los clasificaremos en dos grandes grupos:

Lenguajes Imperativos y Lenguajes Funcionales

VOLVER AL INDICE

Lenguajes Imperativos

Lenguajes de Programación IMPERATIVA o Tradicional*, dónde lo importante es la secuencia de pasos , acciones y condiciones, para llegar a un resultado previsible. Se basa en estructuras, variables e instrucciones de repetición.

Sentencias Imperativas

```
x := 5
```

Sentencias declarativas (funcionales)

```
function f(int x) { return x+1;}
```

La distinción entre construcciones imperativas y declarativas se basa en la distinción entre cambiar un valor existente y declarar un nuevo valor.

```
{ int x=1; /* declara una nueva variable x */
x = x+1; /* asigna un valor a x */
{ int y=x+1; /* declara una nueva variable y */
{ int x=y+1; /* declara una nueva x */
}}}
```

VOLVER AL INDICE

Lenguajes Imperativos:

Los lenguajes de programación tradicionales como **Pascal , C, C++, ADA**, **Java,** entre otros forman una abstracción de la máquina de Von-Neumann caracterizada por:

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

Los programas imperativos poseen una serie de datos globales y una secuencia de comandos ó código. Estos dos elementos forman una abstracción de los datos y código de la memoria principal. Para hacer efectiva dicha abstracción se compila el código fuente para obtener código máquina.

El modelo imperativo tiene gran proximidad a la arquitectura de los computadores convencionales.

VOLVER AL INDICE

Lenguajes Imperativos:

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

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

Las instrucciones de acceso a los datos globales destruyen el contenido previo de un dato asignando un nuevo valor. Las asignaciones introducción al lenguaje Haskell producen una serie de efectos laterales que oscurecen la semántica del lenguaje.

VOLVER AL INDICE

LENGUAJES FUNCIONALES de PROGRAMACIÓN:

Lenguajes FUNCIONALES o Programación Funcional, tienen por fundamento las funciones como estructuras de control.

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

Se caracterizan por la utilización de funciones sobre los elementos de primer orden, así como de la utilización de funciones polimórficas, funciones de orden superior, evaluaciones perezosas y definiciones de listas por comprehensión.

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

VOLVER AL INDICE

LENGUAJES FUNCIONALES de PROGRAMACIÓN:

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

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

Se caracterizan por la utilización de funciones sobre los elementos de primer orden, así como de la utilización de funciones polimórficas, funciones de orden superior, evaluaciones perezosas y definiciones de listas por comprehensión.

VOLVER AL INDICE

Para QUÉ o QUIÉNES utilizan Lenguajes Funcionales:

- Software AG, una de las mayores empresas de software en Alemania vende un sistema experto.
- Ericsson ha desarrollado un nuevo lenguaje de programación funcional Erlang®, que se utilizará en sus aplicaciones de teléfonos celulares.
- Query® es el lenguaje de sistema de base de datos orientado a objetos O2
- ICAD Inc. Market un sistema CAD para ingeniería mecánica y aeronáutica.

VOLVER AL INDICE

GLOSARIO INICIAL:

1. **Elementos de primer orden (first class)**: o Argumentos, son el conjunto de instrucciones lógico-matemáticas, muy precisas, para resolver la función (1ª línea).

2. Tipo o Tipos de datos, es un atributo del conjunto de los datos o valores(dominio) que indica la clase de datos sobre los que se van a procesar. Los tipos específicos se nombran con mayúscula al inicio y los que denotan valores con minúscula: Integer, Bool, Char, …

3. Tipificación (Typing): Es la relación del dato o valor con su dominio, ej.: 5:: Integer ; ´a´ :: Char; inc:: Integer -> Integer; [1,2,3] :: [Integer]; (`b´, 4) :: (Char, Integer)

4. Tipo de una función (Type signature declaration): , es con la cual podemos dar de forma explícita el tipo de una función: inc, inc :: Integer-> Integer (inc=increase)

5. **Tipos Polimórficos**, son cuantificadores universales sobre todos los tipos y describen esencialmente familias de tipos, ej.: (para _todo a), [a] familia de listas que contienen a.

VOLVER AL INDICE

GLOSARIO INICIAL:

6. Funciones Polimórficas: Son aquellas que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta y tienen sentido para más de un tipo. [Integer], [Char], o [[Integer]].length [1,2,3] = > 3 length [`a´, `b´, `c´] = > 3 length [[1], [2], [3]] = > 3

6. Tipo definido por el usuario, debe iniciarse también con minúscula, seguido de del tipo de dato, ej.: miEvento :: Int -> Bool (tipo de chequeo)

6. **Evaluaciones Perezosas**: significa "haz lo solo que que te pida un patrón" a la izquierda de una ecuación o cuantificador (Where o Let). No evalúes toda la expresión (evaluación no estricta).

7. **Encaje por patrones**: Las funciones son un tipo de operadores de prioridad mayor regido por la asociatividad izquierda que establece patrones que cumplen con las propiedades matemáticas tradicionales: Conmutativa, asociativa, etc.

8. **Listas por comprehensión:** El operador no está obligado a declarar el tipo de expresiones, el compilador contiene un algoritmo que infiere el tipo de las expresiones. Si el programador declara el tipo de alguna expresión el sistema corrobora si el tipo declarado coincide con el tipo inferido.

Ej.: eligeSaludo x = if x then "adios"
else "hola" el compilador notará
eligeSaludo :: bool -> String (arrays)

VOLVER AL INDICE

HASKEL

LENGUAJE FUNCIONAL de PROGRAMACIÓN:

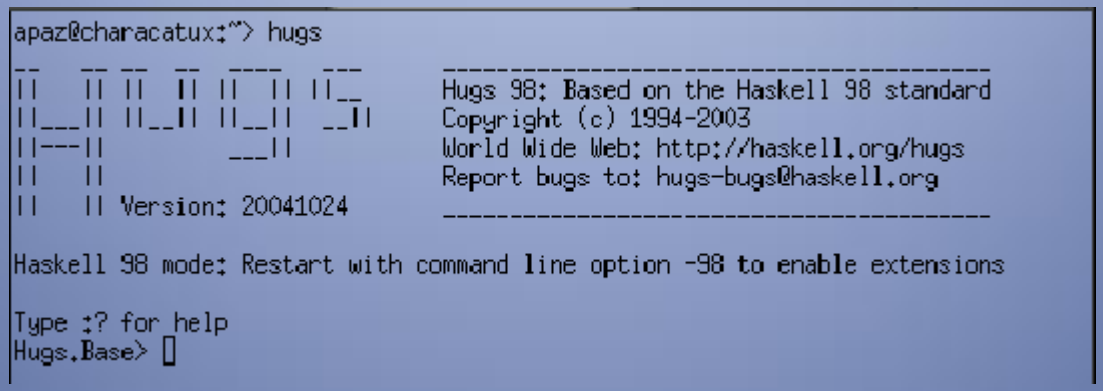
El lenguaje **Haskell (1987-1992) pretendió unificar las características más importantes de los** lenguajes funcionales como las funciones de orden superior, evaluación perezosa, inferencia estática de tipos, tipos de datos definidos por el usuario, encaje de patrones y listas por comprehensión. Al diseñar el lenguaje se observó que no existía un tratamiento sistemático de la sobrecarga con lo cual se construyó una nueva solución conocida como las clases de tipos. El lenguaje incorporaba, además, Entrada/Salida puramente funcional y definición de *arrays* por comprehensión.

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

Posteriormente, surge la versión 1.4 con ligeras modificaciones.

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

VOLVER AL INDICE



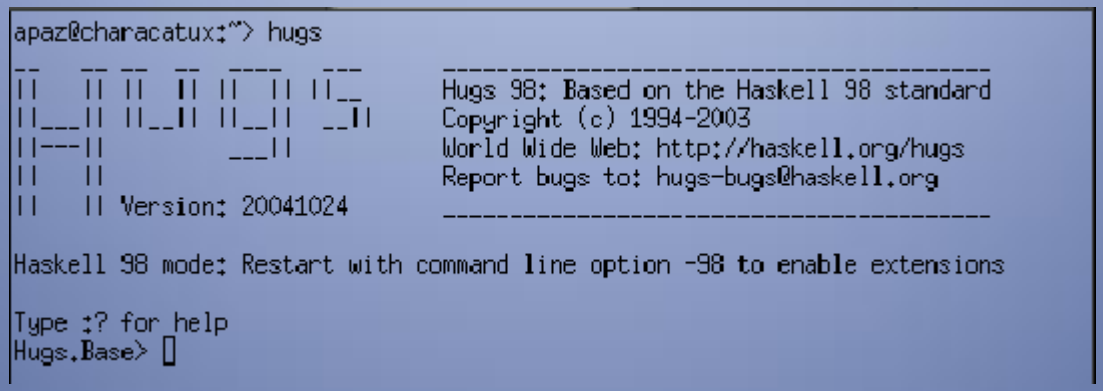
Cuando se inicia WinHugs 98, se obtiene una ventana de sesión.

Después del mensaje de bienvenida, al inicio cada línea verá la palabra Hugs98 esto es un *prompt*, un indicador del estado de procesamiento.

En esta ventana podrá digitar cualquier tipo de *expresión matemática* y podrá ver el resultado de esta presionando intro (*enter*). Una vez que el cálculo es computado y el resultado es mostrado, el *prompt* aparecerá nuevamente. Esto nos indicará que el intérprete está listo para recibir más instrucciones.

No se debe olvidar que el lenguaje o notación en que se escriben las expresiones es Haskell.

VOLVER AL INDICE



- **Conceptos básicos**
- El entorno *HUGS funciona siguiendo el modelo de una calculadora en el que se establece una*
- sesión interactiva entre el ordenador y el usuario. Una vez arrancado, el sistema muestra un
- *prompt "?" y espera a que el usuario introduzca una expresión (denominada* ***expresión***
- **inicial y presione la tecla <RETURN>. Cuando la entrada se ha completado, el sistema**
- evalúa la expresión e imprime su valor antes de volver a mostrar el *prompt para esperar a que*
- se introduzca la siguiente expresión.
- **Ejemplo:** (2+3)*8
  40
- En el primer ejemplo, el usuario introdujo la expresión "(2+3)*8" que fue evaluada por
- el sistema imprimiendo como resultado el valor "40".

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

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

Por ejemplo, puede introducir algunas expresiones aritméticas simples:

```
Hugs> 1+3-2 2
-18
```

y también otras no tan simples:

```
Hugs> 15*0.30+20*0.30+19*0.40
18.1
```

Este último ejemplo calcula el promedio ponderado de tres notas, asignando un peso de 30% a las dos primeras y 40% a la última.

Hugs conoce muchas funciones y operadores que pueden ser usados en estas expresiones. Ellas están definidas en un archivo llamado Prelude, el cual sólo es un programa Haskell (Prelude .hs) que contiene una colección de definiciones de funciones.

Haskell es un lenguaje funcional con un **SISTEMA DE INFERENCIA DE TIPOS** que consiste en:

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

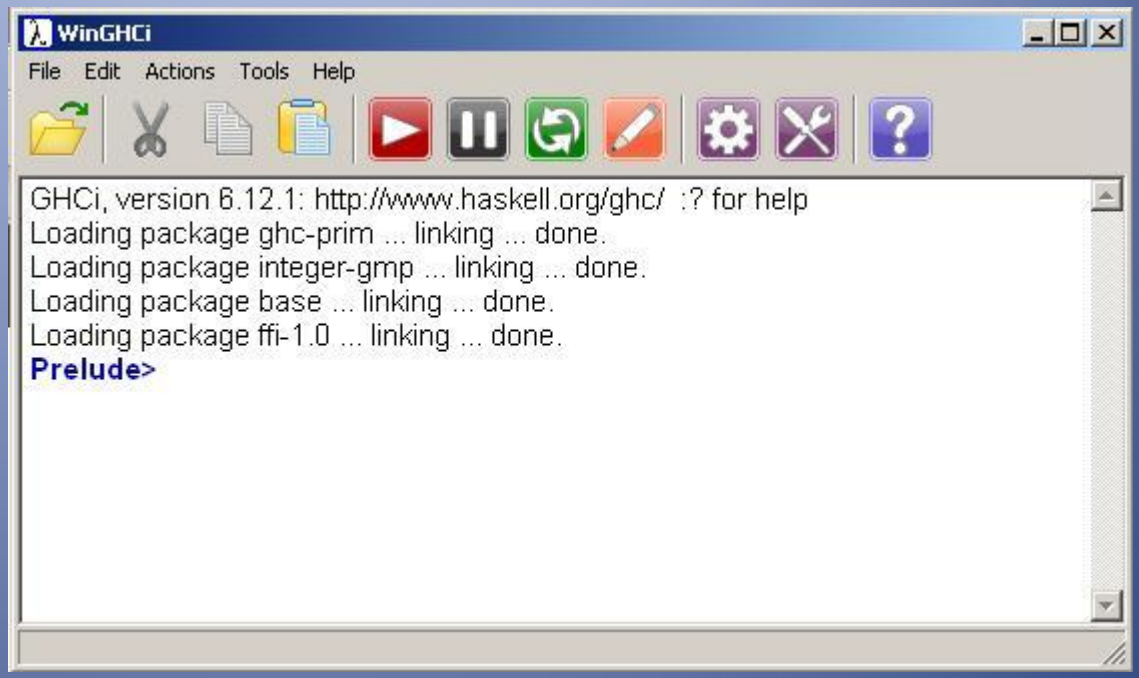
Los **sistemas de inferencia de tipos** permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.
Por ejemplo, si el programador declara la siguiente función:

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

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

Haskell Platform 2011.2.0.0 versión GHCi

La versión Haskell Platform 2011.2.0.0 GHCi (Glasgow Haskell Compiler), provee un entorno similar a las versiones anteriores, pero con un acceso inmediato a Prelude, lo que nos proporciona una variedad mucho más amplia de soluciones preestablecidas. En adelante nos referiremos siempre respecto a la versión más actualizada.



En la nueva plataforma de Haskell, llamada WinGHCi 2011, la cual evolucionó desde la plataforma Winhugs 98, cuya versión actual es la 7.0.2,se tiene acceso directamente a una ventana de sesión con una amplia colección de valores, funciones, tipos y operadores predefinidos bajo WinGHCi Prelude (Prelude = preludio, precede y sirve de entrada), como evaluador.

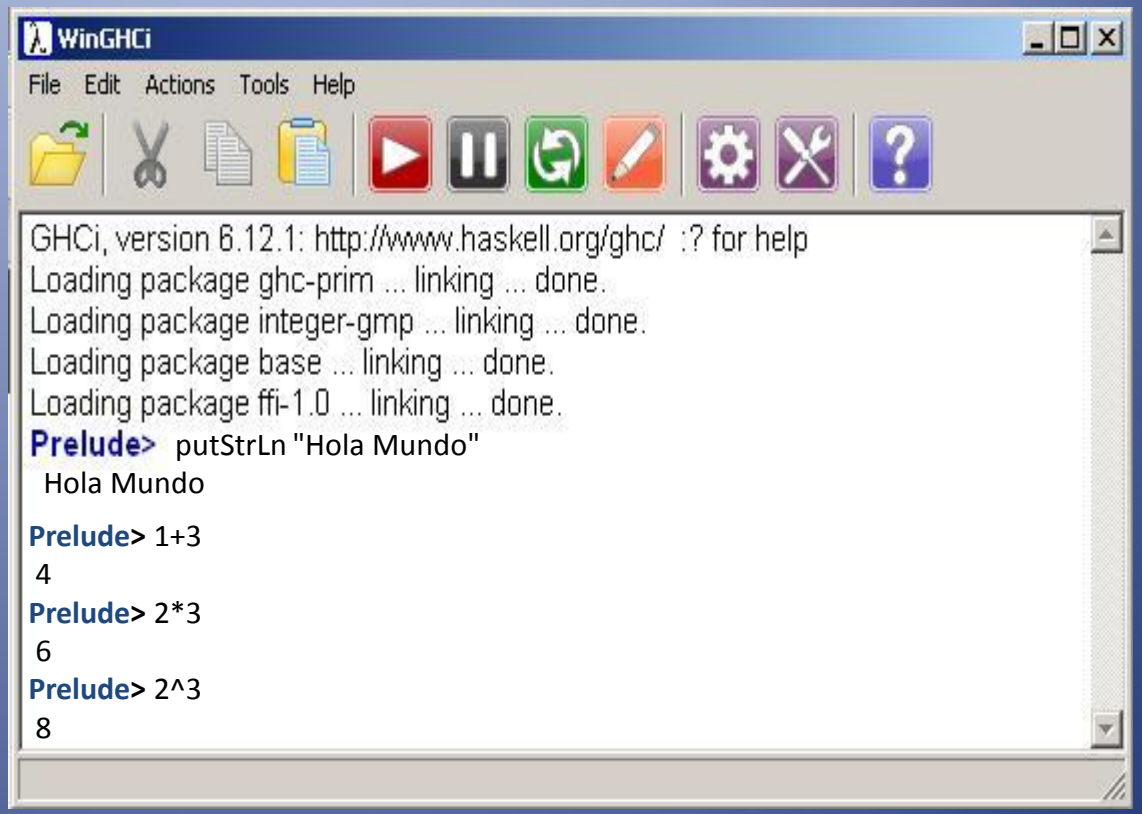
VOLVER AL INDICE

Haskell Platform 2011.2.0.0 versión GHCi

Prelude> es nuestra línea de comandos donde podemos ejecutar directamente los programas.

El programa que muestra "Hola Mundo" en Haskell es:

Prelude> putStrLn "Hola Mundo"
Hola Mundo



Lo más sencillo que podemos plantear luego del "Hola Mundo" es la ejecución de operaciones matemáticas:

```
Prelude> 1+3
4
Prelude> 2*3
6
Prelude> 2^3
8
```

VOLVER AL INDICE

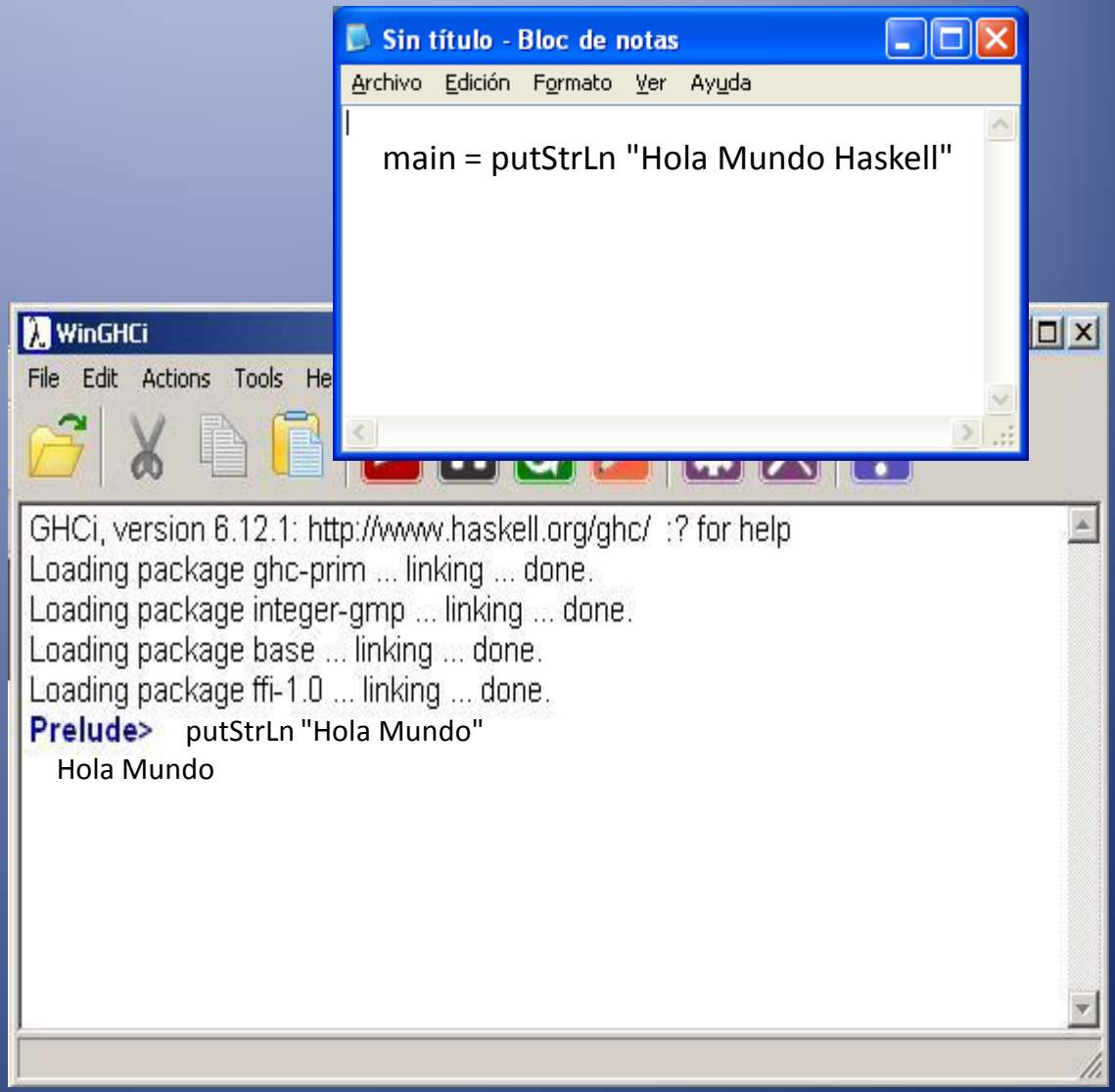
Podemos también generar un archivo independiente con nuestro programa en Haskell, utilizaremos el NotePad para ello:

main = putStrLn "Hola Mundo Haskell"

Luego de codificarlo lo grabamos con extensión hs (ejemplo1.hs) y desde el entorno de Haskell procedemos a recuperar el archivo mediante las opción:

File -> Load y lo ejecutamos mediante la opción Actions -> Run"'main"

Como vemos es muy sencillo proceder a crear un archivo independiente con nuestro programa en Haskell (también podemos generar un ejecutable (exe) seleccionando la opción Tools -> GHC compiler)



VOLVER AL INDICE

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:

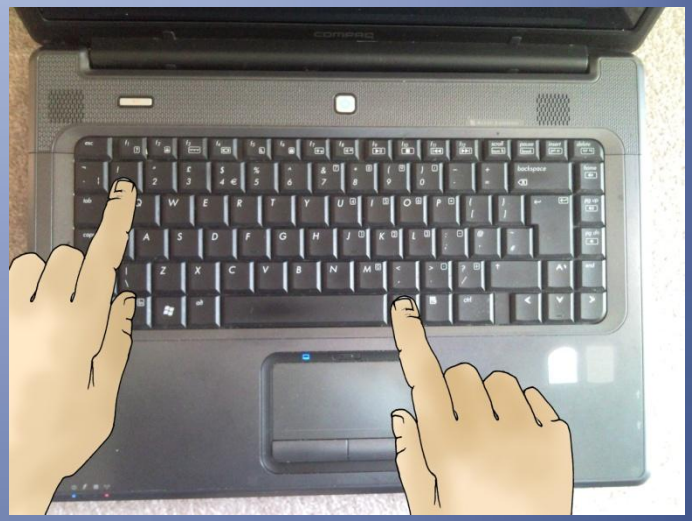
Un **identificador** comienza con una letra del alfabeto seguida, opcionalmente, por una secuencia de caracteres, cada uno de los cuales es, una letra, un dígito, un apóstrofe **(')** o un subrayado **(_)**.

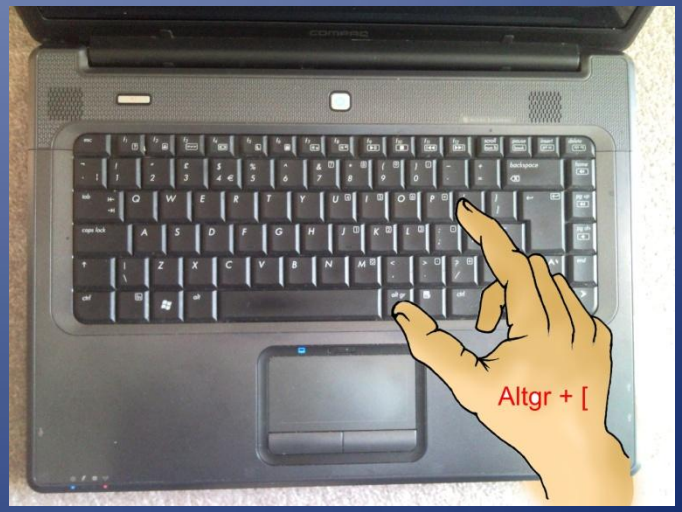
Los identificadores que representan funciones o variables deben comenzar por letra minúscula (los identificadores que comienzan con letra mayúscula se emplearán como funciones constructoras). Los siguientes son ejemplos de posibles identificadores:

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

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

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





Símbolo de OPERADOR

En orden de acceder a la simbología Haskell, nombraremos algunos de los **OPERADORES** y como localizarlos en el teclado.

SÍMBOLOS:

"tal que"	**|** = Alt Gr + 1
"negativo"	**¬** = Alt Gr + 6
"paréntesis recto"	**[** = Alt Gr + [
"paréntesis recto"	**]** = Alt Gr +]
"denominador letra"	'a' = ' '
"potencia"	2^3 = 2^3

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

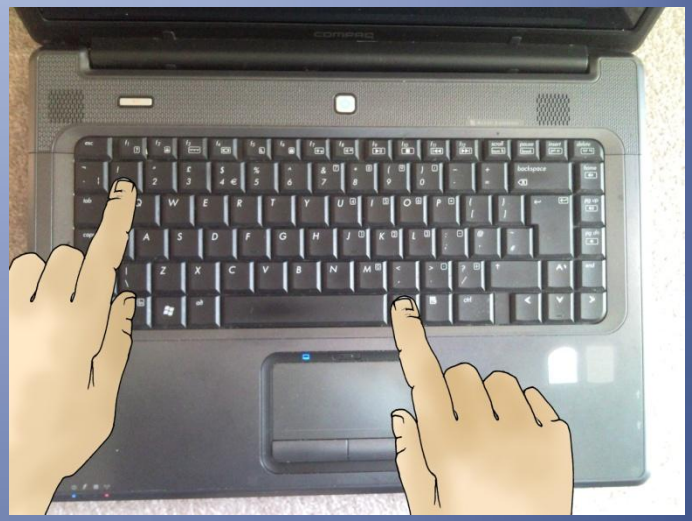
: ! # $ % & * + . / < = > ? @ \ ^ | -

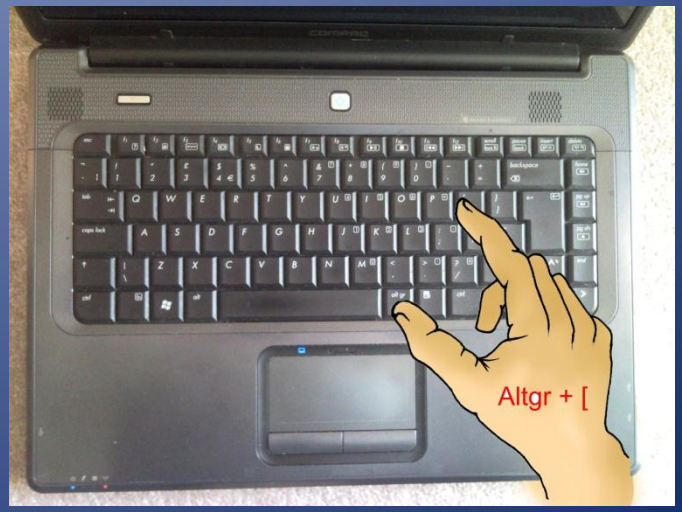
El carácter **(~)** se permite, aunque sólo en la primera posición del nombre. Los nombres de operador que comienzan con **(:)** son utilizados para funciones constructoras como los Identificadores que comienzan por mayúscula mencionados anteriormente. Los siguientes símbolos tienen usos especiales:

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

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

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





Identificadores y Operadores:

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

Cualquier identificador será tratado como un símbolo de operador si está encerrado entre comillas inversas (`):

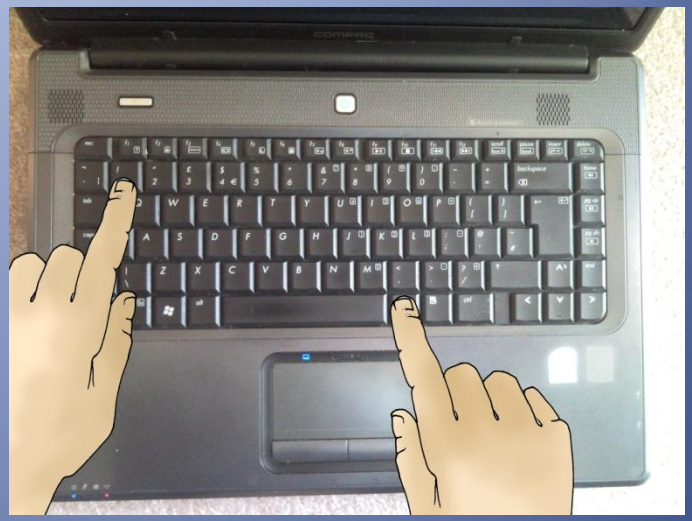
"x `id` y" es equivalente a "id x y"

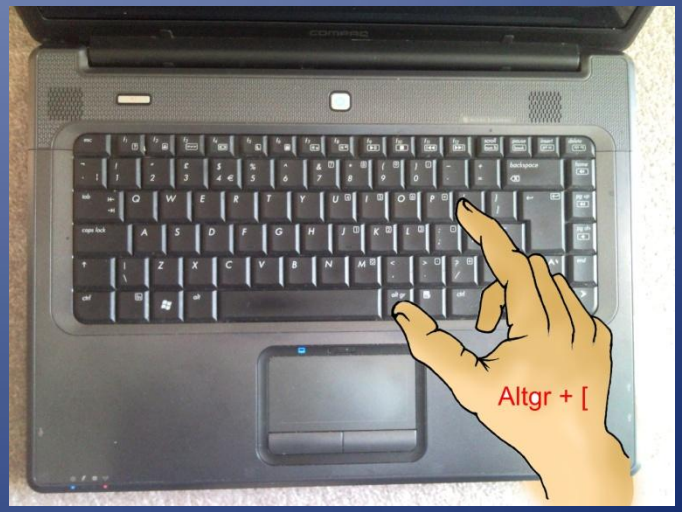
Cualquier símbolo de operador puede ser tratado como un identificador encerrándolo en paréntesis.

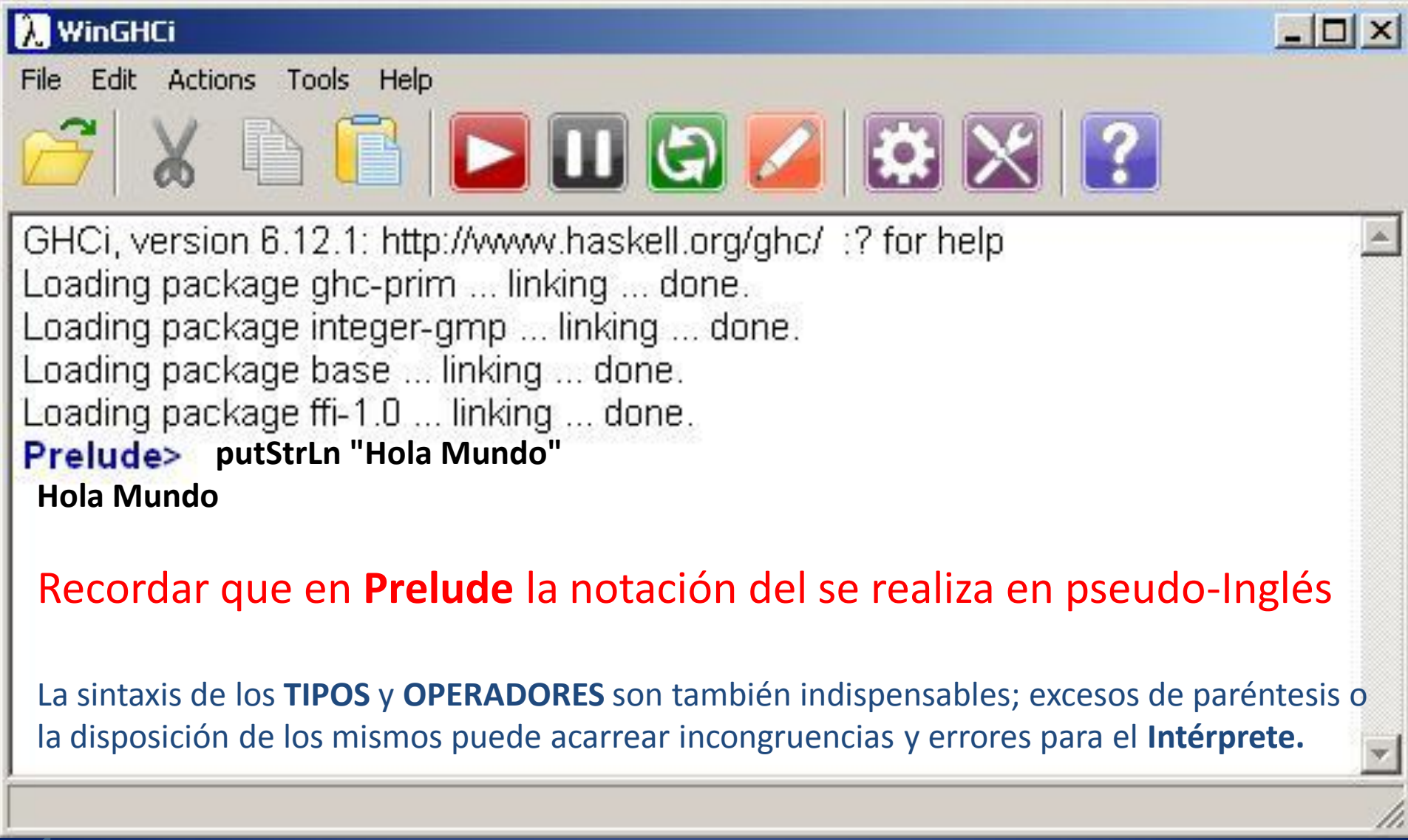
"x + y" podría escribirse como "(+) x y“

Cuando se trabaja en Standard Prelude con símbolos de operador es necesario tener en cuenta la

PRECEDENCIA y la **ASOCIATIVIDAD**







Recordar que en **Prelude** la notación del se realiza en pseudo-Inglés

La sintaxis de los **TIPOS** y **OPERADORES** son también indispensables; excesos de paréntesis o la disposición de los mismos puede acarrear incongruencias y errores para el **Intérprete.**

PRECEDENCIA y ASOCIATIVIDAD

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

B. La **asociatividad**: La regla anterior resolvía ambigüedades cuando los símbolos de operador tienen distintos valores de precedencia, sin embargo, la expresión "1 - 2 - 3" puede ser tratada como "(1 - 2) - 3" resultando -4 o como "1 - (2 - 3)" resultando 2. Para resolverlo, a cada operador se le puede definir una regla de asociatividad. Por ejemplo, el símbolo (-) se puede decir que es:

Asociativo a la izquierda: si la expresión "x-y-z" se toma como "(x-y)-z"
Asociativo a la derecha: si la expresión "x-y-z" se toma como "x-(y-z)"
No asociativo: Si la expresión "x-y-z" se rechaza como un error sintáctico.

En el *standar prelude* el **(-)** se toma como asociativo a la izquierda, por lo que la expresión **"1 - 2 - 3" se tratará como "(1-2)-3".**
Por defecto, todo símbolo de operador se toma como no-asociativo y con precedencia 9.
Estos valores pueden ser modificados mediante una declaración con los siguientes formatos:

infixl digito ops Para declarar operadores asociativos a la izquierda
infixr digito ops Para declarar operadores asociativos a la derecha
infix digito ops Para declarar operadores no asociativos

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

Existen ciertas restricciones en la utilización de estas declaraciones:

- Sólo pueden aparecer en ficheros de definición de función que sean cargados en el sistema

- Para un operador particular, sólo se permite una declaración

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

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

Tabla de precedencia/asociatividad de operadores: **Expresión: Equivalente a: Motivos**

1+2-3 (1 + 2) - 3 (+) y (-) tienen la misma precedencia y son asociativos a la izquierda.
x : ys ++ zs x : (ys ++ zs) (:) y (++) tienen la misma precedencia y son asociativos a la derecha.
x == y || z (x == y) || z (==) tiene más precedencia que (||)
3+4*5 3+(4*5) (*) tiene más precedencia que (+)
y **`elem` z:zs** y **`elem` (z:zs)** (:) tiene más precedencia que `elem`
12 / 6 / 3 error sintáctico (/) no es asociativo
"f x + g y" equivale a **"(f x) + (g y)"** las funciones tiene más precedencia que cualquier símbolo de operador. Por ejemplo, la expresión "f x + g y" equivale a "(f x) + (g y)"
"f x + 1", equivale y es tratada como **"(f x)+1"** en lugar de "f(x+1)".

MATERIAL REFERENCIAL

Varios Haskell Commision*	The Haskell 98 Report	Simon Peyton Jones
RICHARD BIRD	Introduction to Funcional..	Pentice Hall
CRISTIANO M. GASTÓN	Haskell, Lenguajes ya	WordPress 2010®
JOSÉ R. MARCIAL ROMERO	Programación Funcional	UAEMex
ALFREDO PAZ VALDERRAMA	Introducción a la Progr.	UCSP.Pe
JOSÉ A. ALONZO JIMÉNEZ	Funciones de Ord. Súper.	Universidad de Sevilla
PEPE GALLARDO	Haskell	Universidad de Málaga
BLAS C. RUIZ y Otros	Razonando con Haskell	Universidad de Málaga
SIMON THOMPSON	The craft of functional programming (extracto)	
JOSÉ E. LABRA	Introd. Al Lenguaje Haskell	Universidad de Oviedo
JUAN P. VILLA IZASA	Introd. Al Lenguaje Haskel	
ANDRÉS y Otros	Definiciones de Tipos en Haskell	
PACO GUTIERREZ y Otros	Una introducción agradable…	Universidad de Málaga
PAQUI LUCIO	Tipos y Clases	SC.EHU
JULIO J. JAVIER	Haskell, introducción…	UTN. Fac. Córdoba
MAURO JASKELIOFF	Evaluación perezosa	Faceia.UNR

VOLVER AL INDICE

MATERIAL REFERENCIAL

Varios Haskell Commision* The Haskell 98 Report Simon Peyton Jones

Simon Peyton Jones [editor], Microsoft Research, Cambridge
Lennart Augustsson, Sandburst Corporation
Dave Barton, Intermetrics
Brian Boutel, Victoria University of Wellington
Warren Burton, Simon Fraser University
Joseph Fasel, Los Alamos National Laboratory
Kevin Hammond, University of St. Andrews
Ralf Hinze, University of Bonn
Paul Hudak, Yale University
John Hughes, Chalmers University of Technology
Thomas Johnsson, Chalmers University of Technology
Mark Jones, Oregon Graduate Institute
John Launchbury, Oregon Graduate Institute
Erik Meijer, Microsoft Corporation
John Peterson, Yale University
Alastair Reid, University of Utah
Colin Runciman, York University
Philip Wadler, Avaya Labs

HASKELL & HUGS

Propósito General:

El propósito de realizar ésta presentación, no es la exposición dogmática y docta del tema Haskell & Hugs, sino todo lo contrario, acercar al estudiante que por primera vez se ve enfrentado a un lenguaje de programación, sin tener formación previa y requiere de una terminología específica acompañada de las definiciones lo más allegadas a su entendimiento posible para acercarse a la abstracción del lenguaje computacional.

Todos los aportes posibles, serán recibidos y analizados para su posible incorporación en la reedición de este material.
Correo de recepción de aportes: adolfomontielvalentini@hotmail.com

Análisis y compilación del material ante expuesto:

Adolfo Montiel Valentini ® 2011

Prof. Saúl Tenembaum – Matematica Discreta I
Instituto Normal de Enseñanza Técnica (INET)

VOLVER AL INDICE