
Universidad de Huelva

# Práctica 1

# Lenguajes funcionales

# Índice

# Índice

**Funciones puras**

- La definición matemática de ***función*** (o ***aplicación***) establece que una función representa una correspondencia entre dos conjuntos que asocia a cada elemento en el primer conjunto un único elemento del segundo.
- Esto quiere decir que si evaluamos una función *f* sobre un valor *x* y se genera el resultado *a* (es decir, *f(x)=a*) entonces este resultado será válido siempre. Es más, se puede sustituir *f(x)* por *a* en cualquier expresión.
- Por ejemplo, si decimos que la raiz cuadrada de 4 es 2 quiere decir que este resultado es válido siempre.

**Funciones puras**

- En los lenguajes de programación clásicos (*imperativos*) se definen funciones (o métodos, o constructores, …), pero no se respeta esta característica de la definición matemática.
- Por ejemplo, el código

```
a = f(x);
b = f(x);
```

  puede generar valores diferentes para las variables *a* y *b*. Esto se debe a que la función *f* podría tener efectos colaterales (como modificar variables globales, por ejemplo).
- De hecho, algunos autores prefieren hablar de *procedimientos* y evitar el término *función* al describir los lenguajes de programación.

**Funciones puras**

- Se denomina *razonamiento ecuacional* al razonamiento lógico que utiliza la condición de inmutabilidad de las funciones.
- Para poder sacar provecho a este tipo de propiedades es necesario restringir las reglas de definición de funciones para que no puedan tener efectos colaterales.
- Esto afecta sobre todo al concepto de *variable*. En los lenguajes clásicos (*imperativos*) una *variable* es un contenedor que puede almacenar diferentes valores en diferentes momentos. Sin embargo, la noción matemática de *variable* se refiere a un cierto valor que puede ser conocido o desconocido, pero que es inmutable.

## Funciones puras

- Se denomina ***función pura*** al tipo de función que sigue exactamente las nociones matemáticas y utiliza variables inmutables.
- Para trabajar con variables inmutables es necesario prohibir las instrucciones de asignación. Una variable solamente puede ser asignada una vez (en su declaración).
- También es necesario prohibir los bucles, ya que no pueden existir variables índices que modifiquen su valor hasta alcanzar la condición de finalización del bucle.
- El concepto de razonamiento ecuacional indica que si una expresión se evalúa y genera un cierto resultado, entonces ese resultado es también inmutable. No tiene sentido utilizar bucles que evalúen una y otra vez la misma expresión.

## Funciones puras

- Para poder programar sin bucles es necesario hacer un uso intensivo de la recursión.
- Por ejemplo, en un lenguaje imperativo, el factorial se puede calcular de la siguiente forma:

```
int factorial( int x )
{
  int index = 1;
  int fact = 1;
  while(index < x)
  {
    index = index +1;
    fact = fact * index;
   }
  return fact;
}
```

## Funciones puras

- En un lenguaje que utilize funciones puras, el factorial se puede calcular de la siguiente forma:

```
int factorial( int x )
{
  if(x <= 1) return 1;
  else return x * factorial(x-1);
}
```

## Funciones de orden superior

- Se denominan ***funciones de orden superior*** a las funciones que pueden ser utilizadas como datos.
- El uso de funciones de orden superior implica que el valor de una cierta variable o el resultado de una expresión podría ser una función. De igual forma, se pueden utilizar funciones como argumentos de otras funciones y como resultado de una función.
- Para poder utilizar funciones como datos es necesario definir el tipo de dato. Por ejemplo, un tipo de dato podría ser "*funciones con un argumento entero que generan como resultado un valor entero*".
- Para definir estos tipos de datos se suele utilizar un operador "*flecha*", que se suele considerar asociativo a la derecha:

**int -> int**

## Funciones de orden superior

- Para facilitar el uso de tipos de datos funcionales se suele añadir alguna forma de declaración de tipo, que permita asociar un identificador a un cierto tipo de dato funcional. Por ejemplo,

```
typedef   intfun  =  int -> int;
```

- La sintaxis de declaración de tipos de datos sería, por ejemplo,

```
TypeDecl ::=  typedef   id  equal  TypeExpr  semicolon
TypeExpr ::= ( TypeList   arrow )* TypeList
TypeList ::=  TypeBase  ( comma TypeBase )*
TypeBase ::= Type  |  lparen TypeExpr  rparen
```

**Funciones de orden superior**

- Siguiendo esta sintaxis, el tipo de dato

```
( int , String ) ->  int[]
```

denota una función con dos argumentos (un valor entero y una cadena de caracteres) que genera como resultado un array de enteros.

- De igual forma, el tipo de dato

```
int -> int -> int
```

denota una función con un argumento de tipo entero que genera como resultado otra función. Esta segunda función toma como argumento un valor entero y genera como resultado un valor entero.

## Funciones de orden superior

- Ejemplo

```
typedef   IntFun =  int -> int;

public int add5(int x)
{
  return x+5;
}

public int twice(IntFun f, int x)
{
  return f( f(x) );
}

public int add10(x)
{
  IntFun f = add5;
  return twice(f, x);
 }
```

## Funciones de orden superior

- El uso de funciones como dato se puede resolver en muchos lenguajes por medio de punteros a funciones. En lenguaje C, el siguiente código permite asignar la función *suma* a una variable local, *ptr*, de tipo puntero a función. Esta variable puede ser invocada como una función.

```
int suma(int a, int b) { return a+b; }

void ejemplo() {
  int (* ptr) (int,int) = suma;
  int result = ptr(5,7);
}
```

- Internamente, un puntero a función expresa la posición de la etiqueta de comienzo del código de la función.

## Funciones de orden superior

- En un lenguaje orientado a objetos, como Java, podríamos intentar desarrollar funciones de orden superior por medio de interfaces. De esta forma, la declaración

```
typedef   intfun =  int -> int;
```

- Podría simularse como

```
interface  intfun {
   int compute (int x);
}
```

- Una función de orden superior sería un objeto de una clase que cumple esta interfaz.

## Funciones anidadas

- Se denominan ***funciones anidadas*** a funciones que pueden ser definidas dentro del cuerpo de otras funciones, pudiendo acceder a los valores de los argumentos y variables locales de dicha función (lo que se conoce como su *ámbito léxico* o *lexical scope*).

## Funciones anidadas

- Ejemplo

```
typedef   IntFun =  int -> int;

public IntFun add(int n)
{
  public int h(int m) { return n+m; }
  return h;
}

public IntFun twice(IntFun f)
{
  public int g(int x) { return f( f(x) );  }
  return g;
}

public int test()
{
    return twice( add(5) ) (7);
 }
```

## Lenguajes funcionales

- Se denominan ***lenguajes funcionales*** a los lenguajes de programación que soportan funciones de orden superior que admiten funciones anidadas con ámbito léxico. Ejemplos de este tipo de lenguajes son *Scheme*, *ML* o *Smalltalk*.
- Se denominan ***lenguajes funcionales puros*** a los lenguajes de programación que incluyen funciones de orden superior y solo admiten la definición de funciones puras. Por ejemplo, el subconjunto funcional puro de *ML* o el lenguaje *Haskell*.
- También existen lenguajes de programación que solo admiten funciones puras pero no soportan las funciones de orden superior. Por ejemplo *SISAL*.

## Lenguajes funcionales

- Los lenguajes funcionales se suelen dividir en ***lenguajes estrictos*** y ***no estrictos*** en función del tipo de evaluación de expresiones que utilice.
- Se denomina ***evaluación estricta*** a la técnica de programación en la que en tiempo de ejecución una expresión siempre es evaluada y sustituida por su valor.
- Se denomina ***evaluación perezosa*** (***lazy***) o no estricta a la técnica de programación en la que las expresiones solo son evaluadas cuando es necesario utilizar su valor. En tiempo de ejecución las expresiones pueden no ser evaluadas si no son requeridas para ello.

## Expansión de funciones

- Los programas funcionales tienden a utilizar muchas funciones de pequeño tamaño. Esto provoca que en tiempo de ejecución utilicen la pila de forma muy intensa.
- Una forma de reducir el uso de la pila es sustituir la llamada a una fución por la copia directa de su código. Es lo que se conoce como ***inline expansion***.

## Expansión de funciones

- Si consideramos funciones puras el código de una función equivale a una expresión (ya que no hay asignaciones ni bucles). Esto hace que sea sencillo expandir funciones cuando equivalen a expresiones sencillas.
- La expansión de funciones recursivas puede generar bucles infinitos por lo que no es aconsejable.
- Para evitar la explosión de código se utilizan diferentes heurísticas, como solo expandir funciones pequeñas o que solo aparezcan una vez. También se puede eliminar las funciones que dejen de utilizarse al haber sido expandidas.

## Expansión de funciones

- Ejemplo

```
int g(int x) {
   return x + 5;
}

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

```
int g(int x) {
   return x + 5;
}

int f(int x) {
  return { return 1 + 5; } + x;
}
```

Si la función *g* no vuele a utilizarse, puede ser eliminada.

## Recursión de cola

- La programación funcional hace un uso intensivo de la recursión como mecanismo para sustituir a los bucles.
- Se denomina ***recursión de cola*** o ***tail-recursion*** a una construcción en la que lo último que se ejecuta en una función es la llamada a otra función.

## Recursión de cola

- Por ejemplo

```
int A()
{
  int x = C;
  return B(x);
}
```

- En tiempo de ejecución se cargaría el registro de activación de *A*, se ejecutaría el código *C*, se ejecutaría la llamada a *B* y por último se desapila el registro de activación de *A*. El último paso consiste en recoger el valor devuelto de *B* y copiarlo como valor devuelto de *A*. La llamada a *B* requiere cargar el registro de activación de *B* y descargarlo a su fin. Cuando la llamada a *B* es una recursión, la pila puede llegar a desbordarse si la profundidad de recursión es muy grande.

## Recursión de cola

- Cuando aparece una recursión de cola se puede optimizar el uso de la pila si se desapila el registro de *A* y se indica a *B* que su dirección de retorno es la de *A* y su posición para el resultado es el de *A*. De esta forma la pila crece mucho menos y se ahorran multitud de cambios de contexto consecutivos.
- En muchas ocasiones estos pasos se reducen al único salto final, por lo que la recursión de cola resulta tan eficiente como un bucle en un lenguaje imperativo.

## Recursión de cola

- Ejemplo: la función factorial se puede escribir en forma de recursión de cola

```
int fact(n) {
  if (n == 0) return 1 ;
  else return n * fact(n-1) ;
}
```

```
int fact(n) {
  return tail_fact(n,1) ;
}

int tail_fact(n,a) {
   if (n == 0) return a ;
   else return tail_fact(n-1,n*a) ;
}
```

## Evaluación perezosa

- Uno de los principios del razonamiento ecuacional es que si sabemos que *f(x)=B* (siendo *B* cualquier expresión) entonces *f(E)* es equivalente a la expresión *B* si sustituimos en ella toda ocurrencia de *x* por *E*. Esto se conoce como ***sustitución-β***.

$$f(x) = B \quad \Rightarrow \quad f(E) \equiv B[x \mapsto E]$$

## Evaluación perezosa

- Consideremos estos dos programas

```
{
  int loop(int z) {
     return if(z>0) z; else loop(z);
  }

  int f(int x) {
      return if(y>8) x; else –y;
  }

  return  f(  loop( y )  );
}
```

```
{
  int loop(int z) {
     return if(z>0) z; else loop(z);
  }

  int f(int x) {
      return if(y>8) x; else –y;
  }

  return if(y>8) loop(y); else –y;
}
```

- En el de la derecha se ha realizado la sustitución-β de *f( loop(y) ).*
- ¡¡Sin embargo, si y=0 el programa de la izquierda da error mientras que el de la derecha devuelve 0!!

**Evaluación perezosa**

- El ejemplo anterior demuestra que la sustitución-β no afecta al resultado cuando no se producen situaciones de error, pero puede alterar el funcionamiento de un programa ante excepciones.
- Si el programador ha introducido conscientemente las excepciones, el uso de la sustitución-β o la expansión online puede causar efectos indeseados.
- La razón por la que se generan comportamientos diferentes es que en la primera versión se evaluaba la expresión *loop(y)* mientras que en la segunda no. La sustitución-β demuestra que no era necesaria la evaluación de *loop(y)* y por esa razón la versión de la derecha no da error.

## Evaluación perezosa

- Se denomina **evaluación perezosa** a la técnica de tratamiento de expresiones en la que solo se calcula el valor de una expresión cuando es necesario. Mientras no se requiere el valor de la expresión ésta no se evalua.
- Cuando se utiliza evaluación perezosa, la sustitución-β genera siempre los mismos resultados, evitando los problemas comentados anteriormente.
- La evaluación perezosa se considera una característica importante a introducir en los lenguajes funcionales si se va a hacer uso de la expansión online como técnica de aceleración.

## Currificación

- Una función pura se puede definir como una expresión que vincula a las variables de entrada de esa función. Aplicar una función sobre unos valores significa sustituir las variables de entrada por sus valores.
- Por ejemplo:

**suma(x , y) = { return x + y; }**

- Si aplicamos esta función a los valores 2 y 3 tendremos que

**suma(2,3) = { return 2+3; }**

- NOTA: Si necesitamos el valor de la expresión podemos evaluarla y obtener el resultado '5'. Si utilizamos evaluación perezosa el resultado es la expresión sin evaluar.

## Currificación

- Dado que aplicar una variable a una función equivale a sustituir la variable por el valor deseado, podemos plantearnos sustituir solamente una de las variables, por ejemplo

**suma(2, y) = { return 2+y; }**

- Esto quiere decir que al sustituir una única variable el resultado es una expresión que sigue teniendo una variable libre. Por tanto, el resultado de *suma(2,y)* es una función de la variable *y*.
- Si consideramos las funciones como un tipo de dato más (funciones de orden superior) lo que queremos decir es que el resultado de *suma(2)* es una función.

## Currificación

- En un lenguaje de programación 'normal' el tipo de dato de la función suma sería

**suma: (Int, Int) -> Int**

- Si queremos expandir las variables de una función una a una, el tipo de dato de la función suma sería

**suma: (Int -> ( Int -> Int) )**

- La aplicación de los valores 2 y 3 deberíamos entenderla como

**(suma (2) )(3)**

## Currificación

- La programación funcional asume que el operador de aplicación de una función es asociativo a la izquierda y, por tanto, se puede escribir sin paréntesis

**suma 2 3 ≡ (suma (2) )(3)**

- En la definición de tipos se asume que el operador flecha es asociativo a la derecha y, por tanto,

**Int -> Int -> Int ≡ (Int -> ( Int -> Int) )**

- La interpretación de funciones n-arias como funciones unarias que devuelven funciones (n-1)-arias se debe a Haskell B. Curry. Por este motivo esta técnica se conoce como **currificación** de funciones.
- El lenguaje Haskell también debe su nombre a este matemático.

- Los orígenes de la programación funcional se remontan a la definición del cálculo lambda como modelo de computación.
- El cálculo lambda fue propuesto por Alonzo Church y Stephen Kleene en la década de 1930.
- La ídea básica es que una función se puede describir como una expresión (expresión-lambda) en la que aparecen las variables de entrada y que la aplicación de la función a un valor no es más que la sustitución de ese valor en la expresión. El término lambda se introduce para denotar que el nombre de la función es indiferente y que lo importante es la expresión.
- Por ejemplo, la función "doble" puede expresarse como

$$\lambda\, x.\ x + x$$

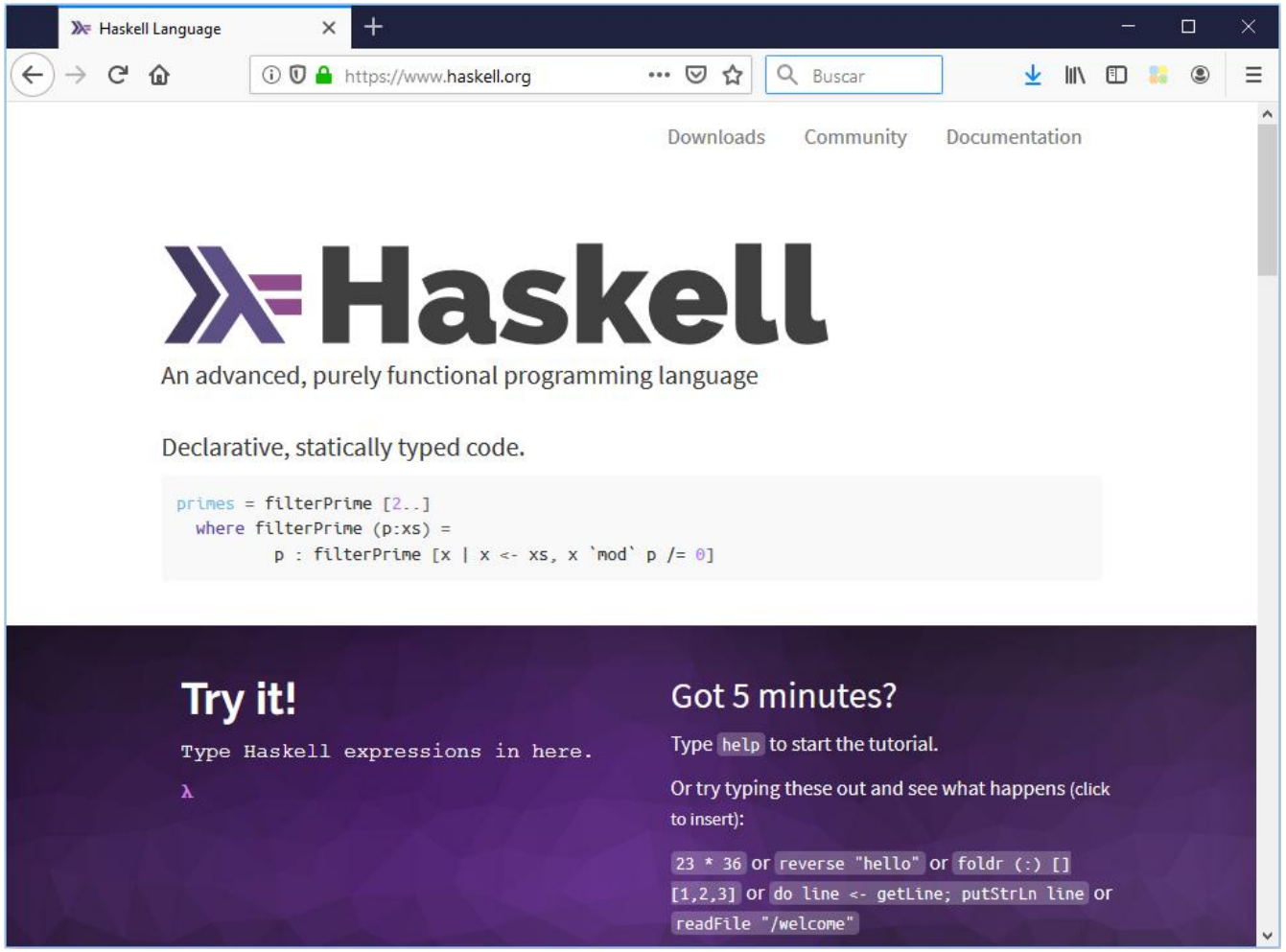
Primeros lenguajes funcionales:

- En 1958, John McCarthy definió el lenguaje LISP (acrónimo de List Processing). El lenguaje fue presentado en 1960 en la publicación "*Funciones recursivas de expresiones simbólicas y su cómputo a máquina, Parte I*". El lenguaje estaba basado en una interpretación del cálculo lambda aplicado a listas.
- A principios de los 1970s, Robin Milner desarrolla el lenguaje ML (acrónimo de MetaLenguaje). El objetivo del lenguaje era desarrollar técnicas de demostración para expresiones de predicados de primer orden y cálculo lambda.

- Durante la década de los 70s se desarrollaron los fundamentos de la programación funcional y se introdujeron conceptos como la inferencia de tipos o la evaluación perezosa.
- A mediados de los 80s existían numerosas propuestas de lenguajes funcionales a nivel académico. (ML, SASL, KRC, NPL, Hope, LazyML, Clean)
- En 1985 se desarrolla Miranda como primer lenguaje de programación funcional con carácter comercial.

- En la conferencia Functional Programming Languages and Computer Architecture de 1987 (FPCA'87) la comunidad científica decide formar un comité para definir un lenguaje funcional estandar.
- En 1990 se publica la primera versión (Haskell 1.0). Sobre esta versión se van sucediendo modificaciones enumeradas como 1.1, 1.2, 1.3 y 1.4.
- Finalmente se presenta Haskell 98 como versión estable del lenguaje.
- La última revisión es Haskell 2010
- Actualmente se trabaja en la siguiente revisión del estandar que será Haskel 2020.

- El portal oficial de Haskell es https://www.haskell.org/



- La plataforma de desarrollo oficial es GHC (Glasgow Haskell Compiler).
- La plataforma contiene los siguientes módulos:
  - **ghc**: compilador
  - **ghci**: intérprete
  - **cabal**: para construir paquetes y bibliotecas
  - **stack**: para desarrollar proyectos

- Comentarios de una línea:

  Comienzan con un doble guión y terminan en el final de línea.

```
-- esto es un comentario de una línea
```

- Comentarios multilínea anidados:

  Comienzan con el lexema {-- y terminan con el lexema --}. Dentro de un comentario multilínea puede aparecer otro comentario anidado.

```
{--
     Esto es un comentario multilínea
      {--  que contiene otro comentario anidado --}
       y termina aquí.
--}
```

- Palabras reservadas:

  **case, class, data, default, deriving, do, else, foreign, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type, where, _**

- Identificadores que comienzan en mayúscula:

  **[“A”-”Z”] ( [“A”-”Z”, ”a”-”z”, “0”-”9”, “_”, “\’”] )***

  (Se utilizan como identificadores de módulos, clases y constructores)

- Identificadores que comienzan en minúscula:

  **[“a”-”z”, “_”] ( [“A”-”Z”, ”a”-”z”, “0”-”9”, “_”, “\’”] )***

  (Se utilizan como identificadores de variables y funciones)

- Operadores reservados:

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

- Se pueden definir nuevos operadores utilizando combinaciones de los símbolos:

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

- Separadores:

  [ , ] , ( , ) , { , } , ; , ,

- Literales de tipo entero:
  - En formato decimal: 1234
  - En formato octal: 0o77
  - En formato hexadecimal: 0xFF
- Literales de tipo real:
  - En formato decimal: 12.34
  - En formato científico: 6.023e+23
- Literales de tipo carácter:
  - Carateres imprimibles o de escape: ‘a’ , ‘\n’ , ‘\75’ , ‘\o37’ , ‘\x2A’
- Literales de tipo cadena (String):
  - Lista de caracteres: “hola \n mundo”

- Indentado (o sangrado):
  - La gramática de las sentencias **where**, **let**, **do** y **of** admite un bloque definido entre llaves y separado por punto y coma.

```
size :: Stack a -> Int
size s = length (stkToLst s)
          where {
             stkToLst Empty = []   ;
             stkToLst (MkStack x s) = x:xs where { xs = stkToLst s }
          }
```

- Indentado (o sangrado):
  - Los separadores se pueden eliminar si se cumplen las siguientes carcaterísticas de sangrado (indentation):
  - El siguiente lexema se toma como la posición de la llave abierta omitida.
  - Si la siguiente línea comienza con más espacios se considera la continuación de la línea anterior. Si tiene los mismos espacios se inserta un punto y coma. Si tiene menos espacios se inserta la llave cerrada.

```
size :: Stack a -> Int
size s = length (stkToLst s) where
                stkToLst Empty = []
                stkToLst (MkStack x s) = x:xs
                                                where  xs = stkToLst s
```

- Un ***programa*** escrito en Haskell está formado por módulos. Cada módulo debe estar definido en un fichero con extensión ".hs"
- Un ***módulo*** es un conjunto de declaraciones que pueden ser de diversos tipos: definición de tipos de datos (type, newtype, data), definición de clases (class, instance, default), definición de valores, definición de funciones y definición de operadores.
- La construcción básica de Haskell es la expresión. Una **expresión** permite calcular un valor y tiene un determinado tipo de dato.

- Ejemplo de programa: Test1.hs

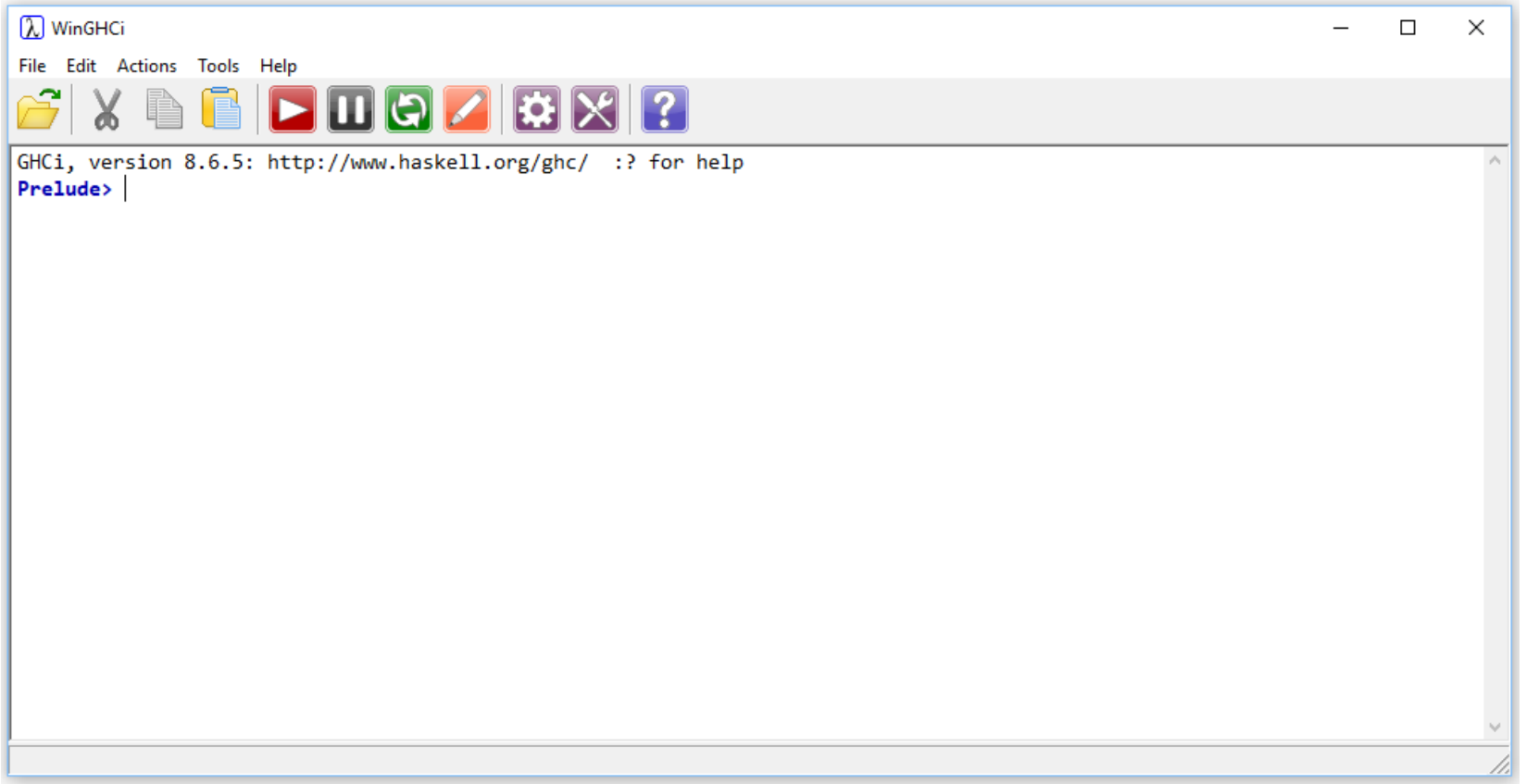
```
{--
    Primer ejemplo de un programa en Haskell
 --}

main = putStrLn "Hello World"
```

- Para compilar el fichero debemos ejecutar el comando

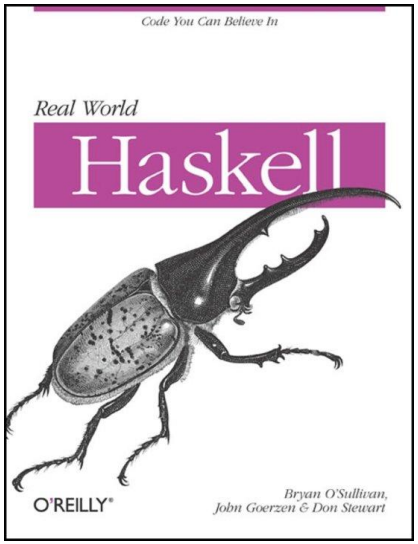
```
> ghc --make Test1.hs -o test1.exe
```

- El intérprete **ghci** permite evaluar expresiones de forma directa



Ejercicios:

- Probar en el intérprete de Haskelll los diferentes ejemplos incluidos en el capítulo “Getting started” del libro “Real World Haskell”




http://book.realworldhaskell.org/read/getting-started.html