

# PRÁCTICA 1

## LENGUAJES FUNCIONALES



**Universidad de Huelva**

**Tutor de prácticas: Antonio Palanco Salguero**

**¿qué hago aquí?**

**Teoría y prácticas INDEPENDIENTES**

**Evaluación independiente con la que se hace media.**

**¿Qué vamos a ver?**

**Paradigmas de la programación**

**Lenguajes Funcionales**

**HASKELL**

Los paradigmas de programación son modelos para resolver problemas comunes con nuestro código.



### Imperativa

El primer paradigma que se suele estudiar es el paradigma imperativo.

Para resolver un problema se deben realizar una serie de pasos y el programador es el encargado de describir de forma **ordenada y sistemática** los pasos que debe seguir el ordenador para obtener la solución.

BASIC	C
Fortran	Pascal
Perl	PHP

### Imperativa: ejemplo de programa – búsqueda método de la burbuja

```
void intercambiar(int *x,int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void burbuja(int lista[], int n){
    int i,j;
    for(i=0;i<(n-1);i++)
        for(j=0;j<(n-(i+1));j++)
            if(lista[j] > lista[j+1])
                intercambiar(&lista[j],&lista[j+1]);
}
```

### Declarativa

Se basa en unidades conceptuales básicas que se pueden combinar según unas determinadas reglas para generar nueva información.

El dominio, será el conjunto de todas esas unidades conceptuales

Hay que “preguntar” a la máquina en base a ese dominio y es la “máquina” la que debe de dar respuesta, si existe, con alguna de las unidades o combinación de ellas

Se divide en dos subparadigmas: programación **funcional (función como unidad lógica)** y programación **lógica (predicado como unidad lógica)**

### **Declarativa: ventajas**

Descripciones compactas y muy expresivas.

Desarrollo del programa no tan orientado a la solución de un único problema.

No hay necesidad de emplear esfuerzo en diseñar un algoritmo que resuelva el problema ya que al escribir el código, no es necesario determinar el procedimiento según el cual se alcanza el resultado (ordenar los pasos).

#### **Ejemplo:**

```
$nombres = array_values($listaparticipanes);
```



### **Funcional: subparadigma de programación declarativa**

Función (no procedimiento) como unidad del dominio o concepto descriptivo básico

**No escribiremos programas, definiremos funciones que describen el problema.**

**Resolución: evaluación de una función basada en las previamente definidas.**



<http://www.haskell.org/haskellwiki/Introduction>

### Funciones puras

¿Qué es una función pura?

*Una función representa una correspondencia entre dos conjuntos que asocia a cada elemento en el primer conjunto un único elemento del segundo*

$$f(x)=a$$

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.

$$a + b = f(x) + b$$

**Funciones puras: no se respeta en lenguajes imperativos**

### PROCEDIMIENTOS VS FUNCIÓN

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

Pueden 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).

### 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



### Funciones puras

Se denomina por tanto ***función pura*** al tipo de función que sigue exactamente las nociones matemáticas y utiliza variables inmutables.

Para trabajar con variables inmutables...

No existen los bucles...



Asignación, salvo en declaración.

No es posible usar variables índices

### Funciones puras

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;
}
```

FACTORIAL LENGUAJE IMPERATIVO

### Funciones puras

En un lenguaje que utiliza 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);  
}
```

FACTORIAL EN LENGUAJE FUNCIONAL PURO

# RECURSIVIDAD



### Funciones de orden superior

En ciencias de la computación las **funciones de orden superior** son funciones que cumplen al menos una de las siguientes condiciones:

- Tomar una o más funciones como entrada: funciones como parámetros
- Devolver una función como salida: función como salida

Para poder utilizar funciones como datos es necesario definir el tipo de dato y se suele utilizar el operador “flecha” y la **declaración de tipo** que permita asociar al identificador a un cierto tipo de dato funcional

**typedef intfun = int -> int**

*funciones con un argumento entero que generan como resultado un valor entero*

### Funciones de orden superior

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

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

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

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 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*).

### Por tanto

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***.

### ESTRICTOS

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

### NO ESTRICTOS

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

### ¿QUÉ ES LA PROGRAMACIÓN FUNCIONAL?

El objetivo en la programación funcional es definir QUÉ CALCULAR, pero no cómo calcularlo.

Haskell es un lenguaje de programación:

- **funcional puro**
- con **tipos polimórficos estáticos**
- con evaluación perezosa (**lazy**)

El nombre lo toma del matemático Haskell Brooks Curry especializado en lógica matemática. Haskell está basado en el **lambda cálculo**.

### Características de Haskell

- Inferencia de tipos. La declaración de tipos es opcional. El compilador puede calcular el tipo a partir de las expresiones a partir de un análisis estático de las definiciones previas y de las variables del cuerpo.
- Evaluación perezosa: sólo se calculan los datos si son requeridos
- Versiones compiladas e interpretadas
- Todo es una expresión
- Las funciones se pueden definir en cualquier lugar, utilizarlas como argumento y devolverlas como resultado de una evaluación.



### Implementaciones de Haskell

Existen diferentes implementaciones de Haskell

- GHC: el más utilizado
- Hugs: Muy utilizado para aprender Haskell. Compilación rápida. Desarrollo rápido de código.
- Nhc98
- Yhc.

Para la realización de las prácticas utilizaremos la implementación Hugs ( <http://haskell.org/hugs/> ).

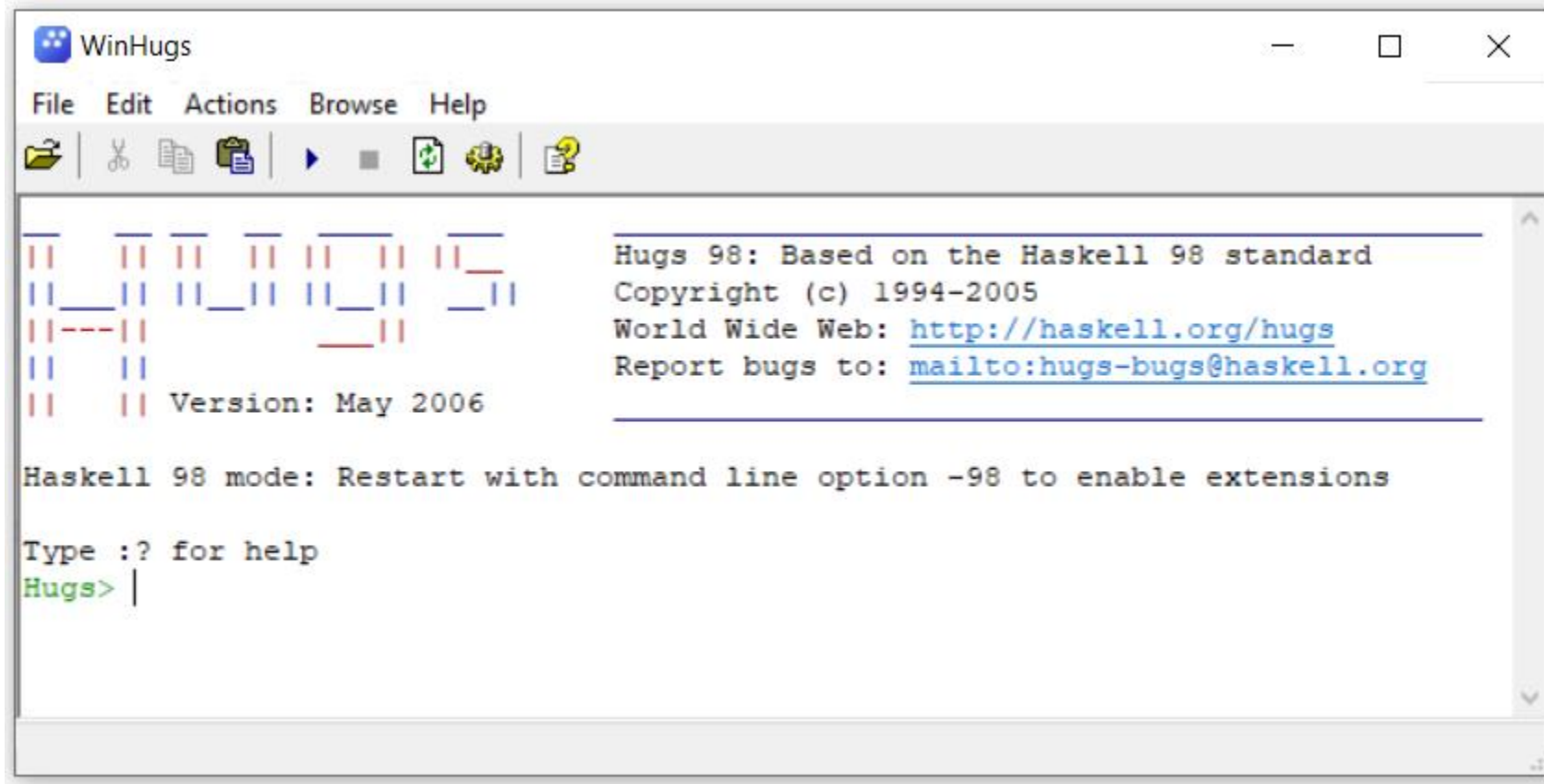
### Implementaciones de Haskell

Existen diferentes implementaciones de Haskell

- GHC: el más utilizado
- Hugs: Muy utilizado para aprender Haskell. Compilación rápida. Desarrollo rápido de código.
- Nhc98
- Yhc.

Para la realización de las prácticas utilizaremos la implementación Hugs ( <http://haskell.org/hugs/> ).

# Implementaciones de Haskell



### Objetivo de las prácticas

- veremos una pequeña introducción a Haskell
- construir pequeñas funciones
- tener una idea del modo de programar en Haskell
- Y una pequeña visión sobre sus posibilidades.
- No veremos la conexión de Haskell con otros programas usando herramientas como **HaskellDirect**
- **Programación final en Scala**

### Scala

Scala es un lenguaje de programación híbrido

- Orientado a objetos
- Programación funcional

Se utiliza para aplicaciones en clústeres u ordenadores multicore.



### **Pero... ¿Por qué Haskell?**

**¿Tendencia para los programas corran más rápidos? ¿Qué pasa con la velocidad de los procesadores?**

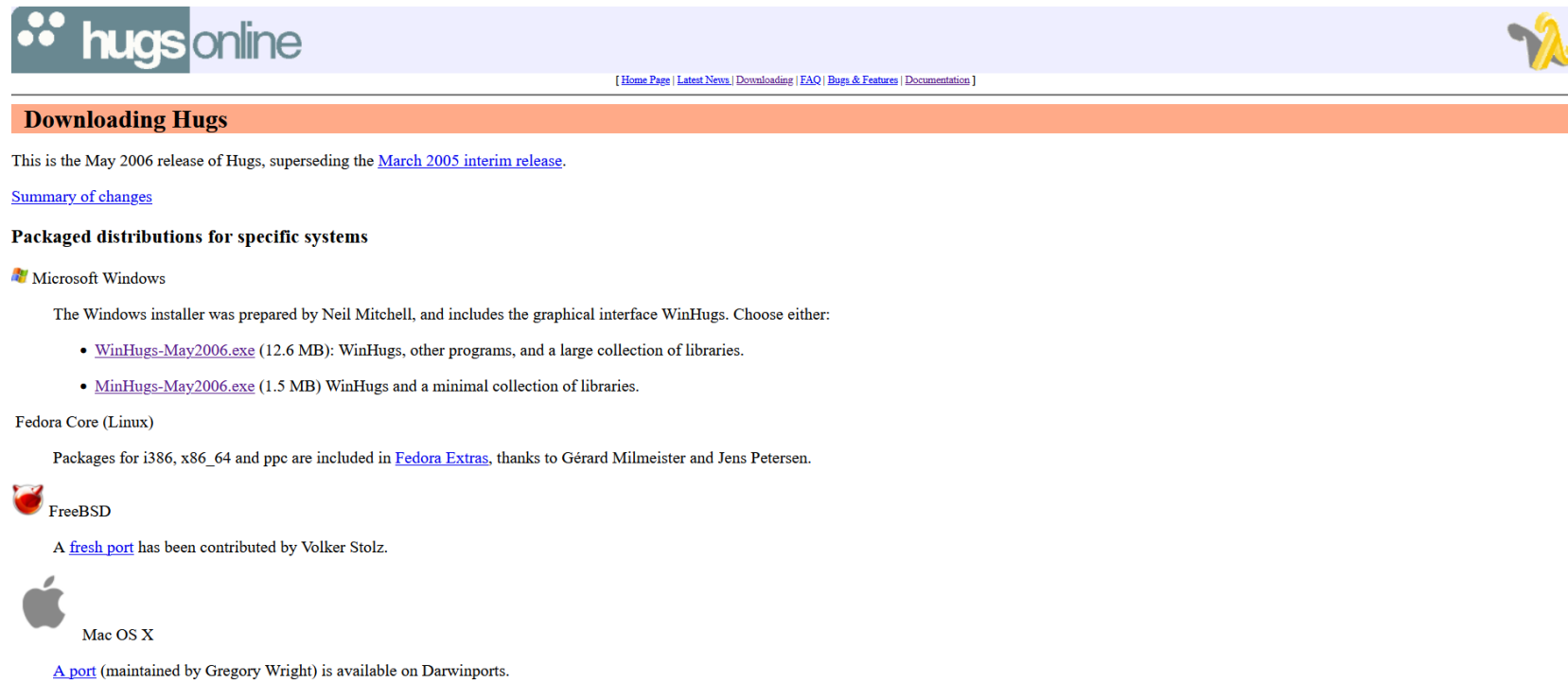
Transistor es un conmutador que necesita un tiempo de computación. Al aumentar la frecuencia de reloj, el transistor conmuta más rápido y llegamos a un límite en el que no se puede aumentar más.

La tendencia es distribuir la computación en diferentes cores. Esto implica diferente forma de programar y que sea sencilla y menos propensa a errores

LIMITE DE PROCESAMIENTO → NECESITAMOS MÁS CORES → LENGUAJE FIABLE → RESTRICCIONES PARA CONSEGUIR ESO

### WinHugs

<https://www.haskell.org/hugs/pages/downloading-May2006.htm>



The screenshot shows the 'hugs online' website header with a navigation bar containing links: [ Home Page | Latest News | Downloading | FAQ | Bugs & Features | Documentation ]. Below the header is an orange banner with the text 'Downloading Hugs'. The main content area states: 'This is the May 2006 release of Hugs, superseding the [March 2005 interim release](#). [Summary of changes](#)'. Under the heading 'Packaged distributions for specific systems', there are three sections: 1. 'Microsoft Windows' with a Windows logo icon, stating the installer was prepared by Neil Mitchell and includes the graphical interface WinHugs. It lists two options: 'WinHugs-May2006.exe (12.6 MB): WinHugs, other programs, and a large collection of libraries.' and 'MinHugs-May2006.exe (1.5 MB) WinHugs and a minimal collection of libraries.' 2. 'Fedora Core (Linux)' with a Linux logo icon, stating packages for i386, x86\_64 and ppc are included in [Fedora Extras](#), thanks to Gérard Milmeister and Jens Petersen. 3. 'FreeBSD' with a FreeBSD logo icon, stating a [fresh port](#) has been contributed by Volker Stolz. 4. 'Mac OS X' with an Apple logo icon, stating a [port](#) (maintained by Gregory Wright) is available on Darwinports.

**Winhugs: carga de ficheros .hs**

```
001-definicion-de-funciones.hs x
1  {- ----- -}
2  -- DECLARACIÓN
3  noNegativo::(Num a, Ord a)=>a->Bool
4  {- PROPÓSITO
5   Devuelve True si x es >= 0, False en otro caso
6   -}
7  -- DEFINICIÓN
8  noNegativo x = x >= 0
9  {-PRUEBAS
10 noNegativo (-2.5) -- devuelve False
11 noNegativo 0-- devuelve True
12 noNegativo 5-- devuelve True
13 -}
14 {- ----- -}
```



### Winhugs: carga de ficheros .hs

Veamos los elementos necesarios para definir una función.

Lo primero que encontramos es un **comentario**.

- Para **comentar un bloque** {- -}
- Para **comentar una línea** --

Después del bloque comentado, encontramos la cabecera de la función.

`<nombre_funcion>::<declaración_de_tipos>`

El **nombre de la función** empieza por una letra minúscula y después puede continuar con mayúscula o minúsculas.

Para **definir los tipos de la función** podemos utilizar variables de tipos pertenecientes a alguna clase (Eq, Ord, Enum, Num, Fractional, etc.) o con tipos básicos (Int ,Integer, Float, Doble, Char, etc.).

### Winhugs: funciones en haskell

`noNegativo::(Num a, Ord a) => a -> Bool`

La función tiene un tipo genérico a **numerable y ordenable**.

A continuación se muestran las secuencias de la instrucción

Observamos que la separación entre variables de entrada y salida no están tan claras.

Ejemplos: `noNegativo 5` / `noNegativo -1` / `noNegativo (-1)`

### Declaración estricta de tipos

HERENCIA

**Nombre\_de\_la\_clase**

tipo1, tipo2, tipo3

### Declaración estricta de tipos

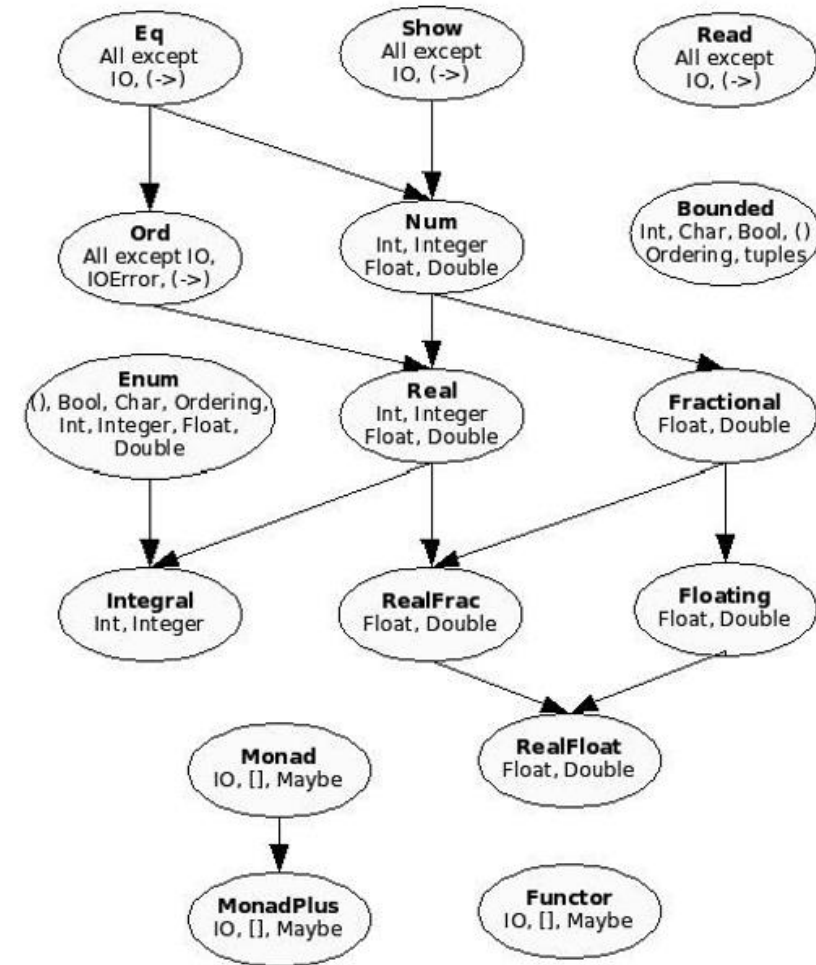
Se hereda de varias clases

(->) denota a una función

IO: entrada / salida

IOError: error de entrada / salida

Clasificación de tipos en Haskell:



### Winhugs: la función suma

Cargar 002-suma.hs

:info suma

¿Qué ocurre?

suma :: Num a => a -> a -> a

```
1  {- ----- -}  
2  -- ejemplo 1 sin cabeceras  
3  |  
4  suma x y = x + y  
5  
6  {- ----- -}
```

La cabecera es inferenciada por el compilador con la clase más genérica posible en base a los argumentos

### Winhugs: la función suma

`suma (mod 5 3) 1`

`suma (mod 5 3)(4 / 2)`

`integral + fractional`

`suma (mod 5 3)(4 div 2)`

`integral + integral`

Existen funciones que permiten convertir tipos: **fromIntegral**

### Winhugs: pruebas

```
iguales::a->a->a->Bool
```

```
iguales x y z = x==y && y==z
```

```
divide::Fractional a => a -> a -> a
```

```
divide x y = x / y
```

```
identidad::a->a
```

```
identidad(Num a)=>a->a->a
```

```
identidad x = x
```