

# PRÁCTICA 2

## INTRODUCCIÓN WINHUGS



Universidad de Huelva

# REPASO DE CONCEPTOS

### ¿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*

- *No existe asignación salvo en la declaración*
- *No existen los bucles*

### 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

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*.

## 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)`

## 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**

## Ejemplos de tipos de datos utilizados por los operadores:

a) Si utilizamos el operador “==” el tipo que utilicemos debe ser comparable (de la clase Eq).

```
Hugs> :info ==
```

```
infix 4 ==
```

```
(==) :: Eq a => a -> a -> Bool -- class member
```

b) Si nuestra función contiene el operador “>”, el tipo debe ser ordenable (de la clase Ord)

```
Hugs> :info <
```

```
infix 4 <
```

```
(<) :: Ord a => a -> a -> Bool -- class member
```

c) Si nuestra función contine el operador “+”, el tipo debe ser numérico

```
Hugs> :info +
```

```
infixl 6 +
```

```
(+) :: Num a => a -> a -> a -- class member
```



### 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
```

### Winhugs: pruebas

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

```
suma x y z = x + y + z
```

```
suma :: Num a => a -> a -> a // suma :: Integer => Integer -> Integer
```

```
suma 1 y z = 1 + y + z
```

¿y si queremos sumar 1.5 1.7 2.0?

# SISTEMA DE TIPADO

### Haskell es un lenguaje **fuertemente tipado**

Toda expresión tiene un tipo de dato concreto y que no puede modificarse de manera implícita.

Haskell no realiza conversiones de tipo implícitas (*casting* o *coercion*). Si queremos transformar el tipo de dato de una expresión es necesario aplicar un operador de conversión de tipos.

El lenguaje C, por ejemplo, realiza transformaciones automáticas entre *int* y *float* o entre *int* y *double*.

### Haskell es un lenguaje **fuertemente tipado**

Esto quiere decir que el tipo de dato de cualquier expresión es conocido en tiempo de compilación

### Haskell utiliza **inferencia de tipos**

Esto quiere decir que en la mayoría de los casos el programador **no está obligado a indicar explícitamente el tipo de dato** de las expresiones sino que el compilador es capaz de inferirla considerando los tipos de datos básicos incluidos y las funciones utilizadas en las expresiones (como hemos visto anteriormente).

En lenguajes como C o Java es necesario declarar el tipo de dato de cada variable a utilizar o el tipo de dato de los argumentos de las funciones.

Haskell permite indicar los tipos de datos (*type signature*), pero por defecto el compilador intentará asignar a las expresiones el tipo de dato más general posible que sea compatible con la expresión

Haskell utiliza **tipos de datos jerárquicos (type classes)**:

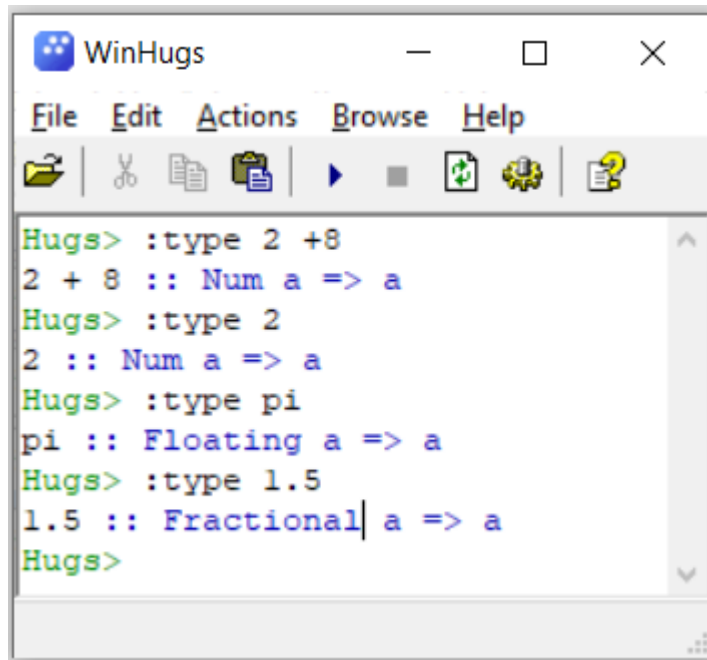
Las *type classes* permiten definir una jerarquía de tipos en la que las subclases contienen las mismas funciones que la clase superior y puede añadir nuevas clases.

Por ejemplo, la clase *Eq* describe cualquier tipo de dato que tenga definido el comportamiento sobre los operadores “==” y “/=”.

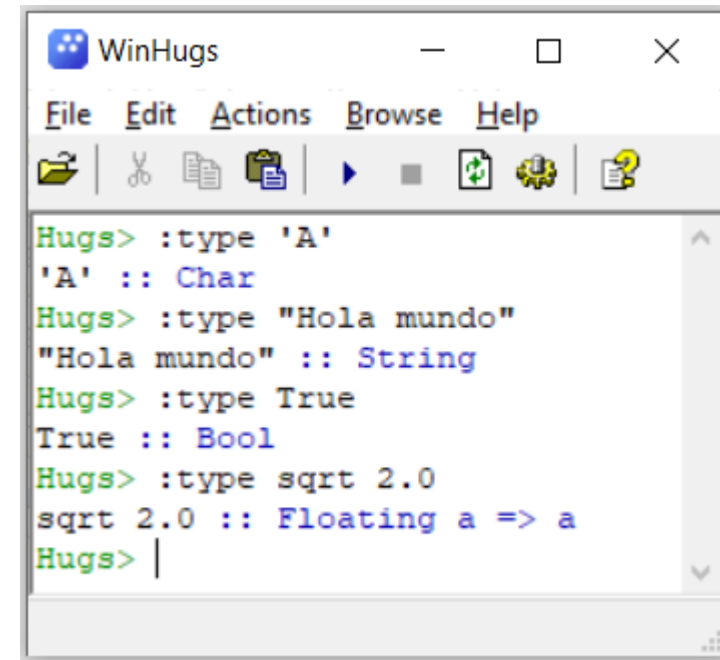
**Eq**  
All except  
IO, (->)

El intérprete de Haskell permite utilizar el comando **:type** para obtener el tipo de dato inferido en una expresión. P

### Ejemplos



```
WinHugs
File Edit Actions Browse Help
[Icons]
Hugs> :type 2 +8
2 + 8 :: Num a => a
Hugs> :type 2
2 :: Num a => a
Hugs> :type pi
pi :: Floating a => a
Hugs> :type 1.5
1.5 :: Fractional a => a
Hugs>
```



```
WinHugs
File Edit Actions Browse Help
[Icons]
Hugs> :type 'A'
'A' :: Char
Hugs> :type "Hola mundo"
"Hola mundo" :: String
Hugs> :type True
True :: Bool
Hugs> :type sqrt 2.0
sqrt 2.0 :: Floating a => a
Hugs> |
```

Haskell utiliza **tipos de datos jerárquicos (type classes)**:

Se puede indicar al intérprete que muestre siempre la información del tipo con el comando **:set +t**

```
Hugs> :set +t
Hugs> 2+8
10
it :: Num a => a
```

Para obtener información sobre un tipo de dato se utiliza el comando **:info**

```
Prelude> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
```



# TIPOS Y CLASES

### Los siguientes tipos están definidos en el módulo Prelude:

#### **Bool:**

Tipo de dato booleano. Admite dos valores: **True** y **False**.

#### **Ordering:**

Tipo de dato que responde al valor de una comparación. Admite los valores **LT**, **EQ** y **GT**.

#### **Char:**

Describe un carácter en formato unicode.

### Los siguientes tipos están definidos en el módulo Prelude:

#### Int:

Describe un tipo de dato entero con el tamaño estandar del procesador (32 o 64 bits).

#### Integer:

Describe un valor entero no acotado. Este tipo de dato permite realizar operaciones con una **precisión ilimitada** aunque las operaciones son computacionalmente más costosas.

#### Float:

Tipo de dato en coma flotante en formato IEEE de 32 bits. Se recomienda utilizar mejor el formato Double.

## Clases estándar contenidas en prelude

### Eq:

Clase que describe tipos de datos sobre los que están definidos los métodos de igualdad ( `==` ) y desigualdad ( `/=` ).

### Ord:

Clase que describe tipos de datos que pueden ordenarse. La clase define los métodos *compare* (que genera un resultado de tipo *Ordering*), *min* (que obtiene el mínimo entre dos valores), *max* (que obtiene el máximo entre dos valores) y los operadores ( `>` ), ( `>=` ), ( `<` ) y ( `<=` ) que devuelven valores de tipo *Bool*.

## Clases estándar contenidas en prelude

### Show:

Clase que describe tipos de datos cuyos valores se pueden mostrar como cadenas.

### Read:

Clase que describe tipos de datos cuyos valores se pueden parsear desde una cadena de caracteres.

### Enum:

Clase que describe tipos de datos que se pueden enumerar. Contiene los métodos *succ* y *pred* que obtiene el sucesor y predecesor de un valor. También contiene los métodos *toEnum* y *fromEnum* (que obtiene el valor a partir del índice entero) y *enumFrom*, *enumFromThen*, *enumFromTo* y *enumFromThenTo* (que construyen listas a partir de valores iniciales y finales).

## Clases estándar contenidas en prelude

### Bounded:

Clase que describe tipos de datos acotados. Contiene los métodos *maxBound* y *minBound* que devuelven los valores máximos y mínimos del tipo.

### Num:

Clase que describe todos los tipos de datos numéricos. Contiene los métodos  $(+)$ ,  $(*)$ ,  $(-)$  y *negate*, que puede escribirse como el prefijo  $(-)$ .

## Clases estándar contenidas en prelude

### Real:

Clase que describe números reales (tanto enteros como en coma flotante). Contiene el método *toRational* que transforma los datos numéricos a formato racional (en forma de fracción de enteros).

### Integral:

Subclase de *Num* y *Real* que describe datos enteros (sin decimales). Contiene los métodos *quot* (división entera truncada hacia el 0), *rem* (resto de la división entera truncada hacia el 0), *div* (división entera truncada hacia menos infinito) y *mod* (resto de la división entera truncada hacia menos infinito).

## Clases estándar contenidas en prelude

### Fractional:

Subclase de *Num* que describe datos en coma flotante. Añade el método ( / ) que describe la división real así como los métodos *recip*, que calcula el inverso o recíproco, y *fromRational* que transforma una fracción a formato en coma flotante.

### RealFrac:

Subclase de *Real* y *Fractional* que añade métodos de redondeo. El método *properFraction* obtiene la parte entera y la parte decimal y los métodos *truncate*, *round*, *ceiling* y *floor* desarrollan diferentes formas de redondeo.



## Clases estándar contenidas en prelude

### Floating:

Subclase de *Fractional* que añade las funciones matemáticas básicas sobre números reales: *pi*, *exp*, *log*, *sqrt*, *\*\**, *logBase*, *sin*, *cos*, *tan*, *asin*, *acos*, *atan*, *sinh*, *cosh*, *tanh*, *asinh*, *acosh*, *atanh*.

### RealFloat:

Subclase que añade métodos de operación a nivel de bit con los datos en coma flotante: *floatRadix*, *floatDigits*, *floatRange*, *decodeFloat*, *encodeFloat*, *exponent*, *significand*, *scaleFloat*, *isNaN*, *isInfinite*, *isDenormalized*, *isNegativeZero*, *isIEEE*.

## Clases estándar contenidas en prelude

### Monad:

Clase que encapsula una forma de combinar cálculos. Una mónada necesita un constructor, un función de combinación (*bind* o *>>=* ) y una función *return*. **La mónada *IO* se utiliza para definir operaciones de entrada y salida.** Las mónadas se usan también como forma de encapsular funciones impuras en Haskell.

### Functor:

Clase que describe un tipo que puede ser mapeado.

# FUNCIONES BÁSICAS

### Operaciones lógicas

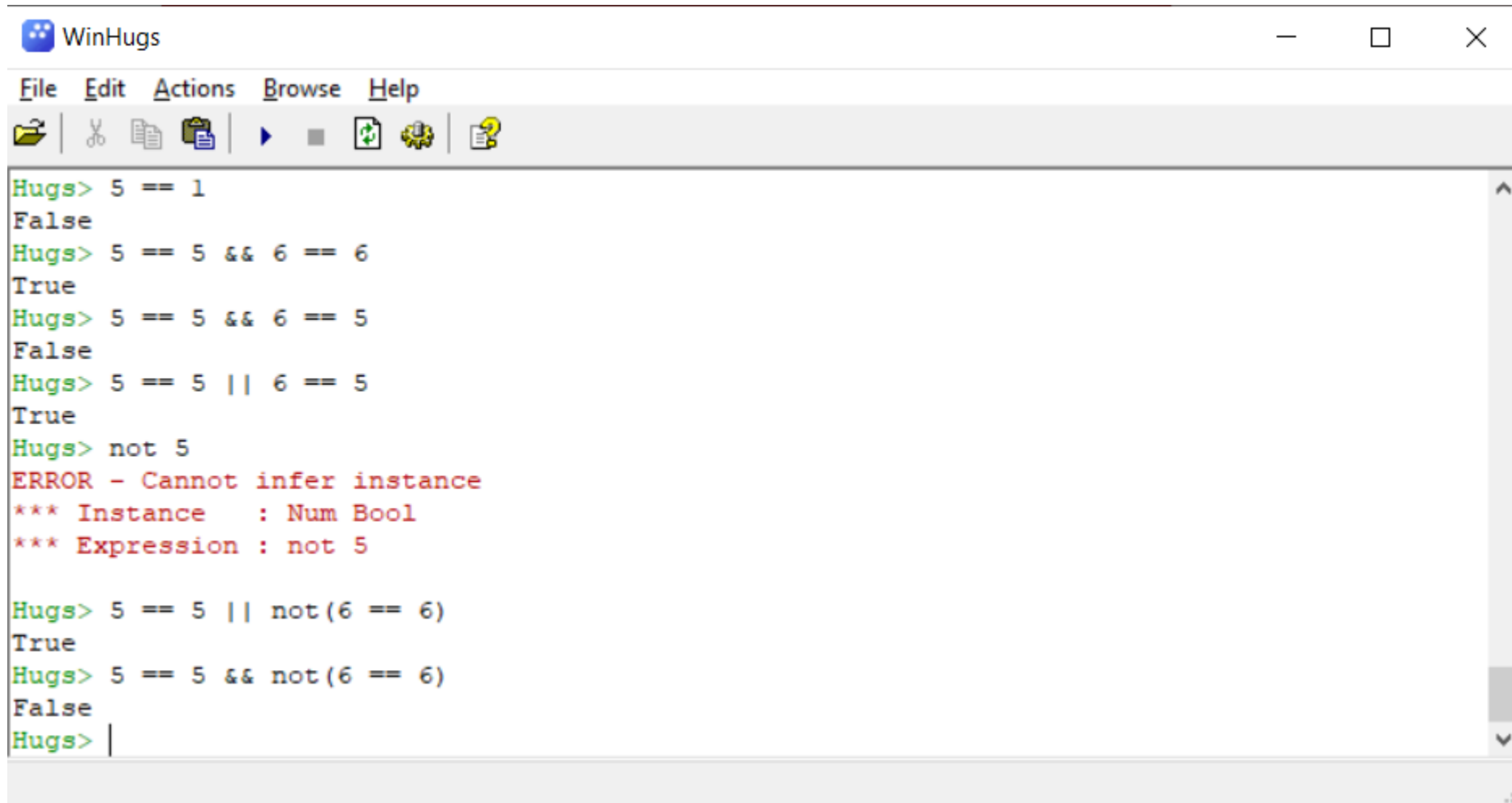
`( && ) :: Bool -> Bool -> Bool` -- AND con cortocircuito

`( || ) :: Bool -> Bool -> Bool` -- OR con cortocircuito

`not :: Bool -> Bool` -- NOT

### **Bounded**

Int, Char, Bool,  
( ), Ordering,  
Tuples

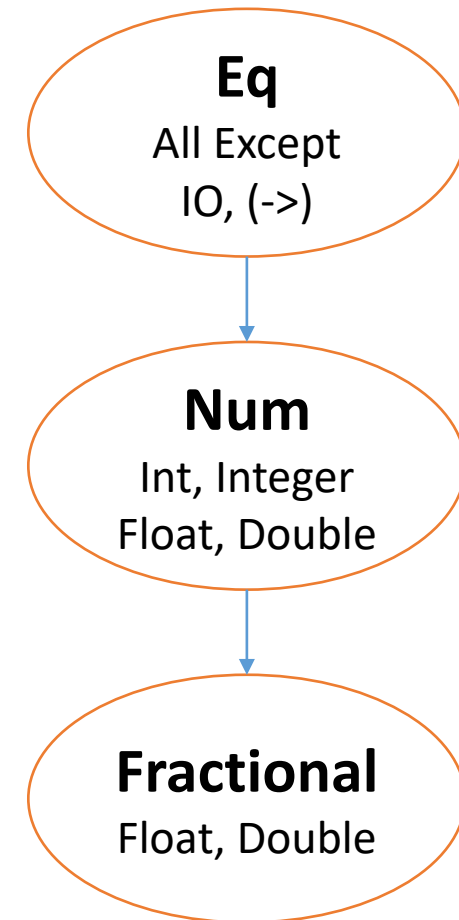


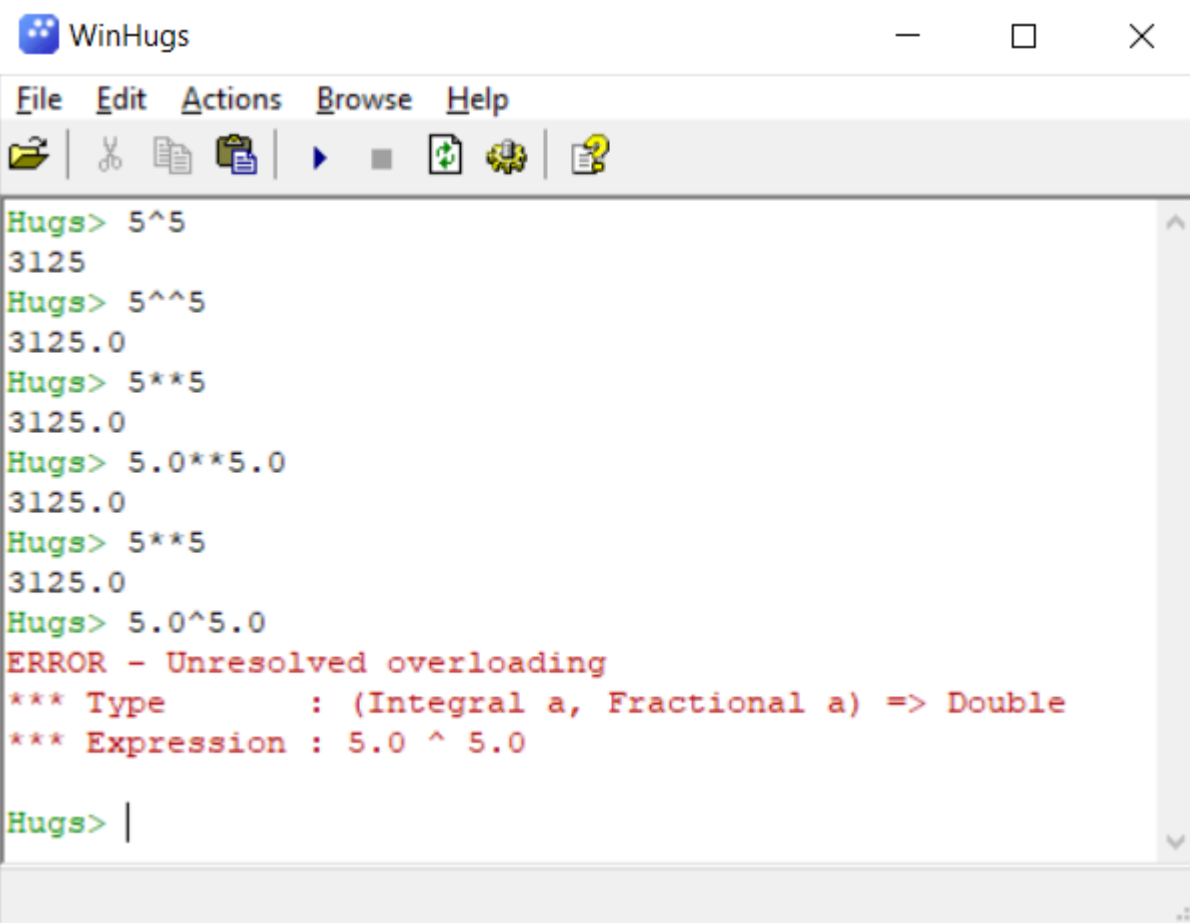
```
WinHugs
File Edit Actions Browse Help
[Icons]
Hugs> 5 == 1
False
Hugs> 5 == 5 && 6 == 6
True
Hugs> 5 == 5 && 6 == 5
False
Hugs> 5 == 5 || 6 == 5
True
Hugs> not 5
ERROR - Cannot infer instance
*** Instance   : Num Bool
*** Expression : not 5

Hugs> 5 == 5 || not(6 == 6)
True
Hugs> 5 == 5 && not(6 == 6)
False
Hugs> |
```

## Operaciones aritméticas

- ( + ) -- Suma definida sobre tipos **Num**
- ( - ) -- Resta definida sobre tipos **Num**
- ( \* ) -- Producto definido sobre tipos **Num**
- ( / ) -- División exacta sobre tipos **Num** generando **Fractional**
- ( ^ ) -- Potencia de un **Num** a un exponente entero
- ( ^^ ) -- Potencia de un **Fractional** a un exponente entero
- ( \*\* ) -- Potencia de un **Fractional** a un exponente **Fractional**





The screenshot shows the WinHugs window with the following content:

```
WinHugs
File Edit Actions Browse Help
[Icons]
Hugs> 5^5
3125
Hugs> 5^^5
3125.0
Hugs> 5**5
3125.0
Hugs> 5.0**5.0
3125.0
Hugs> 5**5
3125.0
Hugs> 5.0^5.0
ERROR - Unresolved overloading
*** Type      : (Integral a, Fractional a) => Double
*** Expression : 5.0 ^ 5.0
Hugs> |
```

Las funciones binarias pueden escribirse en formato infijo si se escribe el nombre entre comillas `.

```
Hugs> quot 25 3
```

```
8
```

```
Hugs> 25 `quot` 3
```

```
8
```

Funciones de división entera:

quot -- División entera con redondeo en 0;    quot (-25) 3 == (-8)

rem -- Resto de división entera con quot;    rem (-25) 3 == (-1)

div -- División entera con redondeo en  $-\infty$ ;    div (-25) 3 == (-9)

mod -- Resto de división entera con div;    mod (-25) 3 == 2



### Otras funciones aritméticas también incluidas en Prelude

**odd** -- Verifica si un número entero es impar

**even** -- Verifica si un número entero es par

**gcd** -- Calcula el máximo común divisor entre dos enteros

**lcm** -- Calcula el mínimo común múltiplo entre dos enteros

**subtract** -- Calcula la resta inversa  $b - a$ .

**abs** -- Calcula el valor absoluto

**signum** -- Calcula el signo de un valor numérico (-1, 0, 1)

### **Pausa y probar funciones con Hugs**

