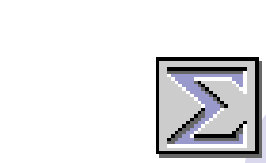
Programación Funcional

Curso 2020-21

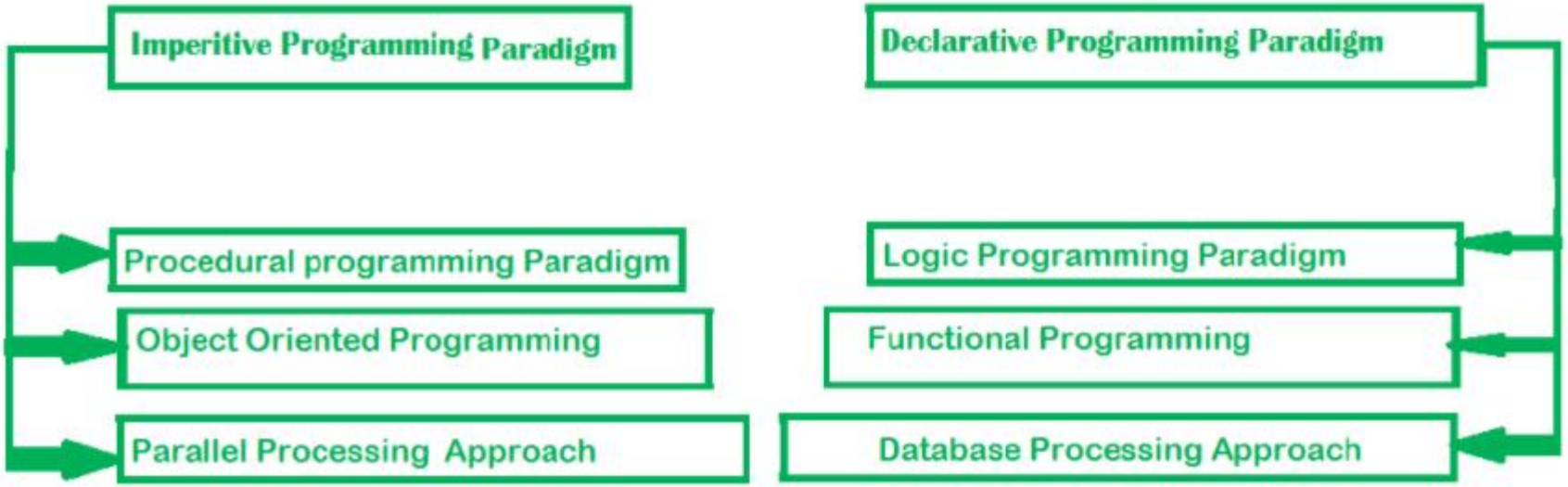
Tema 1.- Introducción

Paradigmas de Programación

- Paradigma = método para resolver algún problema o realizar alguna tarea.
- Paradigma de programación =
  - enfoque para resolver problemas usando algún lenguaje de programación o
  - método para resolver un problema de programación siguiendo algún enfoque.
- Hay muchos lenguajes de programación, pero todos siguen alguna estrategia cuando se diseñan e implementan.
- Ese mecanismo o estrategia es lo que diferencia a los distintos paradigmas y se usa para clasificar los lenguajes de programación.



Programming Paradigms



https://www.geeksforgeeks.org/introduction-of-programming-paradigms/

La programación declarativa expresa la lógica de la computación sin describir el flujo de control de la computación.

Programación Funcional

- Paradigma de *programación declarativa* cuyo mecanismo de computación es la evaluación de funciones matemáticas, evitando los estados y datos mutables.

- Énfasis en la aplicación de funciones, en contraste con el estilo de programación imperativo que pone el énfasis en los cambios de estado.

Paradigma	Mecanismo
imperativo	ejecución de comandos
funcional (declarativo)	evaluación de expresiones
lógico (declarativo)	demostración/refutación

- Son lenguajes funcionales Lisp, FP, ISWIN, ML, Miranda, Haskell, …

La programación funcional se caracteriza por:

- programar consiste en *definir funciones* por aplicación y composición de otras más simples

```
doble x = x + x
cuadruple = doble . doble
```

- el método de computación consiste en *evaluar expresiones* que resultan de aplicar funciones (de usuario o pre-definidas) hasta obtener su *valor*

```
doble 3
 → 3 + 3     [def. de doble]
 → 6         [def. de +]
```

Ventajas de la Programación Funcional

- Programas más cortos (de 2 a 10 veces), claros y concisos:

```
quicksort [] = []
quicksort (prim:resto)
    = quicksort [y | y<-resto, y<prim] ++
      [prim] ++
      quicksort [y | y<-resto, y>=prim]
```

- Lenguajes (e.g. Haskell) y sistemas más fáciles para iniciarse y experimentar

- Menos errores, mayor fiabilidad y mejor mantenimiento
  - *Incremento de la productividad del programador (entre 9% y 25% según Ericsson)*
  - *Todo programa (significativo) acabado se modifica muchas veces*

Principales características de los lenguajes funcionales modernos

- tipado fuerte ➔ detección de errores en compilación

  `quicksort (3,1,2)` produce un error de tipos

  `quicksort [3,1,2]` obtiene `[1,2,3]`

- polimorfismo ➔ aumenta la reusabilidad

  `quicksort` puede usarse con listas de elementos de cualquier tipo (que cuente con una operación <)

- evalución perezosa ➔ modularidad, facilidad de escritura

```
pares = [2*x | x <- [1..]]
take 5 pares
```

Principales características … (cont.)

- funciones de orden superior ➔ alto nivel de abstracción
  - *funciones como "ciudadanos de primera clase"*
  - *aplicación/composición de funciones ➔ modularidad*

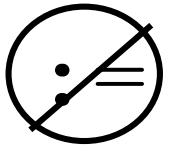
```
cuadruple = doble . doble
```

- gestión de memoria ➔ libera de carga al programador
  inicialización, almacenamiento y liberación automática

- Un programa es una serie de definiciones y una ejecución es la evaluación de una expresión que usa las entidades definidas en el programa.

Principales características … (cont.)

- Los programas funcionales se basan en dar definiciones usando el símbolo de igualdad (=), que nada tiene que ver con la asignación.



  - En programación funcional, el concepto de posición de memoria a la que se le asigna un valor no existe

  - Si en un programa aparece **`x = <expresión>`**

`x` es, por definición, igual a **`<expresión>`**

  - Por tanto, en ese mismo programa no puede aparecer otra vez

```
x = ...
```

de lo contrario no sería correcto.

El Lenguaje Haskell

- Haskell es el lenguaje funcional más estándar (hoy en día).

- Su nombre se debe a Haskell Brooks Curry (1900-1982) cuyo trabajo en lógica matemática ha servido de amplio fundamento a la programación funcional

Trabajó en el primer qsortdor electrónico llamado **ENIAC** (Electronic Numerical Integrator and Computer) después de la Segunda Guerra Mundial.

En 1986 la Mitre Corporation creó el Curry Chip, un innovador elemento hardware basado en el concepto matématico de combinador que Curry definió en los 50's, exactamente del mismo modo que estan basadas las implementaciones de lenguajes funcionales (p.e. Miranda y Haskell).

Un poco de historia

Hashell fue ideado a finales de los años 80 por un comité (para paliar la masiva proliferacion de lenguajes funcionales)

1930s: Alonzo Church desarrolla el λ-cálculo (teoría básica de funciones)

1950s: John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).

1960s: Peter Landin desarrolla ISWIN (lenguaje funcional puro).

1970s: John Backus desarrolla FP (+ orden superior).

1970s: Robin Milner desarrolla ML (+ tipos polimórficos e inferencia de tipos).

1980s: David Turner desarrolla Miranda (+ perezoso).

1987: Un comité comienza el desarrollo de Haskell (+clases de tipos, etc)

2003: El comité publica el Haskell Report

In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This document describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.

The committee's primary goal was to design a language that satisfied these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

- Haskell está basado (semántica e implementación) en el λ-cálculo

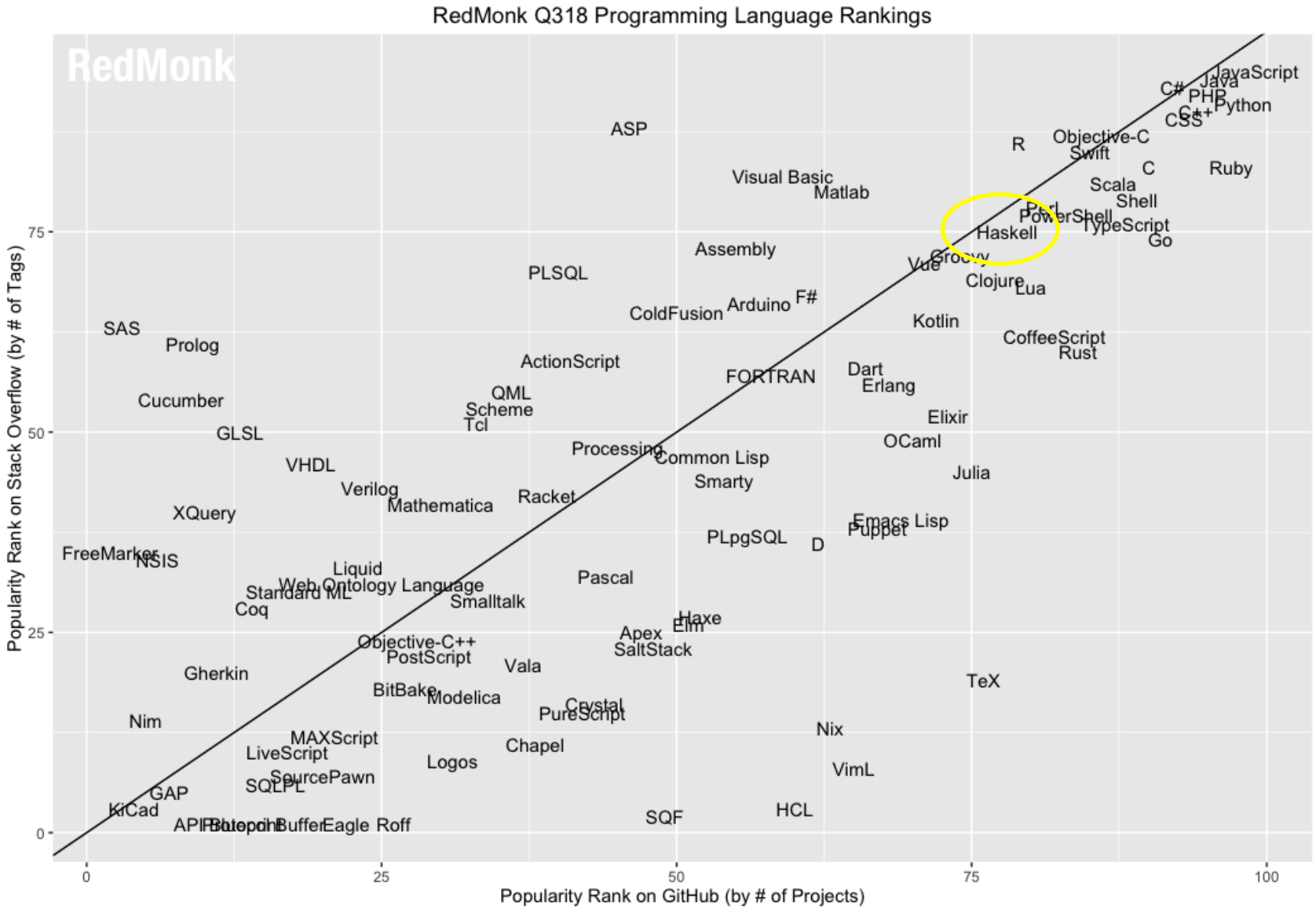
- Haskell posee todas las características antes comentadas y algunas otras como
  - clases de tipos
  - facilidades de módulos
  - I/O puramente funcional

- Haskell se usa en muchas universidades como primer lenguaje de programación

- Haskell es ampliamente usado en la industria en muy diferentes ámbitos
  https://wiki.haskell.org/Haskell_in_industry
  "desde la industria aeroespacial y de defensa, pasando por las finanzas, hasta las web startups, las empresas de diseño de hardware y un fabricante de cortadoras de césped."
  e.g. Facebook, Google, Microsoft, NVIDIA, Alcatel-Lucent.

The RedMonk Programming Language Rankings: June 2020

https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/



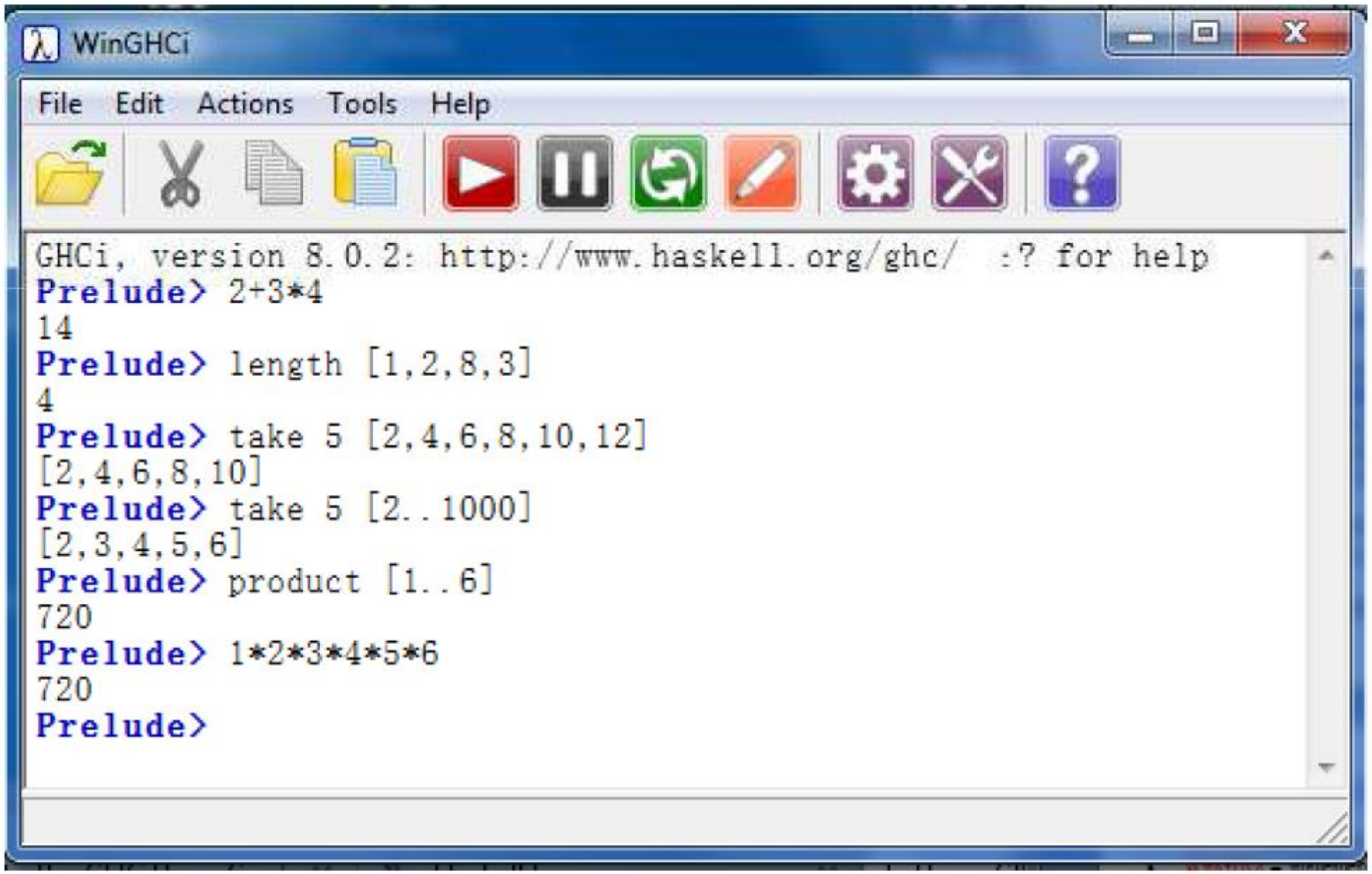
El Sistema GHC

- GHC (Glasgow Haskell Compiler) es el interprete de Haskell que usaremos en el curso.
  - GHCi = entorno interactivo de Glasgow Haskell Compiler
  - WinGHCi = implementación de Haskell que usaremos en el laboratorio.

- Programar en Haskell consiste en definir funciones en un *"script"*:
  - *"script"* = guión, manuscrito
  - fichero que contiene definiciones, declaraciones y comentarios
    (= programa)
  - <nombre>.hs (haskell script)

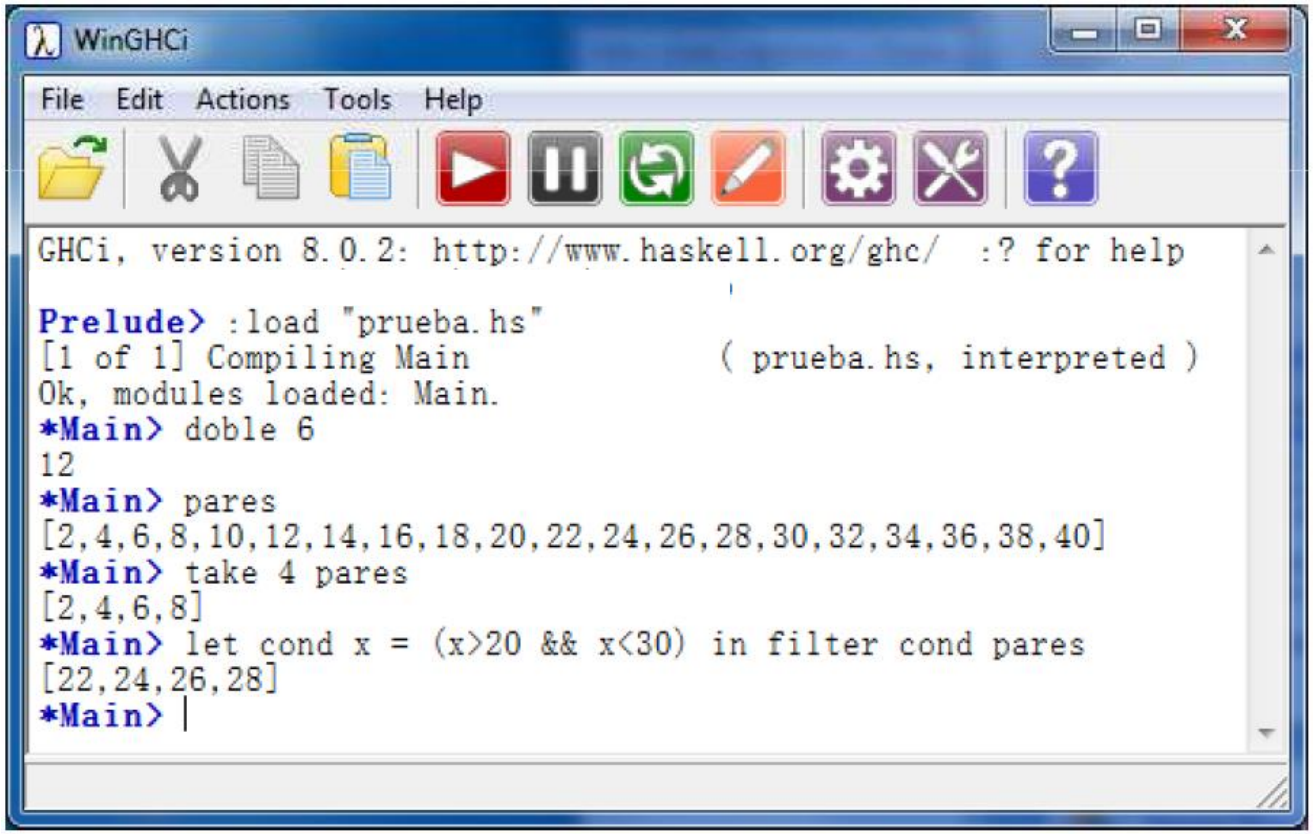
- Almacenando (o cargando) el *"script"*, las funciones definidas pueden ser usadas en la sesión. Además GHC conoce (por defecto) una amplia colección de funciones y operadores, que están contenidas en Prelude.hs

- Ejecutar en WinGHCi es evaluar una expresión en base a los programas cargados en la sesión (Si no se carga ninguno usa Prelude.hs).



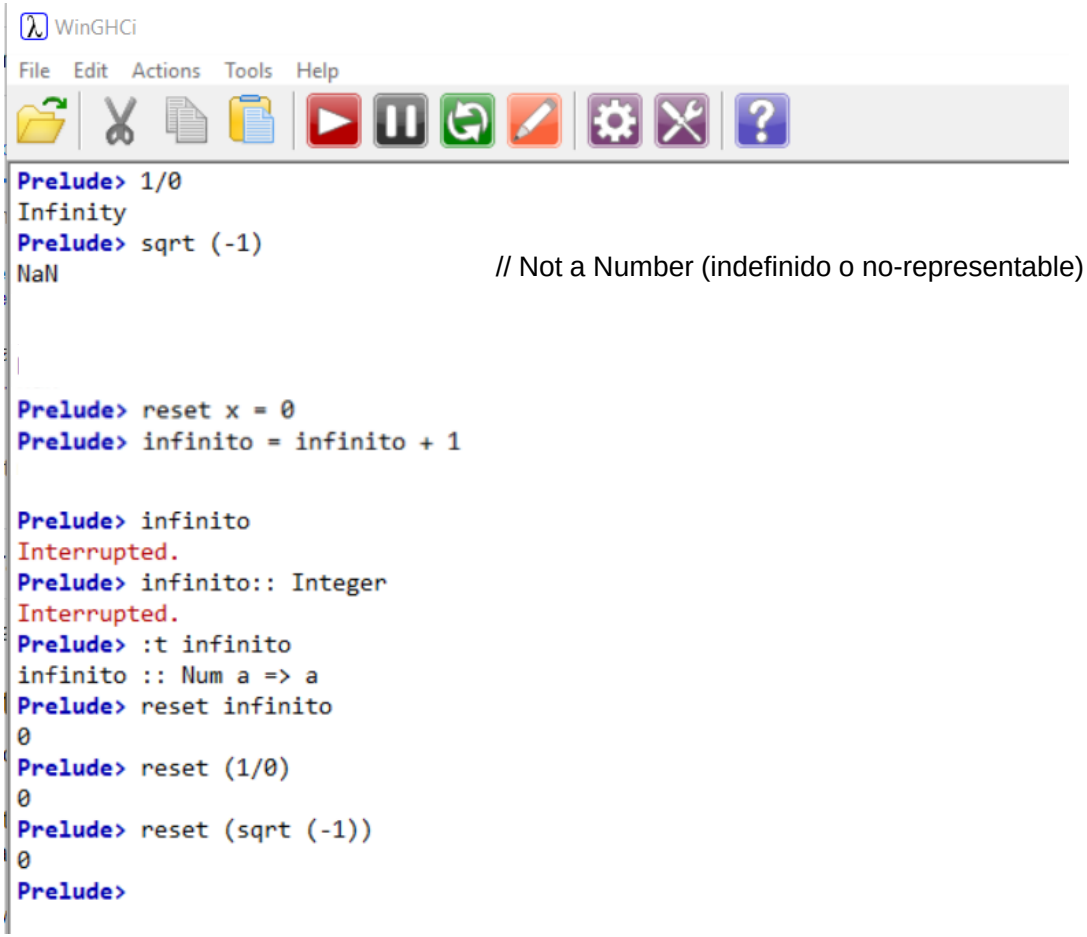
- Si el script `prueba.hs` consiste en las definiciones

```
doble x = x + x
pares = [doble x | x ←[1..20]]
```



Expresiones indefinidas

- No toda expresión tiene un valor (aunque sea correcta y bien tipada):



Contenido de un “script”

- Definiciones:

```
doble x = x + x
```

  es un definición (escrita en Haskell) de la función `doble`

- Declaraciones:

```
doble :: Int -> Int
```

  es un declaración de tipo en Haskell

- Comentarios cortos:

```
-- doble aplicada a un entero devuelve
-- ese entero multiplicado por 2
```

  empiezan con `--` y terminan con “salto de línea”

- Comentarios largos:
  - empiezan con `{-` y terminan con `-}`
  - pueden anidarse
  - facilita el comentar fragmentos de código que incluye comentarios.

Evaluación de expresiones

```
doble x = x + x
```

```
doble 5 → 5 + 5
        → 10
```

```
doble x = x * 2
```

```
doble 5 → 5 *2
        → 10
```

```
cuadruple x = doble (doble x)
```

```
cuadruple 5
      → doble (doble 5)
      → (doble 5) + (doble 5)
      → 10 + 10
      → 20
```

```
cuadruple = doble.doble
```

```
cuadruple 5
      → (doble.doble) 5
      → doble (doble 5)
      → (doble 5) * 2
      → 10 * 2
      → 20
```

Evaluación de expresiones

```
qsort [] = []
qsort (x:xs) = qsort izqd ++ [x] ++ qsort drch
               where  izqd = filter (<x) xs
                      drch = filter (>=x) xs
```

```
qsort[4,6,2,3]
= (qsort [2,3]) ++ [4] ++ (qsort [6])
= ((qsort []) ++ [2] ++ (qsort [3])) ++ [4] ++ (qsort [6])
= ([] ++ [2] ++ (qsort [3])) ++ [4] ++ (qsort [6])
= ([2] ++ (qsort [3])) ++ [4] ++ (qsort [6,5])
= ([2] ++ ((qsort []) ++ [3] ++ [])) ++ [4] ++ (qsort [6])
= ([2] ++ ([] ++ [3] ++ [])) ++ [4] ++ (qsort [6])
= ([2] ++ [3]) ++ [4] ++ (qsort [6])
= [2,3] ++ [4] ++ (qsort [6])
= [2,3,4] ++ (qsort [6])
= [2,3,4] ++ ((qsort []) ++ [6] ++ (qsort []))
= [2,3,4] ++ ((qsort []) ++ [6] ++ (qsort []))
= [2,3,4] ++ ([] ++ [6] ++ [])
= [2,3,4,6]
```

La evolución del programador de Haskell

- Haskellero iterativo *(Programador habitual de Pascal/Ada)*

```
fac n = result (for init next done)
            where init = (0,1)
                  next (i,m) = (i+1, m * (i+1))
                  done (i,_) = i==n
                  result (_,m) = m
for i n d = until d n i
```

- Haskellero iterativo *(Programador habitual de C/APL)*

```
fac n = snd(until ((>n). fst)(\(i,m)->(i+1, i*m))(1,1))
```


- Haskellero junior

```
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

- Haskellero senior

```
fac n = foldr (*) 1 [1..n]
```

- Haskellero fundamentalista

```
fac = product . enumFromTo 1
```

http://www.willamette.edu/~fruehr/haskell/evolution.html