SESION 1

LENGUAJES FUNCIONALES

Universidad de Huelva

Bienvenida

Tutor de prácticas: Antonio Palanco Salguero

¿qué hago aquí?

Teoría y prácticas INDEPENDIENTES: asistencia no obligatoria

No asiste: irá a examen completo

Evaluación independiente con la que se hace media.

Bienvenida

La parte práctica (50% de la nota final) se evalúa por medio de:

- un examen de prácticas (20% de la nota final) [POR DECIDIR]
- y de un trabajo individual (80% de la nota final). [POR DECIDIR]

Bienvenida

Calendarización

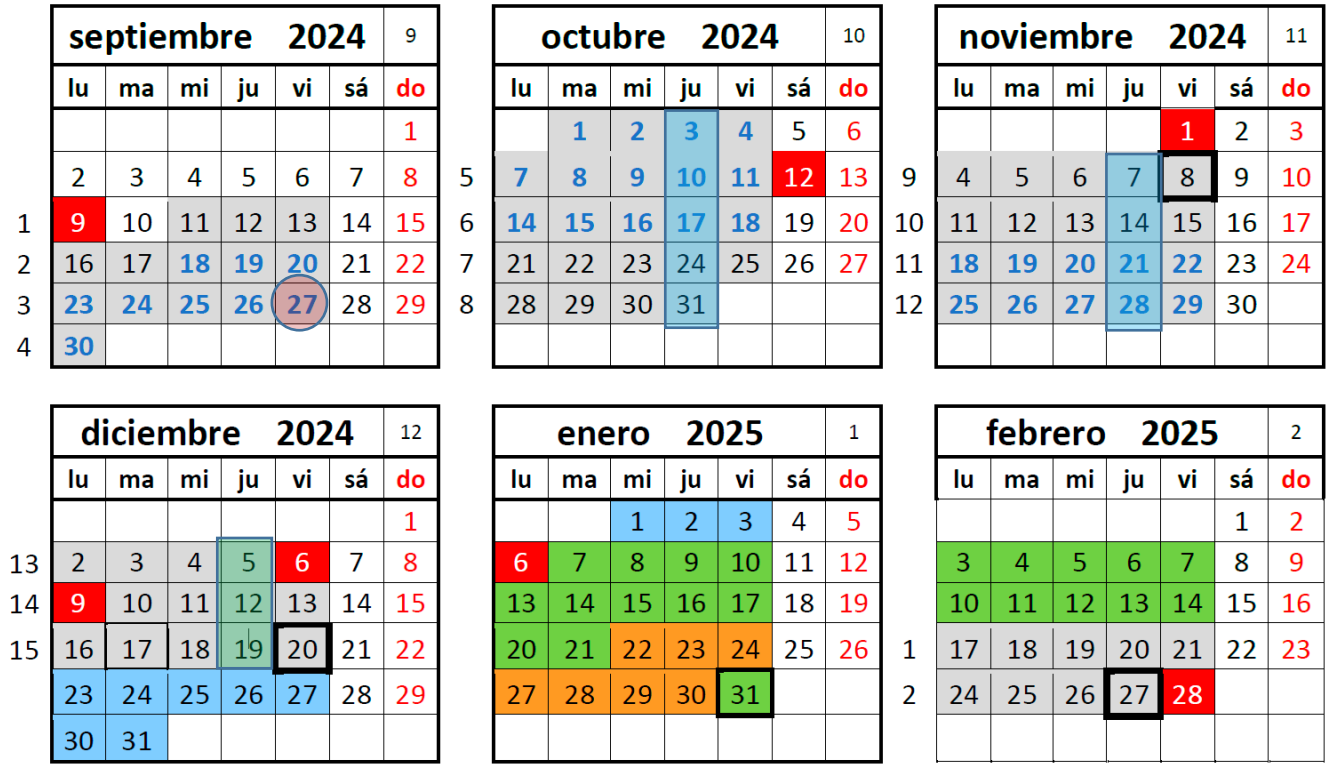
13 sesiones

1: introducción

8-9: haskell, ghc, scala, etc..

2-3: trabajo autónomo

1 o 2: Presentación proyectos



	septiembre 2024						9
	lu	ma	mi	ju	vi	sá	do
							1
	2	3	4	5	6	7	8
1	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22
3	23	24	25	26	27	28	29
4	30						

	octubre 2024						10
	lu	ma	mi	ju	vi	sá	do
		1	2	3	4	5	6
5	7	8	9	10	11	12	13
6	14	15	16	17	18	19	20
7	21	22	23	24	25	26	27
8	28	29	30	31			

	noviembre 2024						11
	lu	ma	mi	ju	vi	sá	do
					1	2	3
9	4	5	6	7	8	9	10
10	11	12	13	14	15	16	17
11	18	19	20	21	22	23	24
12	25	26	27	28	29	30	

	diciembre 2024						12
	lu	ma	mi	ju	vi	sá	do
							1
13	2	3	4	5	6	7	8
14	9	10	11	12	13	14	15
15	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
	30	31					

	enero 2025						1
	lu	ma	mi	ju	vi	sá	do
			1	2	3	4	5
	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
	27	28	29	30	31		

	febrero 2025						2
	lu	ma	mi	ju	vi	sá	do
						1	2
	3	4	5	6	7	8	9
	10	11	12	13	14	15	16
1	17	18	19	20	21	22	23
2	24	25	26	27	28		

¿Qué vamos a ver?

Paradigmas de la programación

Lenguajes Funcionales

HASKELL (breve introducción)

1.1. Paradigmas de programación

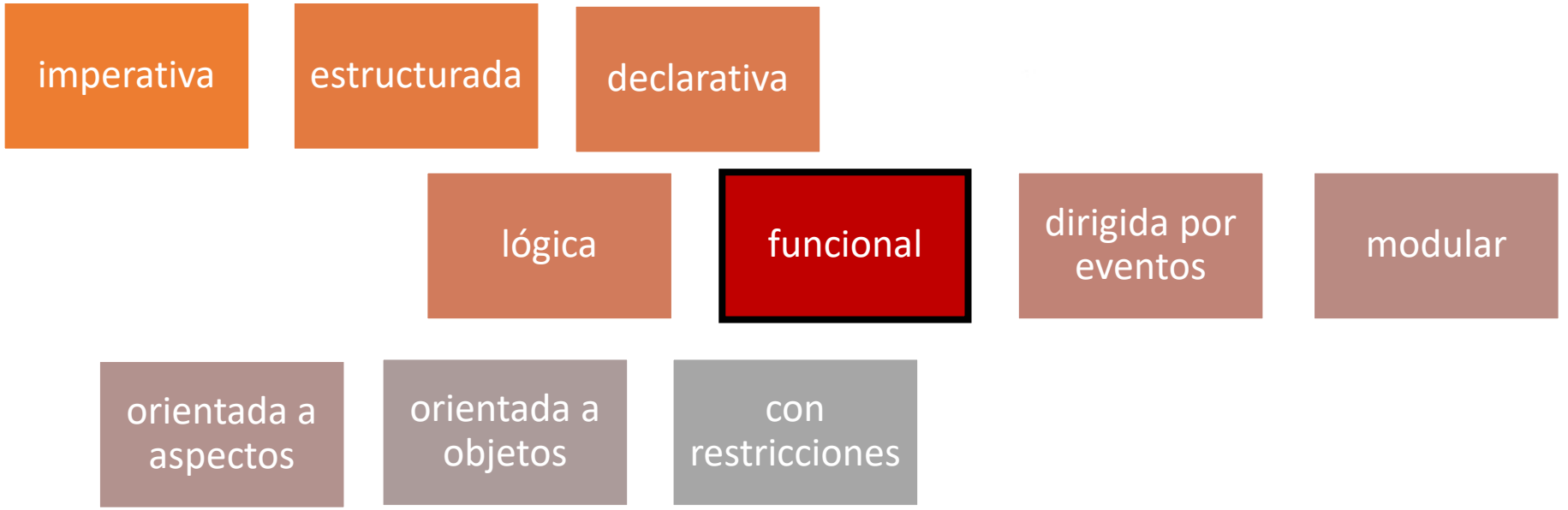
Paradigma de programación: ¿qué es?

Es un conjunto de principios y reglas a seguir para escribir el código de un programa que resuelva un problema (concreto o no concreto). ¿#?

Existen varios tipos de paradigmas, y cada paradigma tiene sus propias características y enfoques específicos para resolver problemas.

1.1. Paradigmas de programación

Los paradigmas: modelos para resolver problemas comunes. ¿me dices alguno?



IMPERATIVA

Imperativa

El primer paradigma que se suele estudiar es el paradigma imperativo. ¿Cómo resuelve un problema?

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	Java	Cobol	Python	…

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.

El código es, por un lado, fácilmente comprensible, pero, por el otro, requiere **muchas líneas de texto fuente.**

Imperativa: cálculo numero primo

```
10 PRINT "Ingrese un número: "
20 INPUT N
30 IF N <= 1 THEN GOTO 80
40 LET I = 2
50 IF N MOD I = 0 THEN GOTO 90
60 LET I = I + 1
70 IF I <= N / 2 THEN GOTO 50
80 PRINT N; " es primo"
85 GOTO 100
90 PRINT N; " no es primo"
100 END
```

BASIC

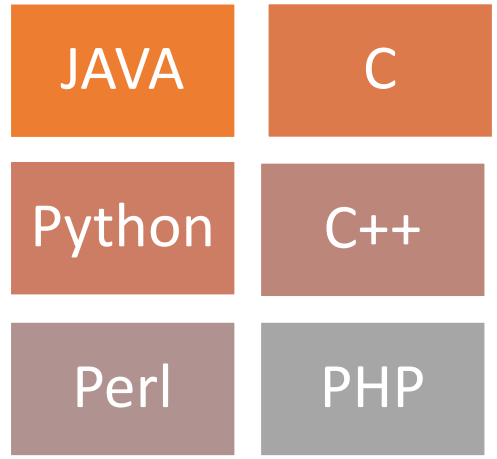
ESTRUCTURADA

Estructurada

Surge para mejorar los “códigos espagueti”.

Se añaden 3 estructura básicas: ¿?

- **Secuencia**: ejecución en el orden de aparición.
- **Selección o condición**: se ejecuta sentencia en función del valor de una variable.
- **Iteración**: ejecuta 1 o un conjunto de sentencias cuando una variable booleana sea verdadera



Estructurada: cálculo numero primo

```
10 INPUT "Ingrese un número: ", N
20 PRIME = 1
30 FOR I = 2 TO N - 1
40 IF N MOD I = 0 THEN PRIME = 0
50 NEXT I
60 IF PRIME = 1 THEN PRINT N; " es primo"
70 IF PRIME = 0 THEN PRINT N; " no es primo"
```

BASIC

Estructurada: cálculo numero primo

```
PROGRAM prime_number
  INTEGER :: n, i, flag
  flag = 1
  PRINT *, 'Ingrese un número:'
  READ *, n

  IF (n <= 1) THEN
     flag = 0
  END IF

  DO i = 2, n / 2
     IF (MOD(n, i) == 0) THEN
        flag = 0
        EXIT
     END IF
  END DO

  IF (flag == 1) THEN
     PRINT *, n, ' es primo.'
  ELSE
     PRINT *, n, ' no es primo.'
  END IF
END PROGRAM prime_number
```

Fortran

```
program PrimeNumber;
var
  n, i: integer;
  isPrime: boolean;
begin
  write('Ingrese un número: ');
  readln(n);
  isPrime := true;

  if n <= 1 then
    isPrime := false;

  for i := 2 to n div 2 do
  begin
    if n mod i = 0 then
    begin
      isPrime := false;
      break;
    end;
  end;

  if isPrime then
    writeln(n, ' es primo.')
  else
    writeln(n, ' no es primo.');
end.
```

Pascal

Estructurada: cálculo numero primo

C

```c
#include <stdio.h>

int main() {
    int n, i, flag = 1;
    printf("Ingrese un número: ");
    scanf("%d", &n);

    if (n <= 1) flag = 0;
    for (i = 2; i <= n / 2; ++i) {
        if (n % i == 0) {
            flag = 0;
            break;
        }
    }

    if (flag == 1)
        printf("%d es primo.\n", n);
    else
        printf("%d no es primo.\n", n);

    return 0;
}
```

C#

```csharp
using System;

class Program {
    static void Main() {
        Console.Write("Ingrese un número: ");
        int n = int.Parse(Console.ReadLine());
        bool isPrime = true;

        if (n <= 1) isPrime = false;

        for (int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }

        if (isPrime)
            Console.WriteLine($"{n} es primo.");
        else
            Console.WriteLine($"{n} no es primo.");
    }
}
```

Estructurada: cálculo numero primo

```perl
print "Ingrese un número: ";
my $n = <STDIN>;
chomp($n);
my $is_prime = 1;

if ($n <= 1) {
    $is_prime = 0;
}

for (my $i = 2; $i <= $n / 2; $i++) {
    if ($n % $i == 0) {
        $is_prime = 0;
        last;
    }
}

if ($is_prime) {
    print "$n es primo.\n";
} else {
    print "$n no es primo.\n";
}
```

Perl

```php
<?php
echo "Ingrese un número: ";
$n = intval(fgets(STDIN));
$is_prime = true;

if ($n <= 1) {
    $is_prime = false;
}

for ($i = 2; $i <= $n / 2; $i++) {
    if ($n % $i == 0) {
        $is_prime = false;
        break;
    }
}

if ($is_prime) {
    echo "$n es primo.\n";
} else {
    echo "$n no es primo.\n";
}
?>
```

PHP

Estructurada: cálculo numero primo

Java

```java
import java.util.Scanner;

public class PrimeCheck {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Ingrese un número: ");
        int n = scanner.nextInt();
        boolean isPrime = true;

        if (n <= 1) isPrime = false;

        for (int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }

        if (isPrime)
            System.out.println(n + " es primo.");
        else
            System.out.println(n + " no es primo.");
    }
}
```

Python

```python
n = int(input("Ingrese un número: "))
is_prime = True

if n <= 1:
    is_prime = False

for i in range(2, n // 2 + 1):
    if n % i == 0:
        is_prime = False
        break

if is_prime:
    print(f"{n} es primo.")
else:
    print(f"{n} no es primo.")
```

Estructurada: cálculo numero primo

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PrimeNumber.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 N PIC 9(5).
77 I PIC 9(5).
77 FLAG PIC 9 VALUE 1.
PROCEDURE DIVISION.
    DISPLAY "Ingrese un número: ".
    ACCEPT N.

    IF N <= 1
        MOVE 0 TO FLAG
    END-IF.

    PERFORM VARYING I FROM 2 BY 1 UNTIL I > N / 2
        IF N MOD I = 0
            MOVE 0 TO FLAG
            EXIT PERFORM
        END-IF
    END-PERFORM.

    IF FLAG = 1
        DISPLAY N " es primo."
    ELSE
        DISPLAY N " no es primo."
    END-IF.

    STOP RUN.
```

Cobol

Estructurada – ordenación burbuja (imperativo y estructurado con funciones)

```
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]);
}
```

50	26	7	9	15	27	Array Original
Primera Pasada:						
26	50	7	9	15	27	Se intercambian el 50 y el 26
26	7	50	9	15	27	Se intercambian el 50 y el 7
26	7	9	50	15	27	Se intercambian el 50 y el 9
26	7	9	15	50	27	Se intercambian el 50 y el 15
26	7	9	15	27	50	Se intercambian el 50 y el 27
Segunda Pasada:						
7	26	9	15	27	50	Se intercambian el 26 y el 7
7	9	26	15	27	50	Se intercambian el 26 y el 9
7	9	15	26	27	50	Se intercambian el 26 y el 15

DECLARATIVA

Declarativa

Se basa en unidades conceptuales básicas que se pueden combinar según unas determinadas reglas para generar nueva información. ¿En qué dominio trabaja?

El dominio, será el conjunto de todas esas unidades conceptuales. ¿Cómo funciona?

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

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

Declarativa

¿En qué se diferencia fundamentalmente de la programación imperativa?

La programación imperativa se centra en el “cómo”, y la declarativa, en el “qué”.

En la programación declarativa, se describe directamente el resultado final deseado (el qué), mientras que en la imperativa son instrucciones paso a paso, describen de forma explícita como llegar al resultado.

¿qué ventajas podemos obtener con la programación declarativa?

Declarativa: ventajas

Descripciones compactas y muy expresivas.

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

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:

mayores = filtrar(X>5,lista);

LÓGICA

Lógica: subparadigma de programación declarativa

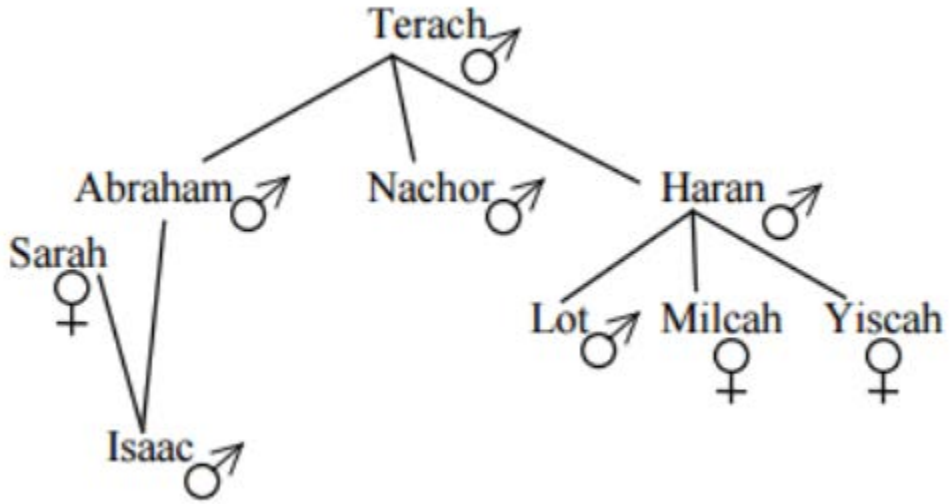
Utiliza el predicado lógico como ***concepto descriptivo básico.***

El dominio (mundo) se describe mediante los predicados: que relacionan los objetos según las reglas de la lógica de predicados.

Resolución: se plantean afirmaciones que el sistema resolverá (si existe solución) en base a los predicados.

PROLOG

Lógica: ejemplo de relaciones de descendencia




```
es_hijo(X,Y)

es_hija(X,Y)

es_abuelo(X,Z):
```

Lógica: ejemplo de relaciones de descendencia

```
es_padre(terach, abraham).
es_padre(terach, nachor).
es_padre(terach, haran).
es_padre(abraham, isaac).
es_padre(haran, lot).
es_padre(haran, milcah).
es_padre(haran, yiscah).

es_madre(sarah, isaac).

es_hombre(terach).
es_hombre(abraham).
es_hombre(nachor).
es_hombre(haran).
es_hombre(isaac).
es_hombre(lot).

es_mujer(sarah).
es_mujer(milcah).
es_mujer(yiscah).

es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).

es_hija(X,Y):- es_padre(Y,X), es_mujer(X).

es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

```
2 ?- es_padre(haran,lot), es_hombre(lot).

3 ?- es_padre(abraham,lot), es_hombre(lot).

4 ?- es_padre(abraham,X), es_hombre(X).

5 ?- es_padre(haran,X), es_mujer(X).

6 ?- es_hijo(lot,haran).

7 ?- es_hija(X,haran).
```

FUNCIONAL

Funcional: cálculo numero primo?

```
//Opcion 1
esPrimo :: Integer -> Bool
esPrimo n
  | n < 2             = False
  | n `elem` [2, 3]   = True
  | n `mod` 2 == 0 || n `mod` 3 == 0 = False
  | otherwise         = null [ x | x <- [5, 7..isqrt n], n `mod` x == 0 || n `mod` (x + 2) == 0 ]
  where isqrt = floor . sqrt . fromIntegral
```

Funcional: subparadigma de programación declarativa

Definir QUÉ CALCULAR, pero no cómo calcularlo

Utiliza la 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: no se respeta en lenguajes imperativos

PROCEDIMIENTOS VS FUNCIÓN

```
int x = 0;
void incrementar() {
    x++; // Efecto secundario: modifica la variable global x
}
```

Efecto colateral: ocurre cuando una función no solo devuelve un resultado, sino que también **modifica el estado** fuera de su propio contexto, como actualizar una variable global, modificar una estructura de datos mutable, realizar operaciones de entrada/salida, etc.

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. ¿qué pasa con las variables?

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…

Asignación, salvo en declaración.

No existen los bucles…

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

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**

Veámoslo por partes

Funciones de orden superior: Regla de declaración de tipo

TypeDecl ::= **typedef** **id** **equal** TypeExpr **semicolon**

Esto indica que una declaración de tipo comienza con la palabra clave **typedef**, seguida de un identificador (**id**), luego el símbolo **=**, una expresión de tipo (**TypeExpr**), y finaliza con un **punto y coma**.

Funciones de orden superior: Expresión de tipo

TypeExpr ::= (TypeList **arrow**)* TypeList

La expresión de tipo permite definir una lista de tipos (**TypeList**), seguida de una o más flechas (**->**) que indican que una función **toma varios argumentos y retorna un tipo final**.

Funciones de orden superior: Lista de tipos

```
TypeList ::=  TypeBase  ( comma TypeBase )*
```

La lista de tipos está formada por uno o más tipos base (**TypeBase**), separados por comas.

Funciones de orden superior: Tipo base

```
TypeBase ::= Type  |  lparen TypeExpr  rparen
```

La lista de tipos está formada por uno o más tipos base (**TypeBase**), separados por comas.

Funciones de orden superior: ejemplo 1

Declaración de una función que suma dos enteros:

Identificador del tipo: Suma

Expresión de tipo: Int -> Int -> Int

```
typedef Suma = Int -> Int -> Int;
```

TypeDecl ::= **typedef** **id** **equal** TypeExpr **semicolon**

TypeExpr ::= (TypeList **arrow**)* TypeList

TypeList ::= TypeBase (**comma** TypeBase)*

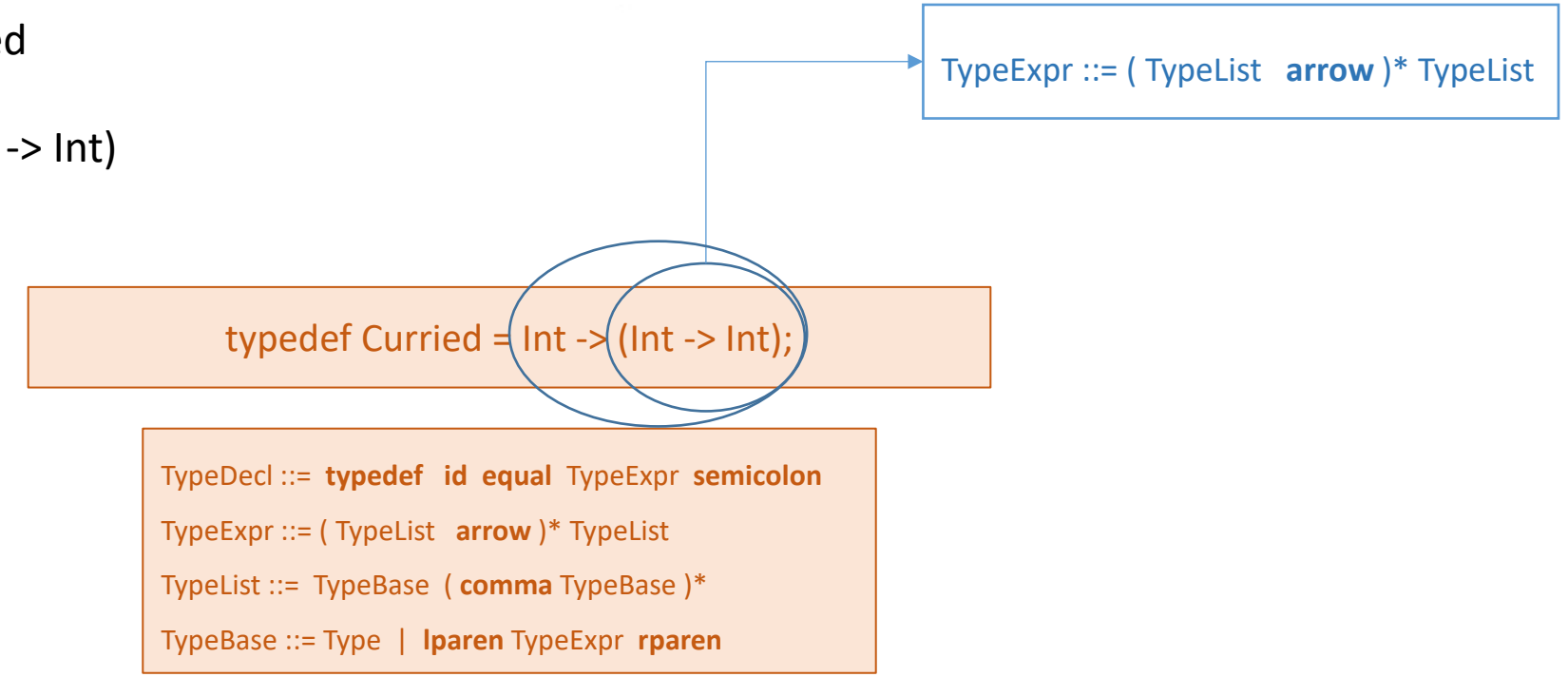
TypeBase ::= Type | **lparen** TypeExpr **rparen**

Funciones de orden superior: ejemplo 2

Declaración de una función que toma un entero y devuelve una función:

Identificador del tipo: Curried

Expresión de tipo: Int -> (Int -> Int)

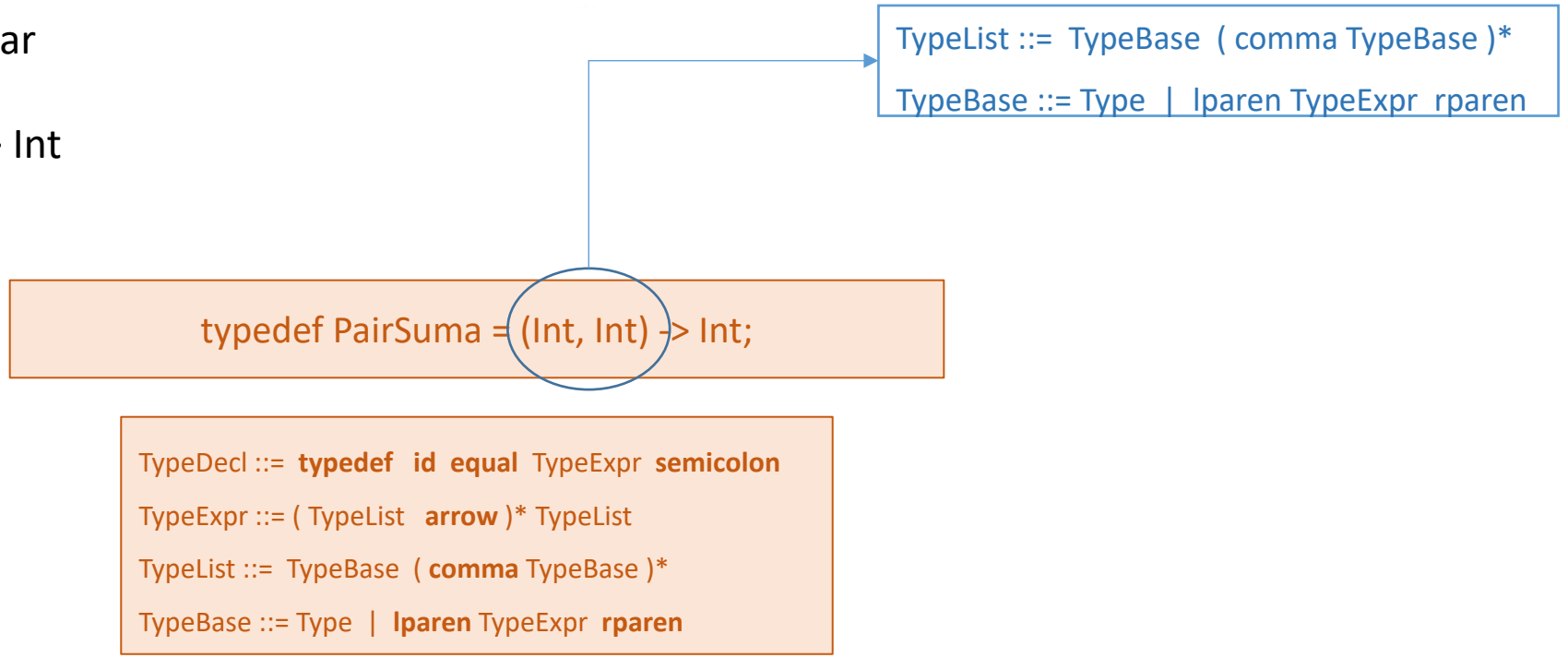


Funciones de orden superior: ejemplo 3

Declaración de una función que toma una tupla de dos argumentos:

Identificador del tipo: Sumapar

Expresión de tipo: (Int, Int) -> Int

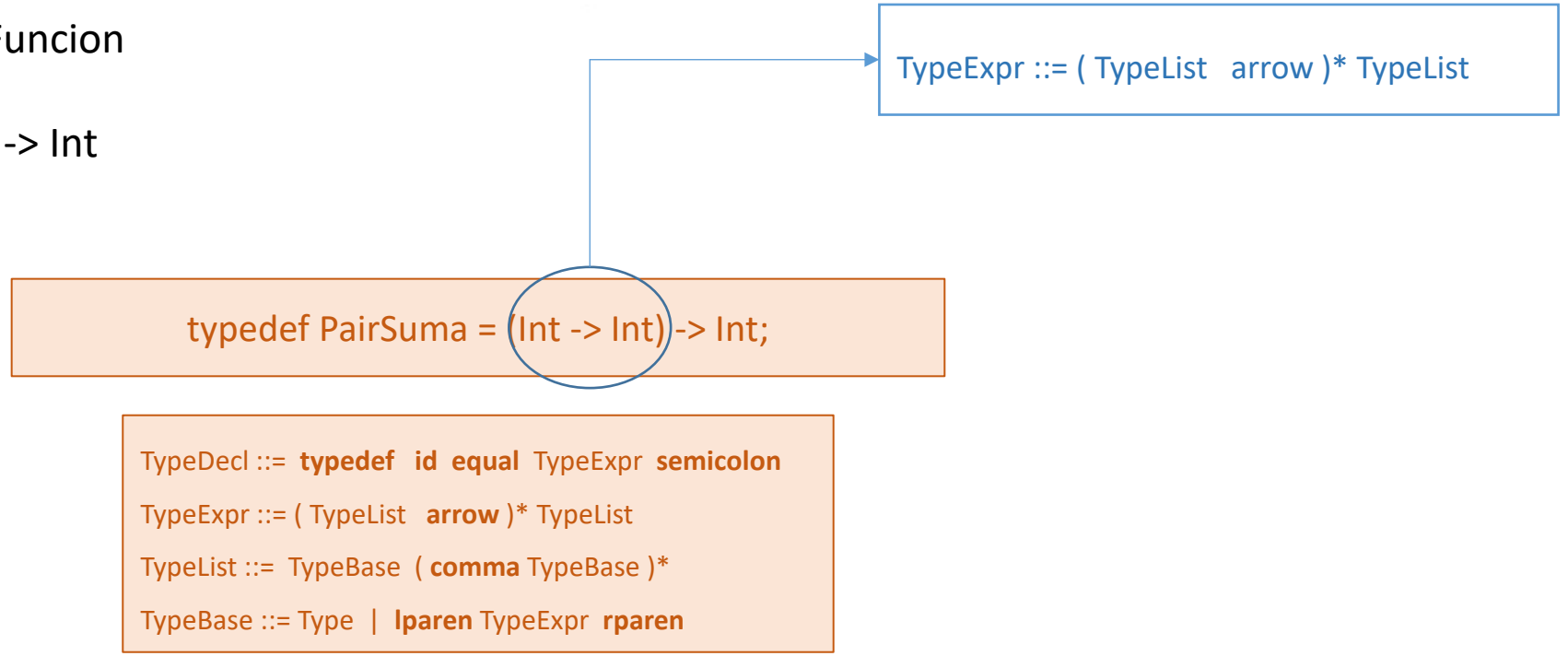


Funciones de orden superior: ejemplo 4

Declaración de una función que toma una función como argumento:

Identificador del tipo: EditarFuncion

Expresión de tipo: (Int -> Int) -> Int



Funciones de orden superior

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

```
miXor :: Bool -> Bool -> Bool
```

```
Integral a => a -> a
```

```
suma3::Int->Int->Int->Int
```

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 (TIPO DE EVALUACIÓN)

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 (TIPO DE EVALUACIÓN)

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

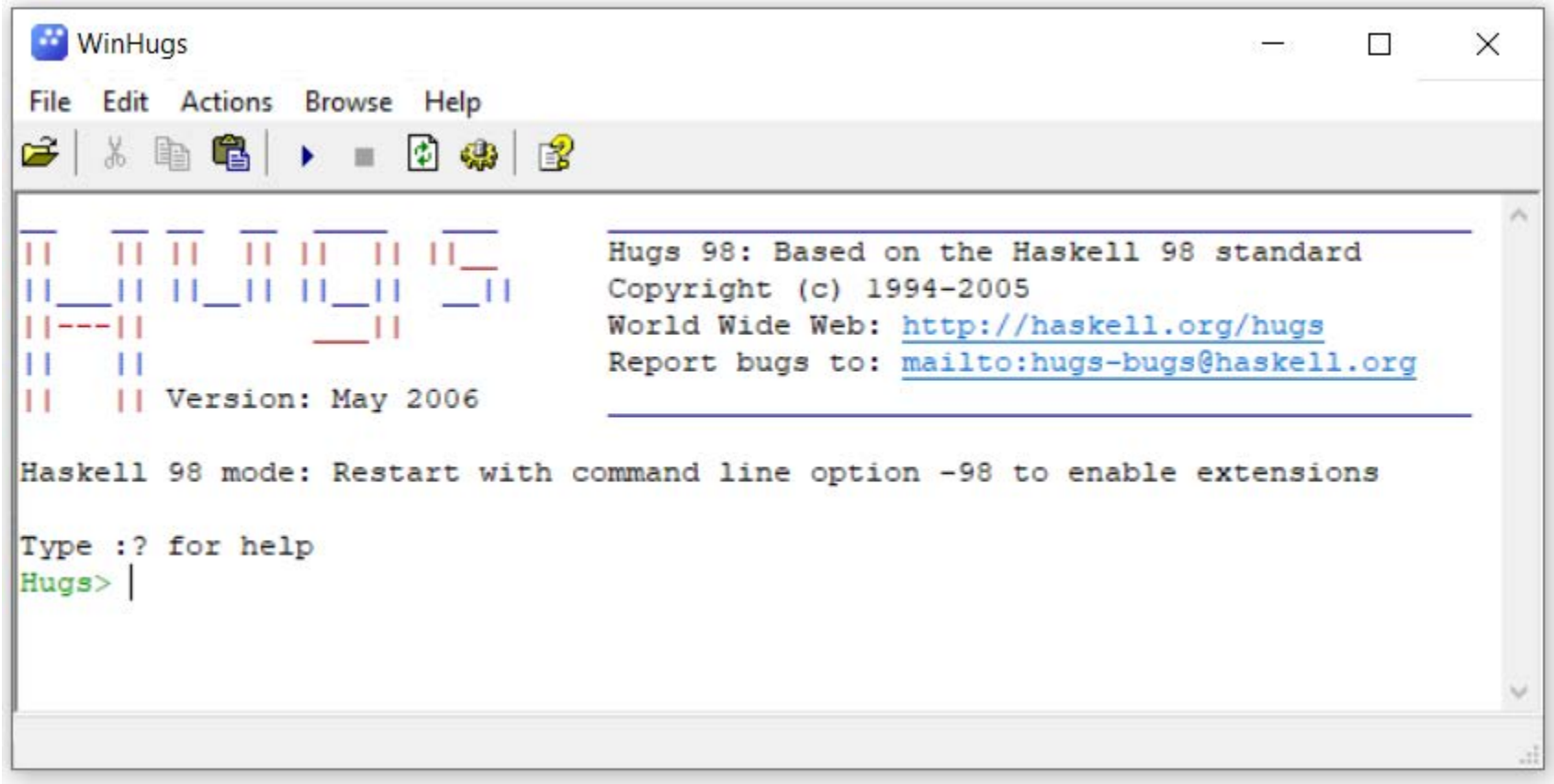
Existen diferentes implementaciones de Haskell

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

Para la realización de las prácticas comenzaremos con la implementación Hugs (**http://haskell.org/hugs/**)

Descargar e instalar

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**
- **Proyecto final en Scala, GHC, Hugs ….**

1.3. HASKELL

HASKELL

- Tipos y funciones básicas
- Definición de tipos
- Programación de funciones
- Entrada/Salida
- Testado de programas
- Mónadas
- Manejo de errores
- Programación paralela y concurrente

Scala

Escala es un lenguaje de programación **híbrido**

- Orientado a objetos
- Programación funcional

Se utiliza para aplicaciones en clústeres u ordenadores multicore. Padre e Apache Spark (Big Data)