# Paradigmas de Programación (EIF-400) **Introducción LP y Paradigmas Parte B**

CARLOS LORÍA-SÁENZ

AGOSTO 2023

EIF/UNA

# Objetivos



- Dominio Computacional: Conocimiento, Lenguajes, Máquinas, Paradigmas
- Abstracción computacional clásica
  - Datos (memoria)
  - Operaciones y control
- Modelos teóricos de computación: Lenguajes Formales y Máquinas (autómatas)
- Facetas: Declarativa vs Operacional
- Dinámico versus estático
- Interpretado versus Compilado

# Dominio, Conocimiento y Representación



- **Dominio:** realidad o mundo de interés
- **Declarativo**: objetos, propiedades, relaciones estáticas del dominio
- **Operativo** (procedimental): acciones, operaciones sobre los objetos (dinámica del dominio)
- **Meta** (reflexivo): Conocimiento sobre lo que se conoce
- **Lenguajes**: representaciones del conocimiento
- **Metalenguajes**: lenguajes para representar lenguajes

# Lenguajes



- Comunicar mensajes entre "**agentes**"
- **Vocabulario** (símbolos básicos)
- Reglas de **Sintaxis** del mensaje
- Reglas de **Semántica** del mensaje
- Capacidad **Generativa**: Emitir el mensaje
- Capacidad **Reconocedora**: "Entender" (parsear) el mensaje
  - Entender sintácticamente
  - Entender semánticamente

# Lenguaje Natural



- Meta-objetos: sustantivos, verbos, adjetivos, adverbios, artículos,…,
- Verbo/adverbio ≈ método/función
- Sustantivo/adjetivo ≈ objeto/dato
- Pregunta central paradigmática:
  - ¿Sustantivo o Verbo primero?
  - ¿En que enfocarse al iniciar una abstracción?

# Traducción y Máquina



- Problema:
  - Agente $P$ genera lenguaje $S$
  - Agente $M$ reconoce lenguaje $T$
  - $S \neq T$
- **Traductor**: Agente $C$ que convierte $S$ en $T$.
- Un **Compilador** es un agente (programa) traductor
- **Caso especial** agente $M$ es una "***máquina***"

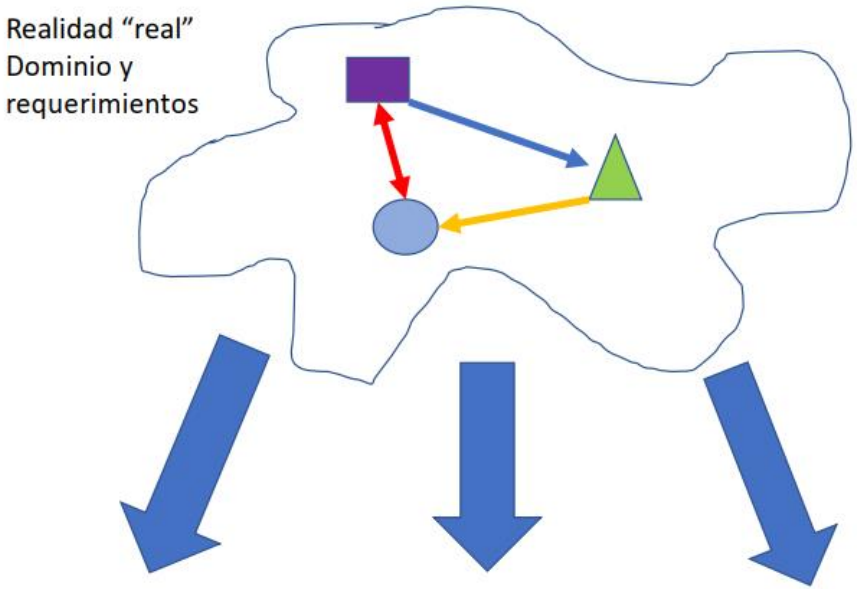
# Paradigmas



- Paradigmas de Programación: enfoques computacionales para abstracción y representación del dominio de estudio (mundo)
- ¿Cómo modelar el mundo (requerimientos)?
- Conceptos para expresar computación
- Ejemplos: "***objeto***", "***función***", "***regla***"
- Una forma de "expresar las cosas" y patrones a seguir, métodos, enfoques, estilos acordes a ese enfoque
- El lenguaje de programación trata de *facilitar* el paradigma en su *diseño*, *sintaxis* y *semántica*

# Idea





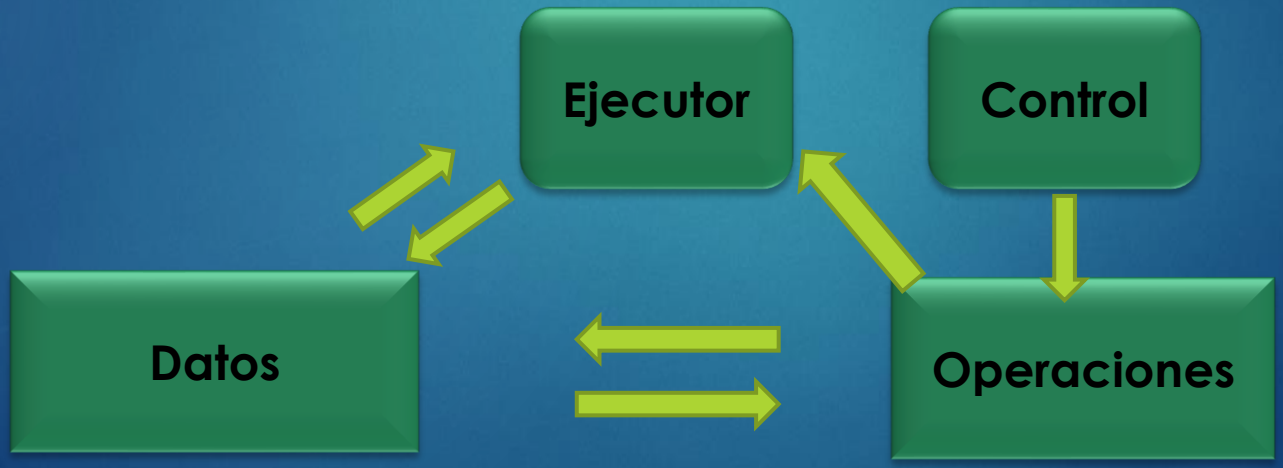

Objetos (datos) | Relaciones (asociaciones) | Funciones (métodos)

Lenguaje (sintaxis/semántica)

# Abstracción de Máquina

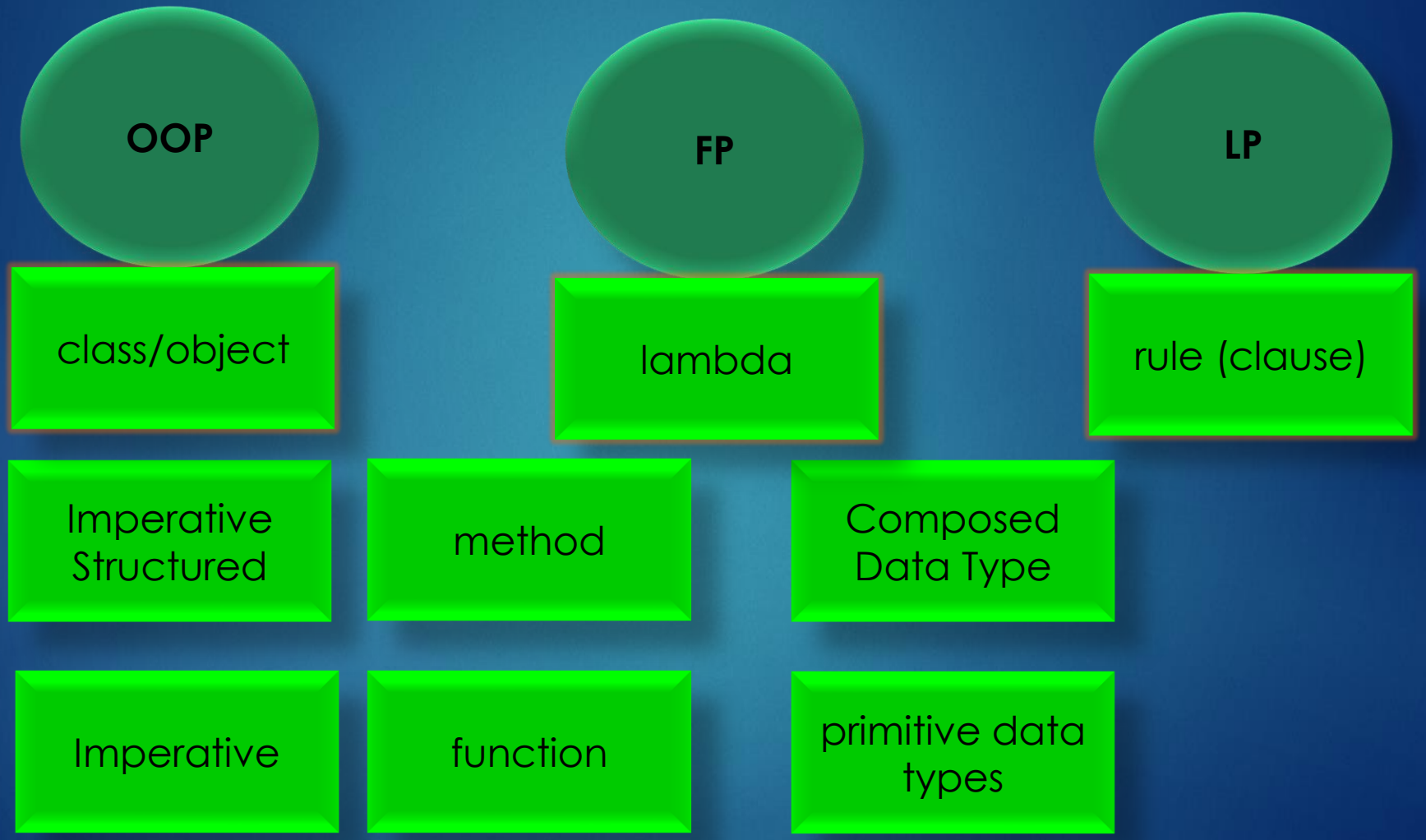


- **Datos** = Formas de memoria(s)
- **Operaciones** = estatutos, expresiones, operadores,…
- **Control** = orden (secuencia) de las operaciones
- **Ejecutor** = Ejecuta operaciones, almacena datos



# Paradigmas "*a lo clásico*"

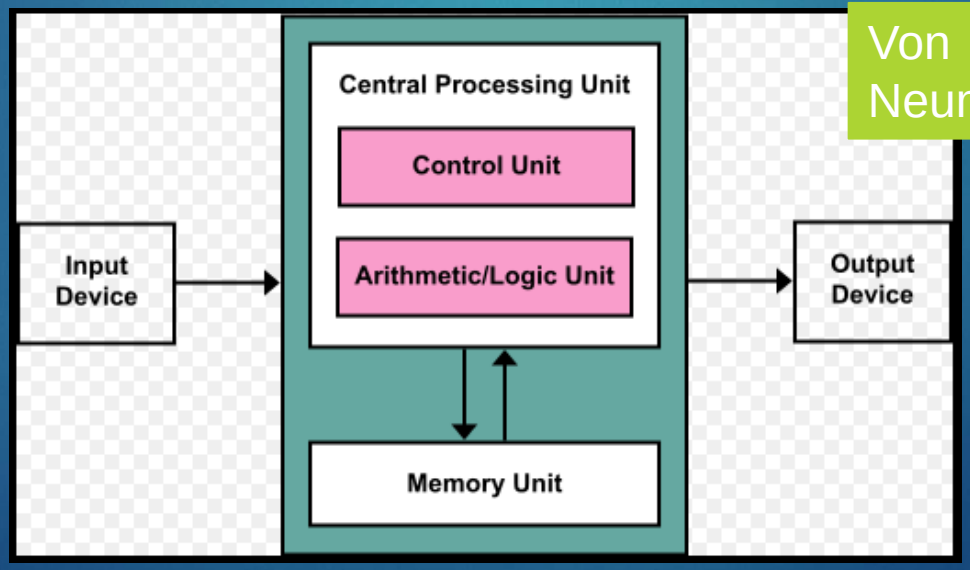




# “Máquina Subyacente”



- Clásica Imperativa: Modelo Von Neumann (Turing Machine)
- Cálculo Lambda: Programación Funcional (FP)
- Prueba de Teoremas: Programación Lógica (LP)




Von Neumann

De Wikipedia

# Máquinas



- Vocabulario (tipos de datos primitivos)
- Autómata Finito (memoria tamaño fijo)
- Autómata Finito de Pila
- Memoria finita versus infinita
- Memoria de pila versus RAM
- Determinismo versus No determinismo

# Modelo Turing/Von Neumann



- Modelo conceptual que simplifica la arquitectura
- CPU (unidad de operaciones)
- Memoria(s) (áreas de almacenamiento)
- Memoria sirve para datos y programas
- "Traer" operación actual (fetch, decode, execute)
- Operaciones para transferir datos entre memoria(s) al/del CPU (load, store)
- Operaciones lógico-aritméticas (add, compare, …)
- Operaciones para bifurcar control a un punto (branch, jump,…)

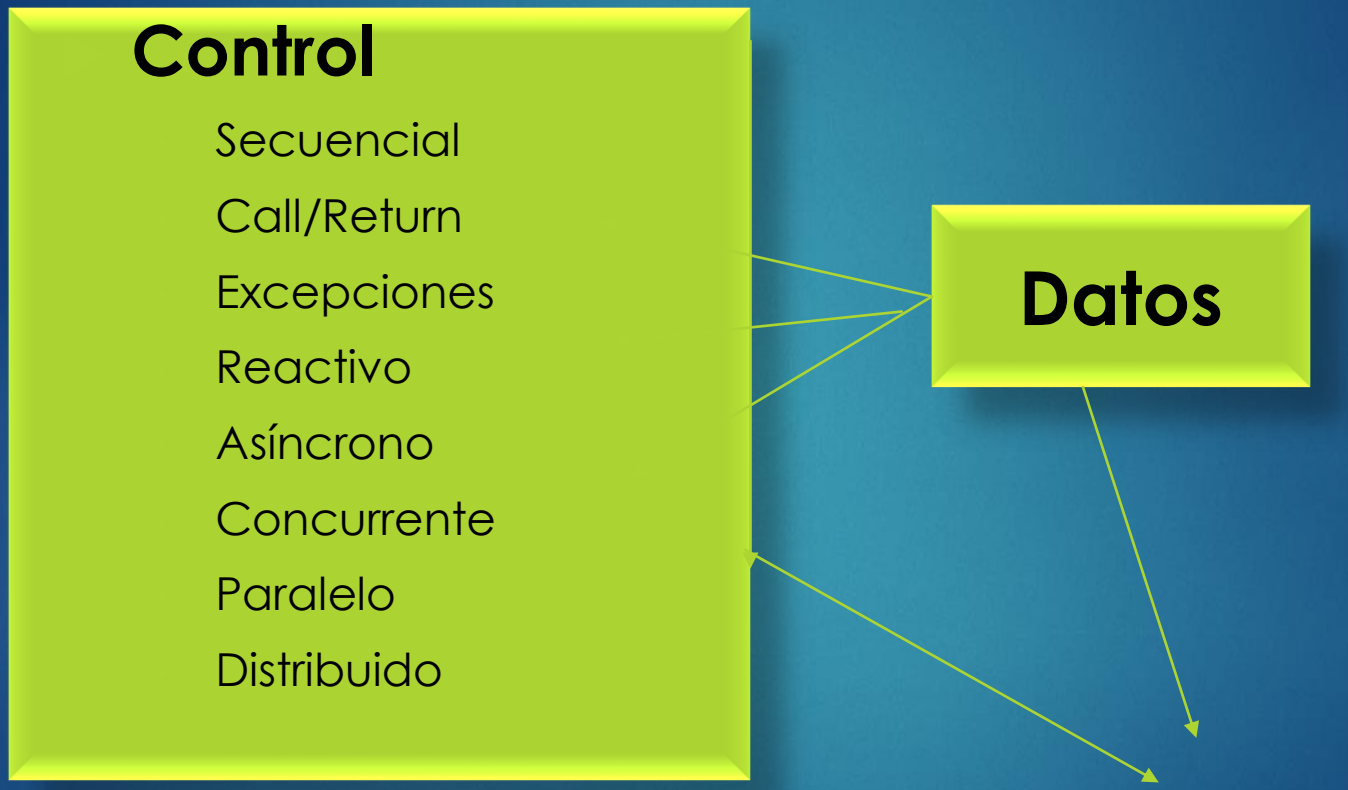
# "Máquinas" en FP/LP



- FP= "Functional Programming"
- LP="Logic Programming"
- Modelos "*poco convencionales*" con respecto a von Neumann/Turing
- **FP**: Máquina es un **simplificador de expresiones**
- Cumple "*leyes*" del Cálculo $\lambda$
- **LP**: Máquina **busca una prueba de un teorema**
- Cumple "*leyes*" de lógica formal (cálculo predicados)

# Facetas Control Alto nivel






- Se combinan con lo paradigmático: OOP, FP, LP

# Control imperativo no estructurado



- Imperativo con verbos lógico/aritméticos (add, cmp, goto, …)
- No hay estructuras de control (`while`, `if-then-else`)
- Control: `if-goto`, eventualmente condicional
- Datos: celdas de memoria y registros de máquina (CPU)
- Ejemplares: Assembler, primeros versiones lenguajes "alto nivel" antes de 1970 (`Fortran`, `Cobol`)

# Control Imperativo estructurado(alto nivel)



- Imperativo estructurado en control:
  - Control son verbos a la máquina subyacente (implícita): `=`, `while`, `if-then-else`, etc
  - Abstración de control: funciones (procedures, subrutinas) `call/return`
- Imperativo plano en datos:
  - Datos primitivos planos (`int`, `float`)
  - Abstracción Datos estructurados: `struct`, []
  - Pointers explícitos `*`. Memoria global (salvo `struct`)
  - Estado mutable por doquier
- Control y Funciones conceptos muy separados
- Típicos ejemplares: `C Pascal. Fortran/Cobol`

# Control imperativo y OOP



- Orientado a objetos en datos (declarativo)
  - La memoria se reparte en objetos (generaliza `struct`)
  - Puede ser manejada automáticamente (colector de basura)
  - Eventualmente `class`
  - Eventualmente herencia/polimorfismo
- Control se abstrae en métodos que pertenecen a los objetos (verbos de objetos)
- Métodos internamente tienen funcionalidad imperativa tradicional
- Función no es de "*primer piso*": no es tipo de objeto
- Ejemplares `C++` (>= `C11-17`), `Java` (>= jdk8)

# Excepción Interesante



- EL "*assembler*" (byte code) de la `JVM` es orientado a objetos
- ¿Por qué?
  - Eso facilita la compilación de `.java` a `.class`

# Tipo de Paradigmas: funcional (FP)



- La función es un tipo de dato (lambda). Preferible pura.
- Plano en datos: primitivos usuales, pero símbolos y listas son preferidos
- En control composición de funciones y recursión muy promovida
- Inmutabilidad es promovida (transparencia-referencial)
- Control imperativo: se prefiere formas de mayor nivel basadas en expresiones no estatutos (combinadores)
- *Pattern-matching*: un parámetro de una función puede ser una estructura de datos
- Manejo automática de memoria (Garbage-collected)
- Pueden ser dinámicos/estáticos. Último caso ¡inferencia de tipos! al rescate
- Abstracción de datos usando "`data types`" (generaliza `struct`, record, pero no necesariamente `class`)
- Pueden haber formas imperativos/mutadores (impuros)
- Ejemplares: `Lisp/Scheme`, `ML` (`Ocaml`, `F#`), `Haskell` (puro), `Python (OOP-FP)`, `Erlang`, `Scala (OOP-FP)`

# Data types (Ocaml)



- Árboles de Expresiones (expr) aritméticas y función to_string. Pruebe acá

Data type

Pattern-matching

```
type expr =
    | Plus of expr * expr          (* means a + b *)
    | Minus of expr * expr         (* means a - b *)
    | Times of expr * expr         (* means a * b *)
    | Divide of expr * expr        (* means a / b *)
    | Value of string              (* "x", "y", "n", etc. *)
;;

let rec to_string e =
    match e with
    | Plus (left, right) ->
      "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
    | Minus (left, right) ->
      "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
    | Times (left, right) ->
      "(" ^ to_string left ^ " * " ^ to_string right ^ ")"
    | Divide (left, right) ->
      "(" ^ to_string left ^ " / " ^ to_string right ^ ")"
    | Value v -> v
;;
```

# Ejercicio



- Reescriba en Java el ejemplo anterior para que compare el poder de pattern-matching y el tamaño del código

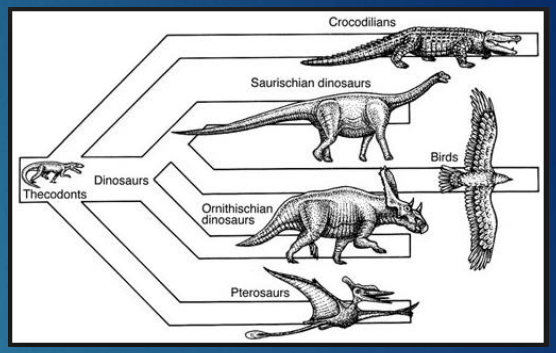
# Tipos de Paradigmas: Lógico tipo Prolog (1970)



- Datos usuales, simbólicos (listas)
- Declarativo uniforme en ambos datos/control: regla declara dato y operación (relación)
- Se promueve recursión
- Máquina muy lejana del tipo Von-Neuman
- Control es automático (backtracking)
- Garbage-collected
- Algo como pattern matching: unificación
- Hay primariamente relaciones
- Puede ser OO. Puede ser imperativo/mutable
- Típico ejemplar: `Prolog`

# Variantes/Combinaciones



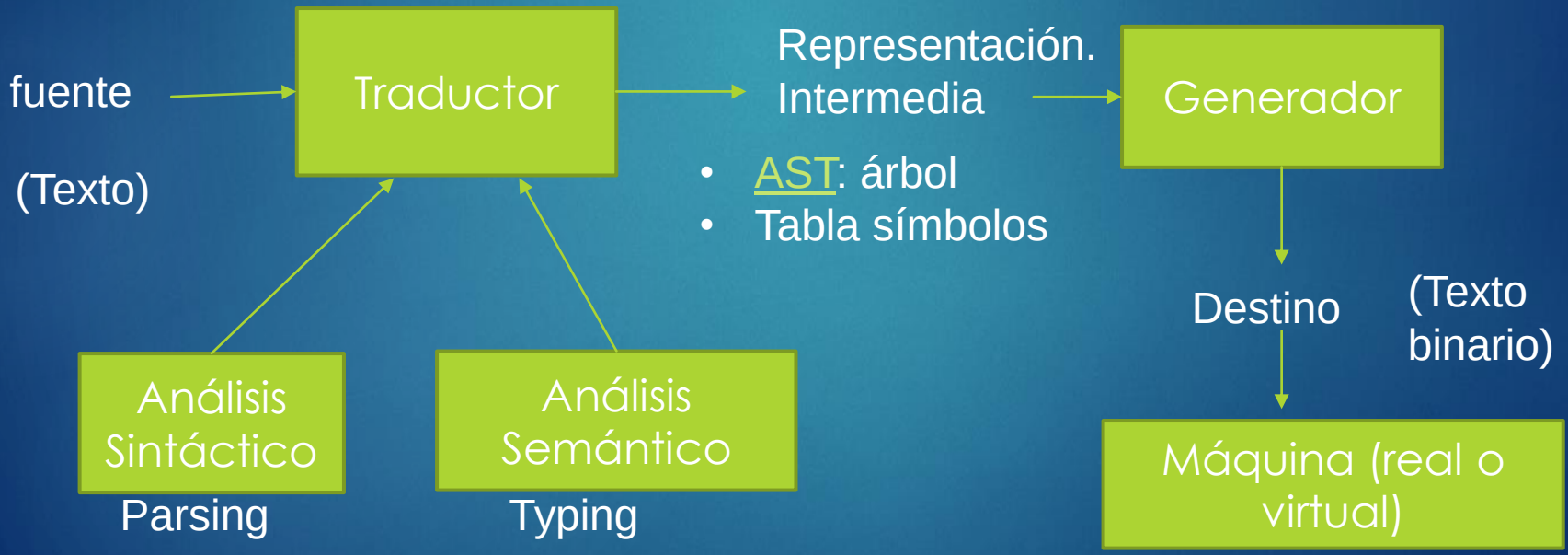



- Funcional/OO: Scala, Python, JS
- C# y Java>=8 tienen lambdas
- Funcional concurrente (reactivo): Erlang, Elixir Scala
- Imperativo concurrente OO no muy clásico: Go, Rust
- OO/Funcional/Lógico/DSL: Clips

# Compilador/Intérprete

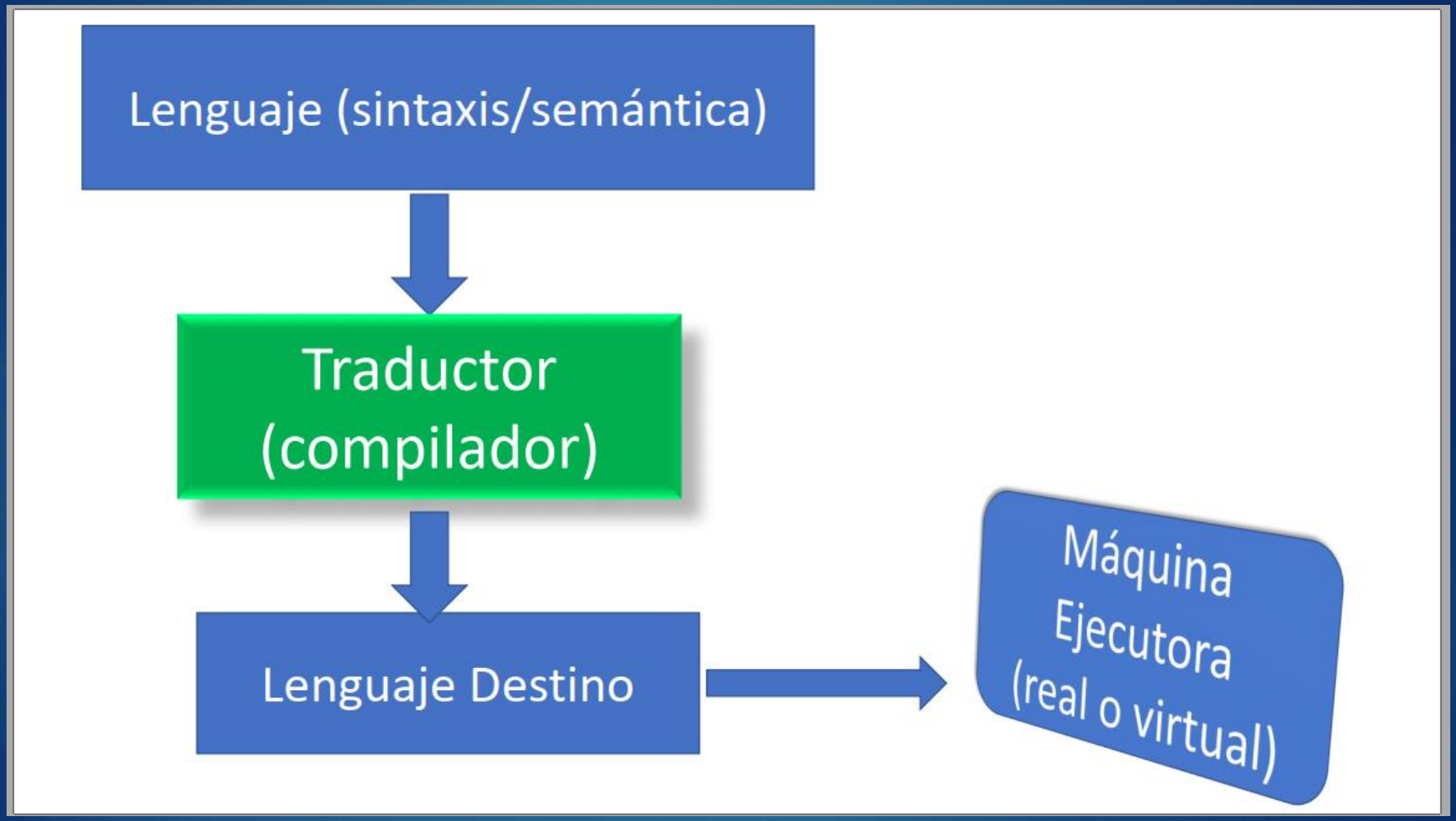


A **compiler** is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code.[1] The most common reason for converting source code is to create an executable program.



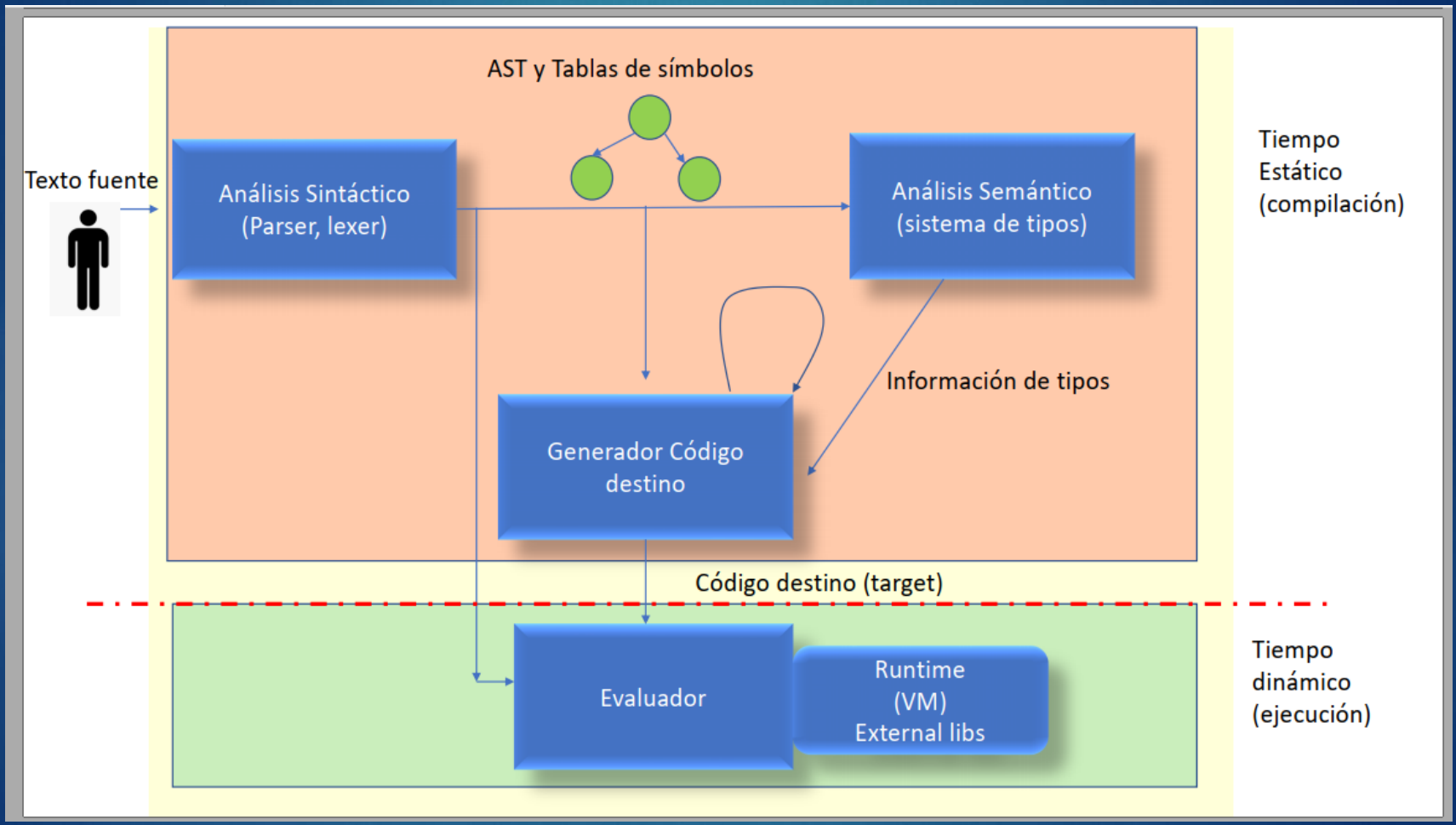
# Compilador




Lenguaje (sintaxis/semántica)
Traductor (compilador)
Lenguaje Destino
Máquina Ejecutora (real o virtual)

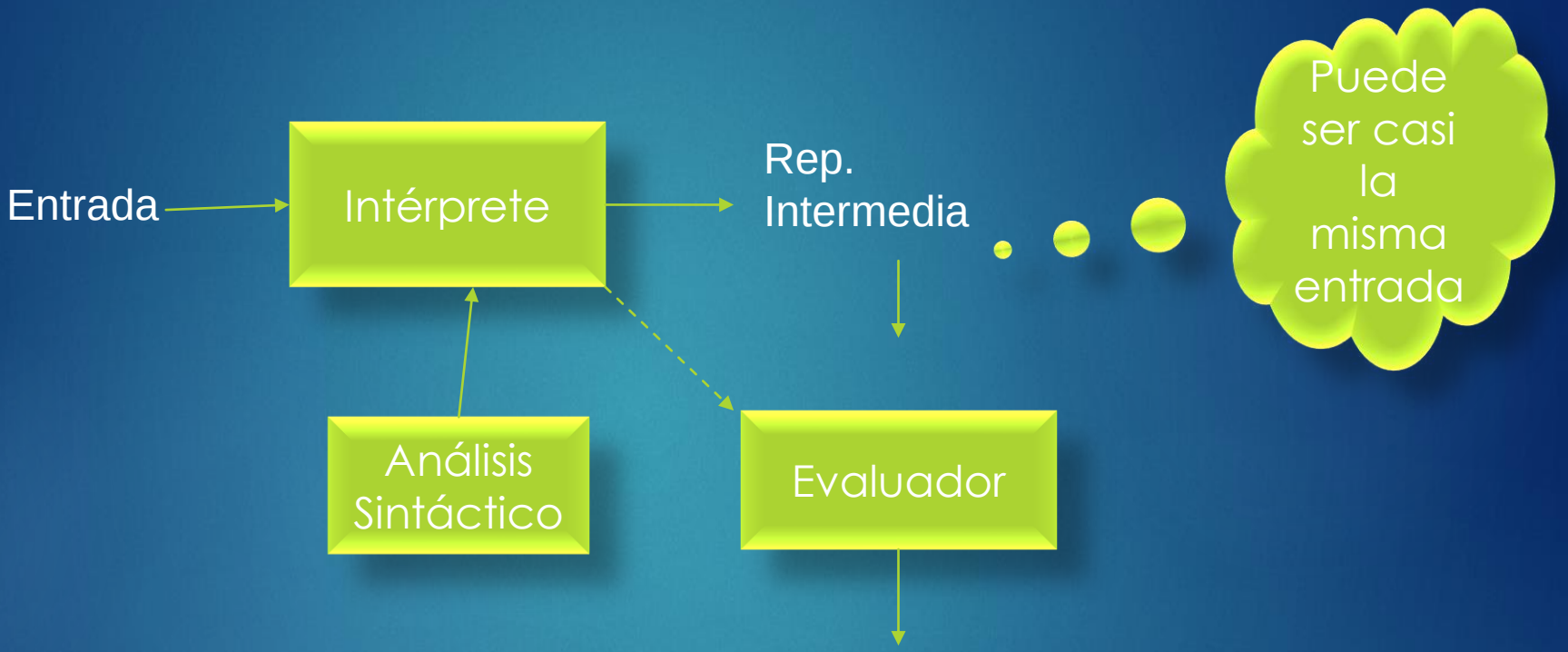
# Modelo más detallado





# Intérprete (usualmente)






Típico ejemplo: REPL (read-eval-print loop)

# Ejemplos: REPL aka Shells



- Python Shell
- Node REPL
- Mongo Shell
- Java>=9 Java Shell
- ....

```
PYTHON:python
Python 3.8.2 (tags/v3.8.2:7b3ab59,
Type "help", "copyright", "credits"
>>> a = 666
>>> a + a
1332
>>>
```

```
JS:node
Welcome to Node.js v13.13.0.
Type ".help" for more information.
> let a = 666
undefined
> a + a
1332
>
```

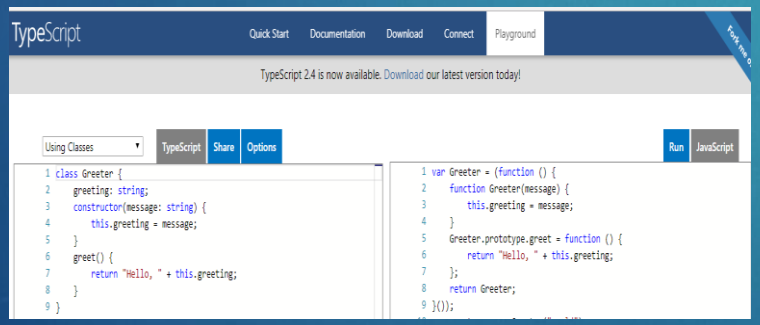
```
MONGO:mongo -nodb
MongoDB shell version v3.6.3
> let a = 666
> a + a
1332
>
```

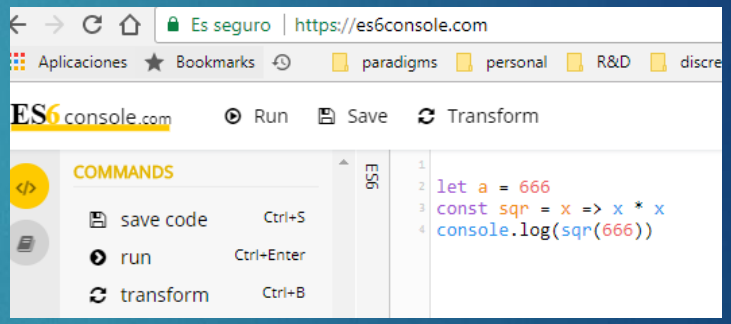
```
JAVA:jshell
|  Welcome to JShell -- Version 14
|  For an introduction type: /help intro

jshell> var a = 666
a ==> 666

jshell> a + a
$2 ==> 1332

jshell>
```

# Elemento interesante: Trans(com)pilación



- Fuente y Destino son lenguaje fuente
- Ejemplos Less, Sass → CSS
- ES6 → ES5: Pruebe acá ES6 Console
- Typescript → Javascript



# Algunos Elementos Relevantes



- "Máquinas virtuales y frameworks" (multiplataforma vs nativo)
- JVM y CLR ( .Net) (memoria manejada, GC)
- Los browsers: son la VM de JS de cliente

# Ejercicio: Explique la diferencia



```
JIT:java -version
java version "14" 2020-03-17
Java(TM) SE Runtime Environment (build 14+36-1461)
Java HotSpot(TM) 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)

JIT:javac -d classes src\java\Fibo.java

JIT:java -cp classes Fibo 10 1000000
*** Testing Fibo(10) 1000000 times ***

Ellapsed(sum=178000000) 0,243000 s

JIT:java -cp classes -Xint Fibo 10 1000000
*** Testing Fibo(10) 1000000 times ***

Ellapsed(sum=178000000) 5,017000 s

JIT:
```

# Lo mismo con C/C++



```
JIT:gcc -o fibo.exe src\c\fibo.c

JIT:fibo 10 1000000
*** Testing Fibo(10) 1000000 times ***
Ellapsed(sum=178000000) 0.393001s

JIT:
```

# Paradigma y Lenguaje



- ¿Por qué nace un "paradigma" `P`?
- ¿Por qué nace un lenguaje `L` para `P`?
- ¿Qué tan "puramente" refleja `L` a `P`?
- ¿Por qué evoluciona?
- ¿Se reproduce?
- ¿Muere?

# Patrones de Iteración y Control



- ¿Quién "*lleva el control*"
- El lenguaje versus la colección de datos
- Ejemplos/Ejercicios: Implementar en Python

$$s = \sum_{i=0}^{n} a_i$$

$$S = \{x^2 : \exists y \, (y \in N \ \wedge \ x = 2y + 1\}$$

# Lambdas: ¿son clases?



- No es necesario tener lambdas directamente en Java
- Pero mucho código sale muy “verboso” (ceremonioso) sin lambdas
- Las lambdas son más sucintas (si se usan bien)
- Adecuado para programación por eventos (reactiva)
- Uso especial: Listeners de GUI
- Concurrencia

# Ejercicio



- Estudie y compare `FrameTest.java` y `FrameTestLambda.java`
- Compile en consola y ponga a funcionar
- Note que el código es más simple (menos líneas de código)
- Menos imperativo más declarativo
- Estudie `StreamTest.java`. Compile y corra. Note los resultados en modo paralelo.

# Ejercicio



- Escriba un programa de consola en Java16 que reciba por consola una lista de enteros seleccione los pares,  los eleve al cuadrado e imprima el resultado
- No use ningún IDE. Trabaje en consola

# Nuevas formas de control



- Asincronía/concurrencia/paralelismo
- Promesas/Observables
- Programación reactiva (no hay un control principal)
- Eventos y el programa reacciona a los mismos
- Iteradores/Generadores/Async/Await (Python, ES6)

# Patrones control



- `for` → `for each` → `.forEach`
- Decisión: ¿quién tiene el control?
- ¿El lenguaje ó los objetos?
- ¿Qué es más orientado a objetos?
- ¿Qué es más declarativo?

# Poder Expresivo



- Requerimiento (establece un "*qué*")
- *El máximo común divisor "gcd(a, b)" entre dos enteros positivos "a" y "b" es el mayor entero "d" que los divide a ambos.*
- No dice "*cómo*"- Se necesita definición más "operacional" (alias un "cómo")
- Una definición recursiva de $\gcd(a, b)$ usando reglas:
- $\gcd(a, a) = a$; $\gcd(0, b) = b$; $\gcd(a, 0) = a$
- Si $a > b$ entonces $\gcd(a, b) = \gcd(a - b, b)$.
- Si $a < b$ entonces $\gcd(a, b) = \gcd(a, b - a)$

# Diferentes paradigmas



```c
// C plain
int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
     while ( a != b ){
         if ( a > b ) a = a - b;
         else b = b - a;
     }
     return a;
}
```

```
% Prolog reglas sabor imperativo
    gcd(A, A, A) :- !.
    gcd(0, B, B) :- !.
    gcd(A, 0, A) :- !.
    gcd(A, B, D) :- A > B, !,
                    A1 is A - B, gcd(A1, B, D).
    gcd(A, B, D) :- B1 is B - A,
                    gcd(A, B1, D).
```

```
// Kotlin recursivo por reglas
fun gcd( a: Int, b: Int) : Int =
 when {
    a == b -> a
    a == 0 -> b
    b == 0 -> a
    a > b  -> gcd(a - b, b)
    else   -> gcd(a, b - a)
  }
```

# Código x86 (MinGW)



```
00401460 <_gcd>:
401460:   55                    push   %ebp
401461:   89 e5                 mov    %esp,%ebp
401463:   83 7d 08 00           cmpl   $0x0,0x8(%ebp)
401467:   75 05                 jne    40146e <_gcd+0xe>
401469:   8b 45 0c              mov    0xc(%ebp),%eax
40146c:   eb 2c                 jmp    40149a <_gcd+0x3a>
40146e:   83 7d 0c 00           cmpl   $0x0,0xc(%ebp)
401472:   75 1b                 jne    40148f <_gcd+0x2f>
401474:   8b 45 08              mov    0x8(%ebp),%eax
401477:   eb 21                 jmp    40149a <_gcd+0x3a>
401479:   8b 45 08              mov    0x8(%ebp),%eax
40147c:   3b 45 0c              cmp    0xc(%ebp),%eax
40147f:   7e 08                 jle    401489 <_gcd+0x29>
401481:   8b 45 0c              mov    0xc(%ebp),%eax
401484:   29 45 08              sub    %eax,0x8(%ebp)
401487:   eb 06                 jmp    40148f <_gcd+0x2f>
401489:   8b 45 08              mov    0x8(%ebp),%eax
40148c:   29 45 0c              sub    %eax,0xc(%ebp)
40148f:   8b 45 08              mov    0x8(%ebp),%eax
401492:   3b 45 0c              cmp    0xc(%ebp),%eax
401495:   75 e2                 jne    401479 <_gcd+0x19>
401497:   8b 45 08              mov    0x8(%ebp),%eax
40149a:   5d                    pop    %ebp
40149b:   c3                    ret
```

```
// C plain
int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
     while ( a != b ){
         if ( a > b ) a = a - b;
         else b = b - a;
     }
     return a;
}
```

Símbolo del sistema

```
JS:gcc gcd.c -o gcd.exe

JS:objdump -D gcd.exe > gcd.code

JS:
```

X86 Assembler

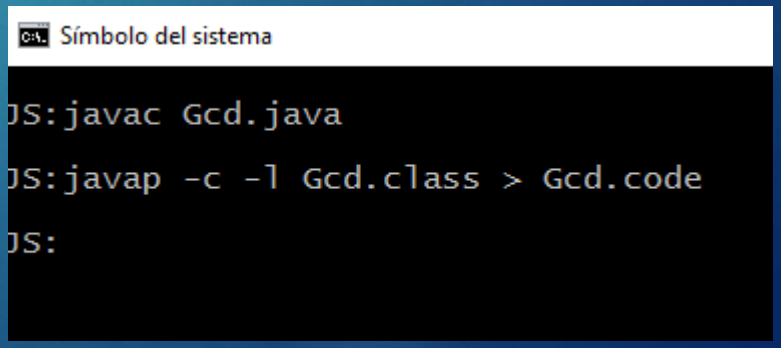
# Código JVM (bytecode)



```
static int gcd(int, int);
  Code:
     0: iload_0
     1: ifne          6
     4: iload_1
     5: ireturn
     6: iload_1
     7: ifne          12
    10: iload_0
    11: ireturn
    12: iload_0
    13: iload_1
    14: if_icmpeq     36
    17: iload_0
    18: iload_1
    19: if_icmple     29
    22: iload_0
    23: iload_1
    24: isub
    25: istore_0
    26: goto          12
    29: iload_1
    30: iload_0
    31: isub
    32: istore_1
    33: goto          12
    36: iload_0
    37: ireturn
```

a = #0 b=#1

```
static int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
     while ( a != b ){
         if ( a > b ) a = a - b;
         else return  b = b - a;

     }
     return a;

}
```



# Ejercicio 8bits gcd



- Considere el simulador 8bit
- Estudie el detalle del simulador acá
- Escriba un programa que calcule el `gcd` de dos números

```c
// C plain
int gcd(int a, int b){
    if (a == 0) return b;
    if (b == 0) return a;
     while ( a != b ){
         if ( a > b ) a = a - b;
         else b = b - a;
     }
     return a;
}
```