.UBAfiuba
FACULTAD DE INGENIERÍA

75.14 / 95.48 Lenguajes Formales
Prof. Dr. Diego Corsi
Lic. Pablo Bergman

# Intérpretes

## Conceptos fundamentales

# Manejo de la complejidad

En Informática:

1. **Objetos compuestos** → Composición de elementos primitivos
2. **Bloques de construcción de nivel superior** → Abstracción con objetos compuestos
3. **Modularidad** → Adopción de vistas a gran escala de la estructura del sistema

**Problemas cada vez más complejos**

↓

**Cualquier lenguaje de programación fijo no es suficiente**

↓

**Deberá recurrirse a nuevos lenguajes para expresar las ideas de manera más eficaz**

↓

# Abstracción metalingüística

**(establecimiento de nuevos lenguajes)**

# El ciclo de vida de los lenguajes de programación

**Objetivos:**

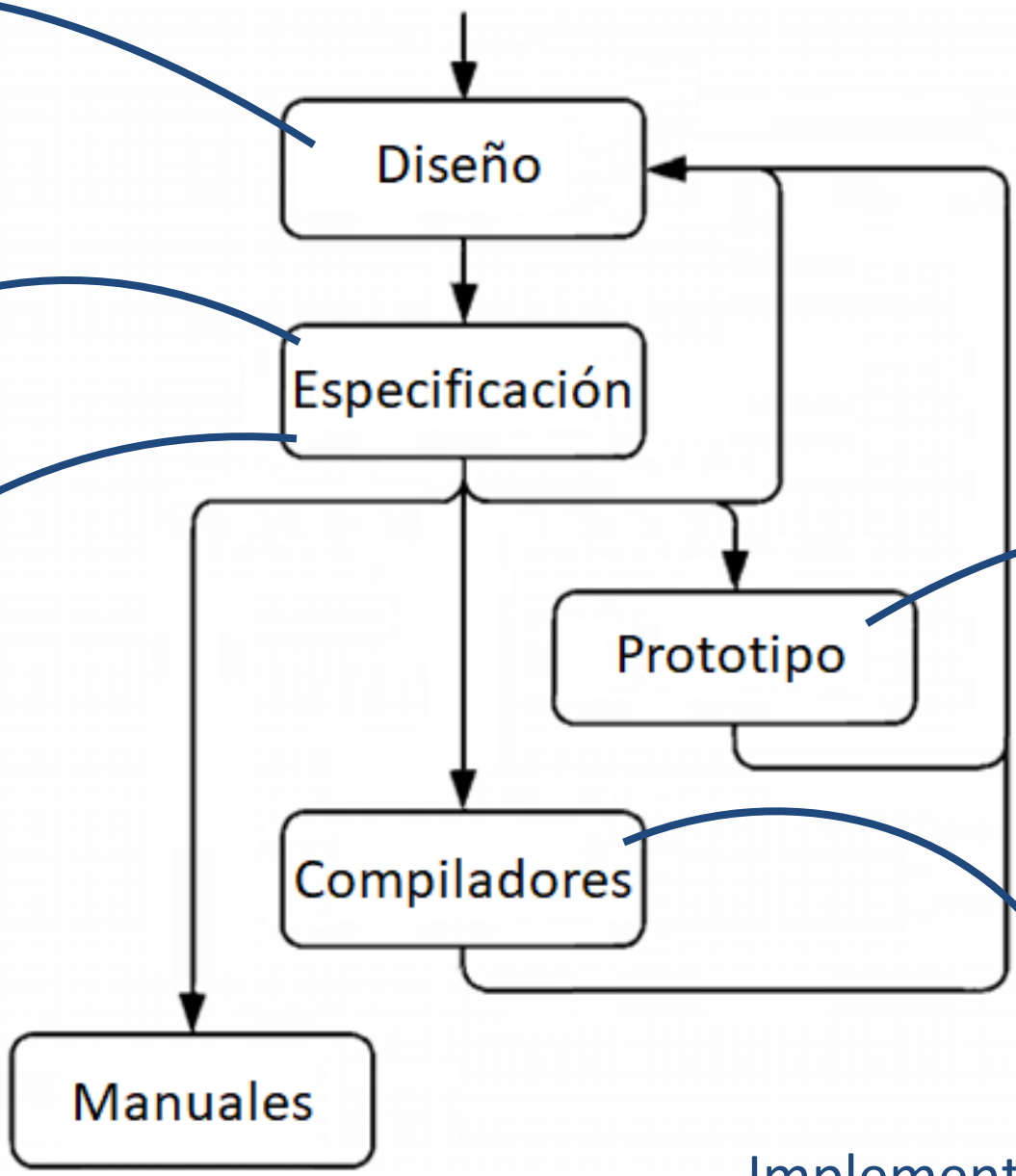
- Simplicidad
- Regularidad

**Modos:**

- Formal
- Informal

**Partes:**

- Sintaxis
- Restricciones contextuales
- Semántica




- Intérprete
- Compilador interpretativo

Implementación de alta calidad

# Niveles de lenguaje de programación

**Bajo nivel (código de máquina, lenguaje ensamblador)**

↓

Algoritmos expresados en términos de instrucciones primitivas, ejecutables directamente en el hardware electrónico

**Alto nivel (C, Python, Java, Clojure)**

↓

Algoritmos expresados en términos que están más cerca de la forma en que uno los conceptualiza

## Ventajas

- Resolución de problemas más rápida
- Más fáciles de leer, entender, mantener
- Más fáciles de aprender
- Más fácil de depurar
- Mayor portabilidad

## Desventajas

- Baja correspondencia con el hardware
- Menor eficiencia

# Una taxonomía de los procesadores de lenguaje

**Procesador de lenguaje:**
cualquier sistema que manipule programas expresados en algún lenguaje

↓

sirven para ejecutar programas, o prepararlos para ser ejecutados

- Editores
- Intérpretes
- Compiladores

- *Procesadores de lenguaje como herramientas de software:* `vi, javac, java`
- *Procesador de lenguaje integrado: Apache NetBeans*

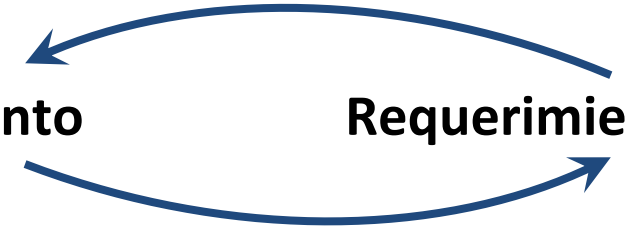
**Compiladores**

- Programa ejecutado en modo de producción
- Ejecución veloz es importante
- Cada instrucción se ejecuta con frecuencia
- Las instrucciones tienen formatos complejos

**Intérpretes**

- Programa utilizado una vez y descartado
- Funcionamiento veloz no es muy importante
- Cada instrucción se ejecuta una sola vez
- Las instrucciones tienen formatos simples
- Modo interactivo, resultados inmediatos

# Problemas prácticos de la interpretación



**Rendimiento** **Requerimiento de memoria**

| Inicio de ejecución | Eficiencia de ejecución | Ocupación de memoria |
|---|---|---|
| no se compila | **Interpretación de código fuente**<br>se repite el análisis de las instrucciones | coexisten el código fuente y un intérprete complejo |
| se compila | **Interpretación de código intermedio**<br>el análisis del código intermedio es simple | coexisten el código intermedio y un intérprete simple |
| se compila | **Interpretación de código de máquina (compilación completa)**<br>el análisis lo realiza el hardware | el código de máquina en la memoria está solo (no hay intérprete) |

# Consideraciones terminológicas

Si un sistema recibe un código fuente y lo traduce a código de máquina, ya sea para una computadora real o para una VM, el sistema es un **compilador**.

Un sistema que ejecuta el código fuente sin traducirlo primero a código de máquina es un **intérprete**.

## Compilación interpretativa

Un **compilador** genera código de máquina de una VM y un **intérprete** emula su ejecución.

### Ventajas

- **Código de máquina de la VM independiente de la arquitectura de la máquina real** (compilador más simple).
- **Código de máquina de la VM diseñado para ser compacto** (restricciones de memoria).
- **Mejor portabilidad** (ideal para entornos heterogéneos en red).
- **Mejor seguridad en la ejecución del programa** (depuración y supervisión).

### Desventaja

- **Peor eficiencia** (ejecución más lenta que el código nativo).

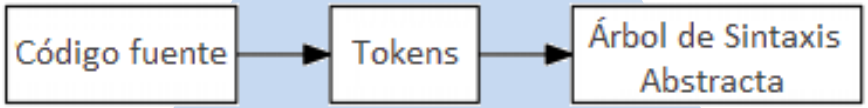
# Funcionamiento de los intérpretes

**Evaluación* inmediata del código fuente**

Intérpretes diminutos que ni siquiera analizan la entrada: *Brainfuck*

**Evaluación* de una representación intermedia del código fuente**

*Tree-walking interpreters* (también puede usarse otra RI que no sea un AST): *Ruby < 1.9*




**Compilación interpretativa 1**

Los *tokens* se traducen a *bytecode* que luego es evaluado: *Lua*

**Compilación interpretativa 2**

La RI se traduce a *bytecode* que luego es evaluado: *Ruby ≥ 1.9*

**Compilación JIT (*just in time* o "justo a tiempo") 1**

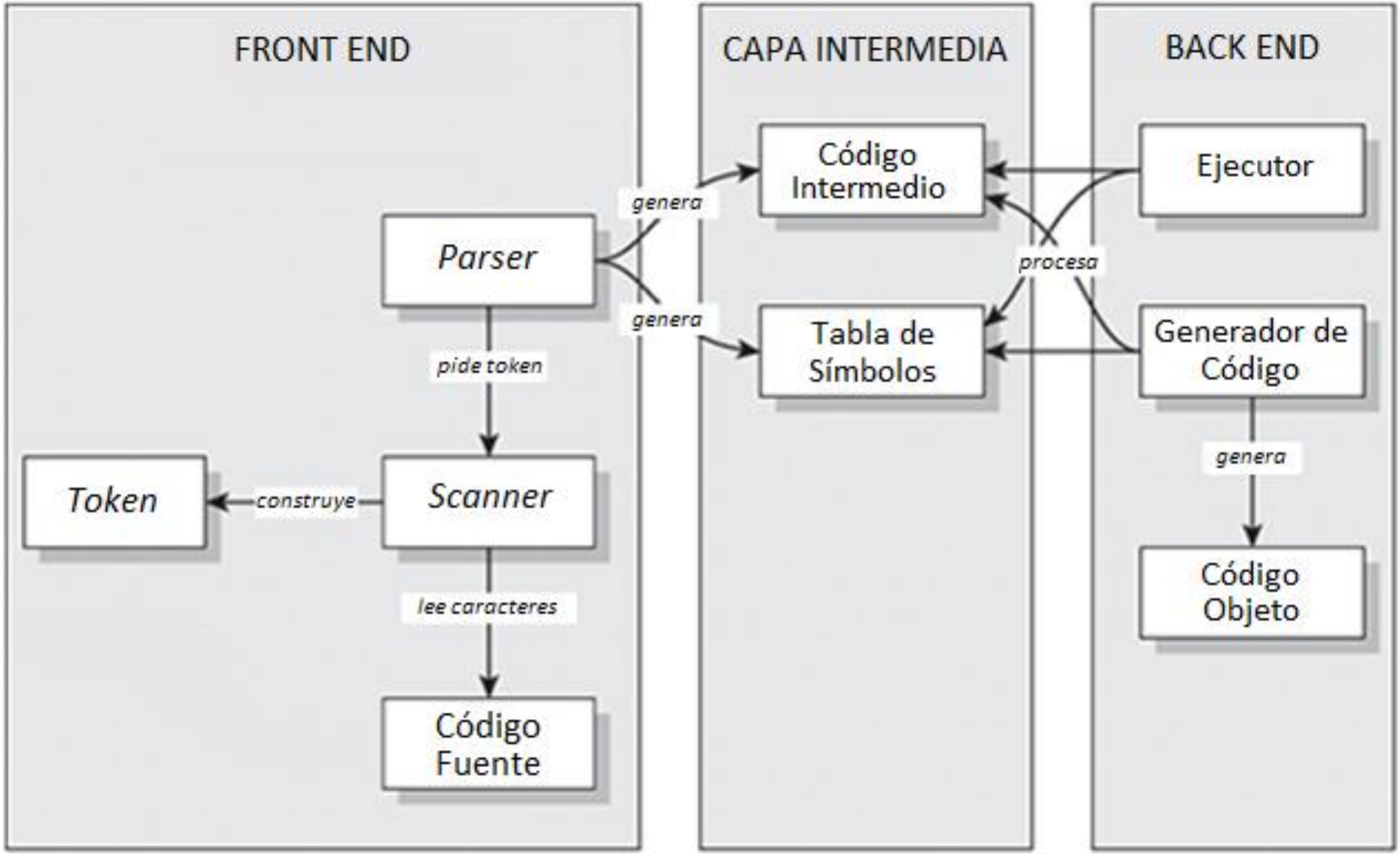
En lugar de generar *bytecode*, se genera código de máquina nativo: *Dart*

**Compilación JIT (*just in time* o "justo a tiempo") 2**

En lugar de evaluar el *bytecode*, se lo traduce a código de máquina nativo: *LuaJIT*

(*) modelo de evaluación basado en ambientes

# Diseño de intérpretes

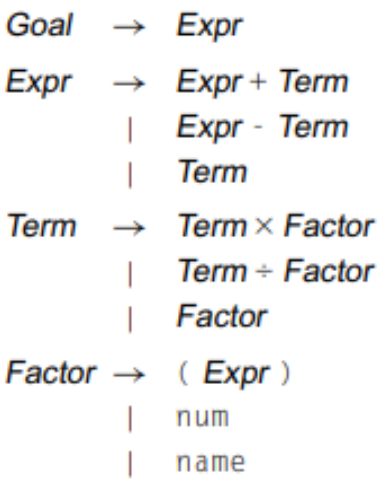


# Representaciones intermedias

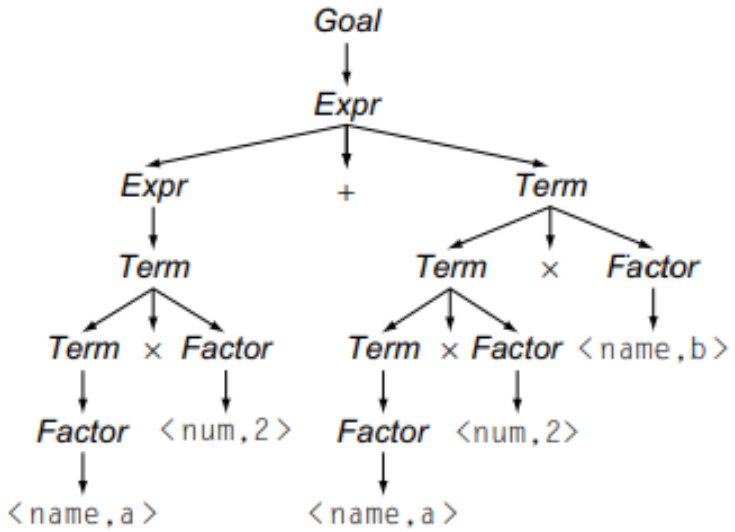
**RI lineales** (se asemejan al pseudocódigo de alguna máquina abstracta)

- Basadas en pila (JVM)
- Basadas en instrucciones con operador, hasta dos argumentos y un resultado (más populares)

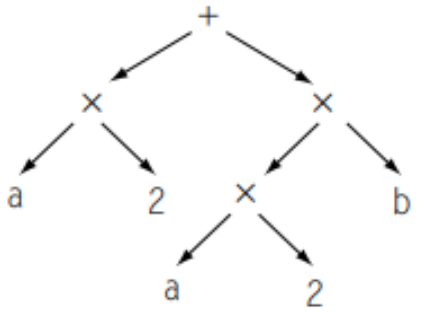
## RI gráficas



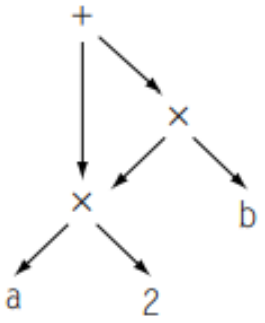

Gramática de *expresión*




Árbol de sintaxis




AST
(abstract syntax tree)




DAG
(directed acyclic graph)

## RI híbridas

- Por ejemplo, una RI lineal de bajo nivel para representar bloques de código lineal y un grafo para representar el flujo de control entre esos bloques

# Una taxonomía de los intérpretes

## Intérpretes iterativos

Esquema de interpretación iterativa

```
inicializar
hacer {
    buscar la siguiente instrucción
    analizar esa instrucción
    ejecutar esa instrucción
} mientras (aún esté en ejecución);
```

## Intérpretes recursivos

Esquema de interpretación recursiva

```
buscar y analizar el programa
ejecutar el programa
```

donde tanto el análisis como la ejecución son recursivos.

# Intérpretes iterativos

## Interpretación iterativa de código de máquina (emulador)

1) Se representa la memoria de la máquina mediante uno o más arreglos.
2) Se representan los registros de la máquina mediante variables (CP, RE).
3) Se busca cada instrucción de la memoria del código.
4) Se analiza cada instrucción aislando su código de operación y sus operandos.
5) Se ejecuta cada instrucción por medio de una sentencia condicional múltiple (un `switch`), con un caso por cada posible valor de *opcode*. En cada caso, **se emula la instrucción por medio de la actualización de la memoria y/o los registros**.

## Interpretación iterativa de lenguajes de comandos y de programación sencillos

- Es llevada a cabo mediante un *intérprete dirigido por la sintaxis* que imita lo que uno hace cuando lleva a cabo el seguimiento de un código fuente manualmente.
- A medida que se avanza en el código, se analizan, validan y ejecutan instrucciones.
- Todo sucede en el *parser* porque **un *intérprete dirigido por la sintaxis* no crea un AST ni traduce el código fuente a *bytecode* o a código de máquina**.
- El intérprete se guía directamente a través de la sintaxis para ejecutar las sentencias de una máquina abstracta.

# Intérpretes recursivos

Lenguajes de programación de alto nivel (Lisp, Prolog, etc.)

↓

El análisis de cada comando implica el **análisis recursivo** de sus subcomandos.
La ejecución de cada comando implica la **ejecución recursiva** de sus subcomandos.

## Intérprete basado en árbol

- Una *rutina de interpretación* para cada tipo de nodo en el AST.
- Un *indicador de estado* para dirigir el flujo de control. Incluye el *modo de funcionamiento* del intérprete (`NormalMode`, `ReturnMode`, `ExceptionMode`, `JumpMode`, etc.).

Los intérpretes recursivos típicos **analizan todo el programa fuente antes de que comience la ejecución**. Por lo tanto, renuncian a la inmediatez.

Son **más adecuados para implementar DSL** (*domain specific languages*) que para los lenguajes de programación de propósito general (que usan *compilación interpretativa*).

# Acerca del aprendizaje sobre compiladores e intérpretes

**¿Por qué aprender sobre compiladores e intérpretes?**

**¿Por qué la mayoría de las instituciones de educación superior en informática ofrecen cursos sobre este tema y, a menudo, los hacen obligatorios?**

Algunas razones típicas son :

(a) Se considera un tema que uno debe conocer para tener una "buena cultura" en informática.

(b) Un buen artesano debe conocer sus herramientas, y los compiladores e intérpretes son herramientas importantes para los profesionales informáticos en general y los programadores en particular.

(c) Las técnicas utilizadas para construir un compilador o un intérprete también son útiles para otros propósitos.

(d) Existe una buena posibilidad de que un programador u otro profesional informático necesite escribir un compilador o un intérprete para un DSL (lenguaje específico de dominio).

# Conclusión

Escribir un compilador o intérprete es un excelente proyecto educativo y mejora las habilidades de comprensión y diseño de lenguajes de programación, diseño de algoritmos y estructuras de datos y una amplia gama de técnicas de programación

**El estudio de compiladores e intérpretes hace que uno sea un mejor programador.**

# Bibliografía


Abelson, H., Sussman, G. J. y Sussman, J. (1996). *Structure and interpretation of computer programs* (2ª Ed.). MIT Press.

Ball, T. (2017). *Writing an interpreter in Go*. Edición del autor.

Cooper, K. D. y Torczon, L. (2012). *Engineering a compiler* (2ª Ed.). Elsevier/Morgan Kaufmann.

Friedman, D. P. y Wand, M. (2008). *Essentials of programming languages* (3ª Ed.). MIT Press.

Grune, D., van Reeuwijk, K., Bal, H., Jacobs, C. y Langendoen, K. (2012). *Modern compiler design* (2ª Ed.). Springer.

Mak, R. (2009). *Writing compilers and interpreters: A modern software engineering approach using Java* (3ra Ed.). Wiley.

Mogensen, T. Æ. (2017). *Introduction to compiler design*. Springer.

Parr, T. (2010). *Language implementation patterns: Create your own domain-specific and general programming languages*. Pragmatic Bookshelf.

Watson, D. (2017). *A practical approach to compiler construction*. Springer.

Watt, D. A. y Brown, D. F. (2000). *Programming language processors in Java: Compilers and interpreters*. Pearson Education Ltd.