Atlas de lenguajes — Polyglot Programming Labs

60 fichas de lenguaje · historia, características y el programa de la clase 041 en cada uno
github.com/vladimiracunadev-create/polyglot-programming-labs · Licencia MIT

- 🦴 C — 1972
- 🍀 Clojure — 2007
- 🏛️ COBOL — 1959
- 🧠 Lisp / Common Lisp — 1958
- 🚀 C++ — 1985
- 🟪 C# — 2000
- 🅳 D — 2001
- 🎯 Dart — 2011
- 📄 Datalog — 1977
- 🏗️ Delphi / Object Pascal — 1995
- 💧 Elixir — 2011
- 🌳 Elm — 2012
- 📝 Emacs Lisp — 1985
- 📞 Erlang — 1986
- 🔬 Fortran — 1957
- 🔵 F# — 2005
- 🐹 Go — 2009
- 🎸 Groovy — 2003
- λ Haskell — 1990
- ➗ J — 1990
- ☕ Java — 1995
- 🟨 JavaScript — 1995
- 📋 JCL — década de 1960
- 🔬 Julia — 2012
- 🟠 Kotlin — 2011
- 🌙 Lua — 1993
- 📐 MATLAB — 1984
- 🏥 M / MUMPS — 1966
- 👑 Nim — 2008
- 📱 Objective-C — 1984
- 🐫 OCaml — 1996
- 📘 Pascal — 1970
- 🐪 Perl — 1987
- 🐘 PHP — 1995
- 🧩 PL/I — 1964
- 🔷 PowerShell — 2006
- 🔗 Prolog — 1972
- 🐍 Python — 1991
- 📊 R — 1993
- 🏸 Racket — 1995
- 🏢 RPG — 1959
- 💎 Ruby — 1995
- 🦀 Rust — 2010
- 🔺 Scala — 2004
- ◯ Scheme — 1975

- 🎈 Smalltalk — década de 1970
- 🗄️ SQL — 1974
- 🕊️ Swift — 2014
- 🔧 Tcl/Tk — 1988
- 🔷 TypeScript — 2012
- 📊 VBA — 1993
- 🟦 VB.NET — 2002
- ⚡ Zig — 2016

Índice de las fichas

> 🗂️ *Índice de clases*

Este es el índice de las 60 fichas de lenguaje del repositorio: una por cada lenguaje que aparece en el programa, sea del núcleo que se implementa en las 136 clases, de los primos del Atlas que aparecen en cada `primos.md`, o de los lenguajes vivos que aparecen en cada `vivos.md`.

Todas siguen la misma estructura: **por qué está en el programa**, tabla de identidad, **historia**, dónde vive hoy, **lo que enseña y no enseña ningún otro**, lo que se ha modernizado, cómo se ejecuta hoy con órdenes reales, **el programa de la clase 041 explicado línea a línea**, y bibliografía con libros y fuentes concretas.

> ⚠️ ***Qué está verificado y qué no.*** *El **programa de la clase 041** de cada ficha es el mismo de la clase, y su estado de verificación es el que declara la clase 040: **las 10 implementaciones del núcleo y 3 primos (Ruby, Perl, Lua) se ejecutan en CI**; los **8 lenguajes vivos compilables** también. El resto son **material de lectura**: están escritos para ser correctos y **no llevan el sello de la máquina**. Cada ficha lo dice en su apartado.*

🔟 Los diez lenguajes del núcleo

Se implementan y se verifican en CI en **las 136 clases de código**. Son los representantes de familia del programa.

Ficha	Año	Familia que representa	Lo que aporta al programa
Python	1991	Scripting dinámico	Legibilidad como criterio; el pegamento del sistema
JavaScript	1995	JavaScript / web	Prototipos, bucle de eventos, coerción débil
TypeScript	2012	JavaScript / web	Tipado gradual y **borrado** de tipos
Java	1995	JVM	Máquina virtual con JIT, recolector, interfaces
C#	2000	.NET	Genéricos **reificados**; origen de `async` / `await`
Go	2009	Sistemas	**CSP**: gorrutinas y canales; lenguaje pequeño
Rust	2010	Sistemas	**Propiedad y préstamo**; concurrencia sin carreras
C	1972	C / llaves	La **ABI** común; memoria manual
SQL	1974	Lógica y declarativa	Declarar el **qué**, no el cómo
PHP	1995	Scripting dinámico	Ejecución **sin estado** por petición

🦕 Los dieciocho lenguajes que siguen vivos

Aparecen en el `vivos.md` de cada clase. **Ocho se ejecutan en CI**; el resto se declaran como adaptados o de lectura. Su índice propio, con los tres niveles de rigor, está en **vivos.md**.

Ficha	Año	Dónde sobrevive	Lo que enseña
COBOL	1959	Banca, seguros, administración	Decimal exacto; el lote
Fortran	1957	Cálculo científico, HPC	Arreglos y vectorización
Ada	1980	Aviónica, ferrocarril, defensa	Tipos con dominio; contratos; SPARK
Pascal	1970	Enseñanza; Free Pascal	Lenguaje pequeño y legible
Delphi	1995	Escritorio de empresa	Componentes, RTTI, `of object`
Common Lisp	1984	IA simbólica, herramientas	Código como dato; imagen viva
AutoLISP	1986	AutoCAD	Lisp incrustado en un producto
Tcl	1988	Diseño de circuitos, Expect	El lenguaje **para ser incrustado**
Perl	1987	Texto, integración, bioinformática	Expresiones regulares; CPAN; taint
C++	1985	Motores, navegadores, BD	RAII; plantillas; comportamiento indefinido
RPG	1959	IBM i: gestión	Firma de programa de servicio; ILE
PL/I	1964	z/OS: seguros, banca	Ambición total y su precio
M / MUMPS	1966	Sanidad (VistA, Epic)	El dato persistente sin impedancia
Smalltalk	1972	Nicho; enorme influencia	Todo es objeto; MVC; refactorización
JCL	1964	z/OS	Orquestación de trabajos con dependencias
Assembler	1949	Firmware, núcleos, mainframe	Lo que hay debajo de todo
VBA	1993	Excel, Access, Office	El lenguaje incrustado con más usuarios
VB.NET	2002	Gestión de empresa	La ruptura de compatibilidad como lección

---

🌳 Por familias

🧱 C / llaves

Representante del núcleo: C. Memoria explícita, tipos declarados, sintaxis de llaves.

Ficha	Año	Lo que aporta
C	1972	La ABI común de la interoperabilidad (clase 157)
C++	1985	RAII, plantillas, abstracción sin coste (clase 132)
Objective-C	1984	Despacho dinámico de mensajes sobre C (clase 111)
Zig	2016	Asignador explícito y `comptime` (clases 128 y 122)
Nim	2008	Sintaxis de guion, binario nativo, macros sobre el AST
D	2001	CTFE, contratos y pruebas en el lenguaje (clase 139)

🌀 Scripting dinámico

Representantes del núcleo: Python · PHP.

Ficha	Año	Lo que aporta
Python	1991	Legibilidad como decisión de diseño
PHP	1995	El modelo sin estado por petición (clase 168)
Ruby	1995	Todo es objeto; bloques; clases abiertas
Perl	1987	Texto, CPAN, TAP y el modo *taint* (clase 153)
Lua	1993	El lenguaje incrustado por excelencia (clase 163)
Tcl	1988	Todo es cadena; todo es canal; Safe-Tcl
R	1993	Pensar en vectores; evaluación no estándar

🌐 JavaScript / web

Representantes del núcleo: JavaScript → TypeScript.

Ficha	Año	Lo que aporta
JavaScript	1995	Prototipos y bucle de eventos (clases 112 y 134)
TypeScript	2012	Tipado gradual estructural, con borrado
Dart	2011	**JIT para desarrollar, AOT para publicar** (clase 174)
Elm	2012	Sin excepciones en ejecución; la arquitectura Elm
ActionScript	1998	**Extinto**: la dependencia de plataforma (clase 164)

☕ JVM

Representante del núcleo: Java. Todos compilan al mismo bytecode.

Ficha	Año	Lo que aporta
Java	1995	JIT, recolector generacional, interfaces
Kotlin	2011	Nulabilidad en el tipo; concurrencia estructurada
Scala	2004	OO + funcional unificados; colecciones persistentes
Groovy	2003	El DSL de Gradle y Jenkins (clase 163)
Clojure	2007	Inmutabilidad total; identidad frente a valor

🟦 .NET

Representante del núcleo: C#. Un ABI común para varios lenguajes (clase 157).

Ficha	Año	Lo que aporta
C#	2000	Genéricos reificados; LINQ; `async` / `await`
F#	2005	Inferencia total; unidades de medida; proveedores de tipos
VB.NET	2002	El lenguaje intercambiable; la migración forzada

λ Funcional tipada (ML)

Sin representante en el núcleo — su influencia llega por Rust, Scala y F#.

Ficha	Año	Lo que aporta
Haskell	1990	Pereza por defecto; efectos en el tipo (clase 118)
OCaml	1996	Módulos y functores; ML pragmático (clase 149)
F#	2005	ML en un ecosistema empresarial
Elm	2012	La garantía que sale de la renuncia

🔬 Lisp

Sin representante en el núcleo.

Ficha	Año	Lo que aporta
Common Lisp	1984	Macros, condiciones con reinicios, imagen viva
Scheme	1975	Continuaciones; recursión de cola; macros higiénicas
Racket	1995	**Crear lenguajes** como biblioteca (clases 122 y 163)
Clojure	2007	Lisp inmutable sobre la JVM
Emacs Lisp	1985	Alcance dinámico, en vivo (clase 088)
AutoLISP	1986	Treinta años de rutinas en despachos de ingeniería

🔗 Lógica y declarativa

Representante del núcleo: SQL.

Ficha	Año	Lo que aporta
SQL	1974	El optimizador decide el cómo (clase 118)
Prolog	1972	Unificación, vuelta atrás, relaciones reversibles
Datalog	1977	**Termina siempre**: la garantía que sale de renunciar

📨 Concurrente / actor

Sin representante en el núcleo — lo más cercano es el CSP de Go.

Ficha	Año	Lo que aporta
Erlang	1986	Actores, supervisión y **"déjalo fallar"** (clase 133)
Elixir	2011	El mismo modelo con otra puerta de entrada (clase 164)

🔢 Array / científica

Sin representante en el núcleo.

Ficha	Año	Lo que aporta
Fortran	1957	BLAS/LAPACK; el rendimiento es memoria (clase 152)
APL	1966	La notación como herramienta de pensamiento
J	1990	Programación **tácita**: componer sin nombrar
Julia	2012	Despacho múltiple con JIT especializante (clase 111)
MATLAB	1984	Generación de código desde el modelo (clase 155)
R	1993	Estadística con el vector como unidad

📱 Móvil / moderno

Ficha	Año	Lo que aporta
Swift	2014	ARC sin recolector; actores en el tipo (clases 131 y 136)
Dart	2011	Dos compiladores para un lenguaje

🗄️ Históricos / shell

Ficha	Año	Lo que aporta
COBOL	1959	Decimal exacto; el lote; el sistema que sobrevive
BASIC	1964	Accesibilidad como diseño; el `GOTO` (clase 083)
Bash	1989	Tuberías: la composición de procesos (clase 161)
PowerShell	2006	**Objetos** por la tubería en vez de texto (clase 159)
JCL	1964	Orquestación con reanudación y recursos (clase 171)
Assembler	1949	El suelo: registros, pila y ABI (clases 127 y 157)

📐 Cómo leer estas fichas

Todas responden a las mismas preguntas, y en el mismo orden:

1. 🎯 **Por qué está en este programa** — el criterio de inclusión, explícito. Nunca "porque es antiguo" ni "porque es popular": **qué concepto aporta que los demás esconden**.
2. **Tabla de identidad** — año, autoría, familia, paradigma, tipado, memoria, ejecución y estado.
3. 📜 **Historia** — de dónde viene, qué problema resolvía y qué decisiones lo formaron.
4. 🏭 **Dónde vive hoy** — sectores y productos concretos, verificables.
5. 🧠 **Lo que enseña** — el apartado central: **el concepto propio, con su coste dicho en voz alta**.
6. 🔄 **Lo que se ha modernizado** — porque casi ninguno se quedó donde estaba.
7. ⚙️ **Cómo se ejecuta hoy** — órdenes reales, que funcionan.
8. 💉 **El programa de la clase 041** — el mismo problema en todos, explicado línea a línea **por comparación con las demás fichas**.
9. 📚 **Fuentes y bibliografía** — documentación oficial y libros concretos, con autoría.

Y todas comparten una regla: cuando un lenguaje **no puede** cumplir el contrato de la clase 041 —SQL, Datalog, Elm, ActionScript, JCL— **se declara la adaptación** en lugar de inventar un programa que no existiría (clase 040).

🧭 Y de vuelta al programa

- **Atlas de familias** — el árbol completo, con las cápsulas por familia.
- **Lenguajes que siguen vivos** — el índice de los 18, con los niveles de rigor.
- **Índice de clases** — las 176 clases del programa.
- **Clase 164: elegir el lenguaje correcto** — donde estas 60 fichas se convierten en una decisión.
- **Clase 176: transferencia a nuevos lenguajes** — y donde se explica para qué sirvió mirarlos todos.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 📚 Índice de clases

Los lenguajes que siguen vivos

🎯 Por qué esta sección existe

Ninguno de estos lenguajes está aquí por curiosidad histórica, por nostalgia ni por capricho. Están aquí porque **siguen en producción hoy**, y porque cada uno **deja a la vista un concepto que los diez lenguajes del núcleo esconden o dan por resuelto**.

El criterio de inclusión es doble y se aplica a todos por igual:

1. **Uso actual verificable.** Hay un fabricante que mantiene el compilador o el intérprete, hay documentación con fecha reciente, y hay sectores identificables donde el lenguaje ejecuta trabajo real todos los días. Cada ficha lo declara y lo enlaza.
2. **Aporte pedagógico propio.** El lenguaje enseña algo que el núcleo no enseña. No se incluye ningún lenguaje "porque es antiguo": se incluye porque **verlo cambia lo que entiendes de los modernos**.

> #### *Dos precisiones que este material no oculta*
>
> ***Primera: no todo se puede resolver con estos lenguajes, y donde no se puede, se dice.*** *JCL no calcula: orquesta trabajos. VBA vive dentro de Excel y no tiene* `stdin`*. AutoLISP no existe fuera de AutoCAD. RPG recibe sus datos por parámetros o por fichero, no por la entrada estándar. En esos casos* ***el contrato se adapta y la adaptación se declara en la página****, en lugar de inventar un programa falso que finja lo contrario. Un JCL que multiplicara números no enseñaría JCL: enseñaría una mentira.*
>
> ***Segunda: muchos de estos lenguajes se han actualizado para resolver problemas actuales.*** *No son fotos fijas. COBOL genera y analiza JSON con sentencias del lenguaje; Fortran descarga bucles a la GPU con* `do concurrent`*; RPG consume APIs REST con* `DATA-INTO`*; Ada 2022 comprueba contratos y SPARK los demuestra; Tcl 9.0 trajo Unicode completo; Perl 5.40 tiene* `try/catch` *y firmas de función; Delphi compila para iOS y Android; MUMPS habla FHIR y corre en contenedores. Por eso* ***cada ficha tiene una sección*** `🔄 Lo que se ha modernizado`*: sin ella, la lista parecería un cementerio, y no lo es.*

🗺️ Los dieciocho, en una tabla

Lenguaje	Año	Dónde sobrevive	Estado	El concepto que deja a la vista
COBOL	1959	Banca, seguros, gobierno, tarjetas, pensiones	🟢	Aritmética **decimal exacta** y la forma del dato declarada
Fortran	1957	HPC, clima, física, CFD, BLAS/LAPACK	🟢	El **array** como ciudadano de primera; el coste del *aliasing*
Ada	1983	Aviónica, espacio, ferrocarril, defensa	🟢	El **tipo lleva una regla del dominio**, no solo un tamaño
RPG	1959	IBM i: ERP, retail, logística, manufactura	🟢	El **ciclo del programa**: un bucle principal implícito
PL/I	1964	Mainframe z/OS: banca, seguros	🟡	El origen del `try/catch`; el coste de querer abarcarlo todo
M / MUMPS	1966	Sanidad, historia clínica, VistA, Epic	🟡	**El lenguaje y la base de datos son la misma cosa**
Smalltalk	1970s	Banca, seguros, trading, telecomunicaciones	🟡	OO puro: hasta el `if` **es un mensaje a un objeto**
Common Lisp	1958	IA simbólica, CAD, investigación, DSL	🟡	**Homoiconicidad y macros**: el código es un dato
AutoLISP	1986	AutoCAD: arquitectura, ingeniería, construcción	🟢	El **lenguaje incrustado** en una aplicación anfitriona
Tcl/Tk	1988	EDA (diseño de chips), redes, testing, GUI	🟡	**No hay sintaxis, solo comandos**: el `if` es de usuario
Perl	1987	Sysadmin, texto, bioinformática, web heredada	🟡	**Regex en la sintaxis** y el **contexto** de evaluación
Delphi / Object Pascal	1995	Escritorio empresarial, TPV, industria, ERP	🟢	**RAD y modelo de componentes**: propiedades y eventos
VBA	1993	Excel, Word, Access, AutoCAD, SolidWorks	🟢	**Una hoja de cálculo es un programa**; la UDF que extiende la fórmula
Pascal	1970	Educación, y sobre todo vía Delphi/Free Pascal	🟡	Tipos definidos por el usuario: **subrangos y enumerados reales**
JCL	1964	z/OS: proceso por lotes de banca y seguros	🟢	**Programa y entorno separados**: inyección de dependencias del SO
Assembler	1949	Firmware, núcleos, cripto, SIMD, embebidos	🟢	**El suelo**: qué es de verdad una variable, una pila, una llamada
C	1972	SO, firmware, redes, bases de datos, intérpretes	🟢	La **ABI universal**: el idioma en que los lenguajes se hablan
C++	1985	Videojuegos, navegadores, finanzas, HPC, robótica	🟢	**Abstracción de coste cero** y **RAII**

🟢 uso amplio y activo · 🟡 nicho vivo pero minoritario

🧪 Tres niveles de rigor, declarados

Este programa distingue siempre lo que **verifica una máquina** de lo que es **material de lectura**. Con estos dieciocho lenguajes, la clasificación es la siguiente:

Nivel 1 — Se ejecutan en CI contra el mismo `casos.json`

COBOL (GnuCOBOL), Fortran (gfortran), Ada (GNAT), Pascal y Object Pascal (Free Pascal), Common Lisp (SBCL), Tcl (tclsh), Perl, C y C++.

Su código se extrae del Markdown, se compila o interpreta y se ejecuta contra los mismos casos de prueba que las diez implementaciones del núcleo. Si el resultado no coincide, **la CI falla**. Es el mismo estándar que se aplica al núcleo, sin excepciones.

Nivel 2 — Contrato adaptado y declarado

RPG, JCL, VBA y AutoLISP.

Estos lenguajes **no pueden** cumplir el contrato `stdin → stdout` sin falsearse: no es una limitación del material, es la naturaleza del lenguaje. RPG recibe parámetros, JCL orquesta un programa que sí calcula, VBA lee celdas y AutoLISP pregunta al usuario o señala un objeto del dibujo. En cada página se explica cómo se adapta el contrato y por qué. **No se verifican**, y así se declara en cada aparición.

Nivel 3 — Correctos pero sin sello de máquina

PL/I, M / MUMPS, Smalltalk y Assembler.

Sí pueden expresar el contrato, pero su cadena de herramientas no está en los *runners* de CI: PL/I necesita z/OS, MUMPS y Smalltalk requieren instalaciones pesadas, y el ensamblador depende de la arquitectura concreta. El código está escrito para ser correcto e idiomático, **sin el sello de la máquina**. Verificar diez de dieciocho no es verificarlos todos, y conviene no vender lo contrario.

📖 Cómo está organizado el material

Cada lenguaje tiene **dos apariciones distintas**, que responden a dos preguntas distintas:

1. Su ficha en el Atlas — *¿qué es este lenguaje y por qué sigue vivo?*

Una página por lenguaje, con la misma estructura en las dieciocho: por qué está en el programa, historia, dónde sobrevive, por qué no ha muerto, **lo que se ha modernizado**, cómo se ejecuta hoy, el programa de la clase 041 explicado línea a línea, la tabla "si vienes de X...", los errores comunes al leerlo, y bibliografía real con libros concretos y documentación oficial fechada.

2. Su sección en cada clase — *¿cómo se ve ESTE concepto en este lenguaje?*

Cada clase de código del programa tiene un fichero `vivos.md` que resuelve **el problema de esa clase** en los lenguajes vivos que aporten algo a **ese concepto concreto**, con código y explicación. No es el mismo texto repetido 136 veces: la nulabilidad en COBOL (que no tiene `null`, y usa niveles `88`) no se parece en nada a la nulabilidad en Ada (que tiene `not null access`), y ahí está el valor.

Son 1632 programas, y cada página los agrupa por el rigor con que están respaldados:

Nivel	Lenguajes	Qué significa
🟢 Se ejecuta en CI	COBOL, Fortran, Ada, Pascal, Common Lisp, Tcl, Perl, C++	Se compila y se ejecuta contra el mismo `casos.json` que las diez implementaciones del núcleo
🟡 Contrato adaptado	RPG, JCL, VBA, AutoLISP	El lenguaje no puede expresar `stdin→stdout`; la adaptación se declara
🟣 Sin sello de máquina	PL/I, MUMPS, Smalltalk, ensamblador	Correctos y revisados, pero ningún compilador libre los verifica aquí

> 🗂️ ***Y más allá de estos dieciocho**, todos los lenguajes del repositorio tienen ficha con esta misma anatomía: **el índice de las 60 fichas** incluye los diez del núcleo y los primos del Atlas.*

🧭 Por dónde empezar

Si te interesa…	Empieza por
Entender el software que mueve dinero	COBOL → JCL → PL/I
El cálculo científico y el rendimiento	Fortran → C → Assembler
Los sistemas donde un fallo cuesta vidas	Ada → Pascal
De dónde salió la orientación a objetos	Smalltalk → C++ → Delphi
Metaprogramación y lenguajes que se extienden	Common Lisp → Tcl
Automatizar la herramienta que ya usas	VBA → AutoLISP
El software sanitario y las bases de datos raras	M / MUMPS
Procesar texto mejor que nadie	Perl → Tcl
Aplicaciones de escritorio empresariales	Delphi / Object Pascal → Pascal
El ERP que factura en media Europa	RPG → COBOL

📚 Bibliografía transversal

Libros que ayudan a leer **toda** esta sección, no un lenguaje concreto:

- **Robert W. Sebesta**, *Concepts of Programming Languages*, Pearson — el manual de referencia sobre diseño de lenguajes; dedica espacio real a COBOL, Fortran, Ada, Lisp y Smalltalk en su contexto histórico, que es exactamente lo que esta sección necesita.
- **Michael L. Scott**, *Programming Language Pragmatics*, Morgan Kaufmann — el porqué de las decisiones de implementación: tiempos de enlace, modelos de memoria, sistemas de tipos.
- **Bruce Tate**, *Seven Languages in Seven Weeks* y *Seven More Languages in Seven Weeks*, Pragmatic Bookshelf — el método de aprender un lenguaje por lo que aporta, que es el método de este programa.
- **Federico Biancuzzi, Shane Warden**, *Masterminds of Programming*, O'Reilly — entrevistas con los creadores de C++, Perl, Smalltalk, Objective-C, APL, Forth, Lua y varios más. La mejor fuente sobre *por qué* cada lenguaje es como es.
- **Jean Sammet**, *Programming Languages: History and Fundamentals*, 1969 — el censo clásico de los lenguajes de la primera era, escrito por alguien que estuvo en el comité de COBOL.
- **Peter Seibel**, *Coders at Work*, Apress — conversaciones con programadores que trabajaron con varios de estos lenguajes cuando eran la corriente principal.

- **ACM HOPL** (*History of Programming Languages*) — las actas de las cuatro conferencias (hopl.info (https://hopl.info/)) contienen los relatos de primera mano de los diseñadores de COBOL, Fortran, Ada, Lisp, Smalltalk, C++ y Pascal. Fuente primaria y de acceso público.

⏮️ Volver al Atlas · 📚 Índice de clases · 🏠 Volver al programa

El árbol de familias

El Atlas es la **segunda capa** del programa. Mientras el **núcleo** (10 lenguajes) se implementa y se verifica, el Atlas te deja **comprender ~40 lenguajes más por sus características**: su historia, su familia, su paradigma, su modelo de memoria y con qué software se ejecutan. La tesis: **aprende el representante, reconoce la familia entera.**

> *Este material es **de lectura** (historia, características, toolchain). No se ejecuta en CI. Cada ficha se fecha y enlaza a la documentación oficial, porque las versiones y herramientas cambian.*

Última revisión: 2026-07. Los datos históricos (autor, año, motivo) son estables; las herramientas y versiones no. Ante cualquier detalle operativo, la fuente de verdad es siempre la documentación oficial enlazada en cada cápsula.

🦕 Lenguajes que siguen vivos — la tercera capa

Además de estas cápsulas, el Atlas tiene **dieciocho fichas completas** de lenguajes antiguos que **siguen en producción hoy**: COBOL, Fortran, Ada, RPG, PL/I, M/MUMPS, Smalltalk, Common Lisp, AutoLISP, Tcl/Tk, Perl, Delphi, VBA, Pascal, JCL, Assembler, C y C++.

No están por nostalgia: están porque se usan. El criterio de inclusión es doble y se declara en cada ficha — **uso actual verificable** (fabricante que mantiene el compilador, documentación fechada, sectores identificables) y **aporte pedagógico propio** (un concepto que los diez del núcleo esconden). Cada ficha lleva además una sección `🔄 Lo que se ha modernizado`, porque muchos de estos lenguajes han incorporado JSON, REST, GPU, contenedores, Unicode o Git para resolver problemas actuales: no son fotos fijas.

Diez de los dieciocho se **ejecutan en CI** contra el mismo `casos.json` que el núcleo. En los que **no pueden** expresar el contrato de una clase —JCL no calcula, VBA vive dentro de Excel— la adaptación se explica en la página en lugar de inventar un programa falso.

👉 Índice de lenguajes que siguen vivos

🌳 El árbol, en una tabla

Familia	Representante del núcleo	Primos (se comprenden por características)	Idea que aporta
C / llaves	**C**	C++, Objective-C, Zig, Nim, D	Memoria, punteros, sintaxis brace
Scripting dinámico	**Python** / **PHP**	Ruby, Perl, Lua, Tcl, R	Tipado dinámico, rapidez de escritura
JavaScript / web	**JavaScript** → **TypeScript**	Dart, Elm, ActionScript	Prototipos, asincronía, web
JVM	**Java**	Kotlin, Scala, Groovy, Clojure	OO nominal en máquina virtual
.NET	**C#**	F#, VB.NET	Multiparadigma sobre el CLR
Funcional tipada (ML)	— (influye en **Rust**)	Haskell, OCaml, Elm, F#	Pureza, tipos algebraicos, inferencia
Lisp	—	Scheme, Racket, Common Lisp, Emacs Lisp	Homoiconicidad, macros
Lógica y declarativa	**SQL**	Prolog, Datalog	Describir el qué, no el cómo
Concurrente / actor	— (CSP en **Go**)	Erlang, Elixir	Procesos, mensajes, tolerancia a fallos
Array / científica	—	APL, J, Julia, Fortran, MATLAB	Operar sobre vectores completos
Móvil / moderno	—	Swift, Dart	Apps nativas y multiplataforma
Históricos / shell	—	BASIC, Bash, PowerShell	Contexto histórico y automatización
Vivos en producción	—	**COBOL, Fortran, Ada, RPG, PL/I, MUMPS, Smalltalk, Lisp, AutoLISP, Tcl, Perl, Delphi, VBA, Pascal, JCL, Assembler**	**Lenguajes antiguos que siguen ejecutándose hoy — ficha propia cada uno**

📇 Fichas de lenguaje

Cada cápsula resume **historia** (autor, año, motivo, de quién hereda), **características** (paradigma, tipos, memoria y ejecución), **con qué se ejecuta** (compilador o intérprete y gestor de paquetes), el mapa **"si ya sabes X..."** hacia un lenguaje del núcleo, y su **estado** actual.

> 🗂️ ***Y cada lenguaje tiene además su ficha completa.*** *Las cápsulas de abajo son el resumen; la ficha desarrolla la* ***historia****, dónde vive hoy,* ***lo que enseña y no enseña ningún otro****, lo que se ha modernizado, cómo se ejecuta con órdenes reales,* ***el programa de la clase 041 explicado línea a línea*** *y la bibliografía.*
>
> 👉 ***Índice de las 60 fichas de lenguaje*** *— incluidos los* ***diez del núcleo*** *(Python, JavaScript, TypeScript, Java, C#, Go, Rust, C, SQL, PHP), que antes no la tenían.*

🧱 Familia C / llaves

C++

> 🗂️ ***Ficha completa de C++***

Bjarne Stroustrup empezó en 1979 en Bell Labs un proyecto llamado "C con clases", que en 1985 se publicó como **C++**. La meta era añadir abstracción orientada a objetos a C sin sacrificar su rendimiento ni su acceso al hardware. Hereda toda la sintaxis y el modelo de memoria de C, y suma clases, plantillas (programación genérica), sobrecarga de operadores, excepciones y una enorme biblioteca estándar (STL). Es **multiparadigma** (OO, genérico, procedimental y cada vez más funcional), de **tipado estático** y **compilado a código nativo**, con **gestión manual** de memoria apoyada hoy en punteros inteligentes (RAII) en lugar del `malloc` / `free` crudo. Se compila con **g++**, **clang++** o **MSVC**, y sus dependencias se gestionan con **vcpkg** o **Conan**. Si ya sabes **C**, C++ te sonará familiar en lo básico pero mucho más grande y con abstracciones de coste cero. Está **muy vivo**: motores de juego, navegadores, sistemas embebidos, HPC y finanzas. El estándar lo mantiene ISO (isocpp.org (https://isocpp.org/)).

Objective-C

> 🗂️ ***Ficha completa de Objective-C***

Brad Cox y Tom Love lo crearon a inicios de los años 80 (hacia 1984) fusionando el modelo de **mensajería de Smalltalk** con el lenguaje C. NeXT lo adoptó y, tras la compra por Apple, fue durante décadas el lenguaje oficial de macOS e iOS. Es un **superconjunto estricto de C**: cualquier programa C es válido, y encima se añade una capa OO dinámica con su característica sintaxis de corchetes `[objeto mensaje]`. Es **multiparadigma** con **tipado estático** (con dinamismo en el despacho de mensajes) y memoria gestionada por **conteo de referencias** (ARC, automático desde 2011). Se compila con **clang** dentro de **Xcode**; la gestión de dependencias histórica es **CocoaPods**. Si ya sabes **C** y algo de OO, reconocerás el núcleo, pero el envío dinámico de mensajes es su gran diferencia. Hoy está en **declive gestionado**: sigue vivo en bases de código Apple heredadas, pero Swift lo ha reemplazado como lenguaje recomendado (documentación de Apple (https://developer.apple.com/documentation/objectivec)).

Zig

> 🗂️ ***Ficha completa de Zig***

Andrew Kelley presentó Zig en 2016 como un lenguaje de sistemas moderno que pretende ser **"un mejor C"** sin la complejidad de C++. Su lema es la ausencia de comportamientos ocultos: no hay asignaciones de memoria implícitas, no hay flujo de control oculto y no hay preprocesador. Es **procedimental**, de **tipado estático** y **compilado a código nativo**, con **gestión manual** de memoria mediante *allocators* explícitos que se pasan como parámetros. Su rasgo estrella es `comptime`: ejecución de código en tiempo de compilación que sustituye a las macros y a los genéricos tradicionales. Trae **compilador propio** (que además puede compilar código C) y un **sistema de build y gestor de paquetes integrados** desde la propia herramienta `zig`. Si ya sabes **C**, Zig te resultará el pariente que arregla sus asperezas conservando el control total. Está en **crecimiento activo pero aún pre-1.0**, popular en tooling y proyectos de sistemas (ziglang.org (https://ziglang.org/)).

Nim

> 🗂️ ***Ficha completa de Nim***

Andreas Rumpf comenzó Nim en 2008 (bajo el nombre *Nimrod*) buscando un lenguaje tan expresivo como Python pero tan eficiente como C. Su sintaxis usa **indentación significativa** al estilo Python, pero **compila a C, C++ o JavaScript** y de ahí a un binario nativo veloz. Es **multiparadigma** (imperativo, OO y funcional), de **tipado estático con inferencia**, y ofrece un **recolector de basura configurable** (incluida la opción ORC, o gestión manual cuando hace falta). Destaca por su sistema de **macros higiénicas** que operan sobre el AST, dándole poder de metaprogramación. Se usa el **compilador** `nim` y el gestor de paquetes **Nimble**. Si ya sabes **Python** te enamorará la sintaxis, y si vienes de **C** apreciarás el rendimiento y el binario autocontenido. Es un lenguaje de **nicho pero maduro y estable**, con comunidad pequeña y dedicada (nim-lang.org (https://nim-lang.org/)).

D

> 🗂️ ***Ficha completa de D***

Walter Bright, autor del primer compilador nativo de C++, diseñó D en 2001 en Digital Mars como un rediseño de C++ que conservara su potencia eliminando su carga histórica. Es **multiparadigma** (imperativo, OO, funcional y metaprogramación), de **tipado estático con inferencia** y **compilado a nativo**. Su modelo de memoria es flexible: **recolección de basura por defecto**, pero permite gestión manual y un subconjunto `@nogc` para código de tiempo real. Brilla en metaprogramación con plantillas y ejecución en tiempo de compilación (CTFE). Cuenta con tres compiladores —**dmd** (referencia), **ldc** (backend LLVM) y **gdc** (backend GCC)— y el gestor de paquetes **dub**. Si ya sabes **C++**, D se siente como su versión ordenada y más rápida de escribir. Su estado es **vivo pero de nicho**: comunidad estable, uso puntual en la industria (dlang.org (https://dlang.org/)).

🌀 Familia Scripting dinámico

Ruby

> 🗂️ ***Ficha completa de Ruby***

Yukihiro "Matz" Matsumoto publicó Ruby en Japón en 1995, buscando un lenguaje **optimizado para la felicidad del programador**: elegante, consistente y agradable de leer. Es **totalmente orientado a objetos** (hasta los números son objetos), con fuerte influencia de Smalltalk, Perl y Lisp. Su tipado es **dinámico y de pato**, se **interpreta** (con compilación JIT en versiones recientes) y gestiona memoria con **recolector de basura**. Es célebre por sus bloques, su expresividad y por el framework web **Ruby on Rails**, que popularizó la "convención sobre configuración". Se ejecuta con el intérprete de referencia **CRuby/MRI**, y sus dependencias se manejan con **RubyGems** y **Bundler**. Si ya sabes **Python**, Ruby te resultará un primo cercano igual de dinámico pero más orientado a expresiones y con una cultura de DSLs muy marcada. Sigue **vivo**, sobre todo en desarrollo web y automatización (ruby-lang.org (https://www.ruby-lang.org/)).

Perl

> 🗂️ ***Ficha completa de Perl***

Larry Wall creó Perl en 1987 como una "navaja suiza" para procesar texto y administrar sistemas Unix, combinando ideas de C, sed, awk y los shells. Su filosofía es "hay más de una forma de hacerlo" (TMTOWTDI). Es **multiparadigma**, de **tipado dinámico** con las famosas variables de contexto (`$` escalar, `@` array, `%` hash), **interpretado** y con memoria gestionada por conteo de referencias. Su gran fortaleza son las **expresiones regulares**, tan integradas que definieron un estándar de facto (PCRE). Se ejecuta con el intérprete `perl` y su ecosistema histórico es **CPAN**, uno de los repositorios de módulos más grandes y antiguos. Si ya sabes **Python** o **PHP**, reconocerás el terreno del scripting, pero Perl es más denso y simbólico. Su estado es **legado activo**: dominó la web de los 90 y hoy persiste en administración de sistemas, bioinformática y scripts heredados (perl.org (https://www.perl.org/)).

Lua

> 🗂️ ***Ficha completa de Lua***

Roberto Ierusalimschy, Luiz Henrique de Figueiredo y Waldemar Celes crearon Lua en 1993 en la PUC-Rio de Brasil como un **lenguaje de scripting embebible**, ligero y portable. Su diseño prioriza un intérprete diminuto que se incrusta fácilmente dentro de aplicaciones escritas en C. Es **multiparadigma** apoyado en una única estructura de datos, la **tabla**, y en **tipado dinámico**; se **interpreta** (con la variante **LuaJIT** para máxima velocidad) y usa **recolector de basura**. No trae gran biblioteca estándar a propósito: su valor es ser el motor de scripting de otra cosa. Se ejecuta con el intérprete `lua` y el gestor de paquetes **LuaRocks**. Si ya sabes **JavaScript**, reconocerás las tablas como objetos y los prototipos. Está **muy vivo** como lenguaje incrustado: videojuegos (Roblox, World of Warcraft), Redis, Nginx y Neovim (lua.org (https://www.lua.org/)).

Tcl

> 🗂️ ***Ficha completa de Tcl***

John Ousterhout diseñó Tcl (*Tool Command Language*) en 1988 como un lenguaje de comandos simple y embebible para dar scripting a herramientas de ingeniería. Su principio radical es que **todo es una cadena de texto**: comandos, valores y estructuras se representan como strings, lo que hace su sintaxis minúscula y uniforme. Es **imperativo**, de **tipado dinámico** (con conversión perezosa según el uso), **interpretado** y con memoria automática. Su compañero histórico es **Tk**, un kit de widgets gráfico multiplataforma que también adoptaron Python y Perl. Se ejecuta con el intérprete `tclsh` (o `wish` para GUI), y sus bibliotecas se distribuyen vía **Tcllib** y el sistema de paquetes propio. Si ya sabes **Bash** o **Python**, encontrarás un scripting familiar pero con un modelo de "todo texto" peculiar. Su estado es **legado de nicho**: automatización de EDA, testing (Expect) y aplicaciones industriales de larga vida (tcl-lang.org (https://www.tcl-lang.org/)).

R

> 🗂️ ***Ficha completa de R***

Ross Ihaka y Robert Gentleman crearon R en 1993 en la Universidad de Auckland como una implementación libre del lenguaje **S** de Bell Labs, orientada a **estadística y análisis de datos**. Es **multiparadigma** con un fuerte sesgo funcional y **vectorizado**: las operaciones se aplican de forma natural a vectores y data frames completos. Su tipado es **dinámico**, se **interpreta** y gestiona memoria con **recolector de basura**. Su verdadera potencia está en el ecosistema: **CRAN**, con miles de paquetes estadísticos, y el conjunto **tidyverse** para manipulación y visualización de datos (ggplot2). Se ejecuta con el intérprete `R` y los paquetes se instalan

con `install.packages()` desde **CRAN**. Si ya sabes **Python** con pandas y numpy, verás un rival directo, pero R nació desde la estadística en vez de adaptarla. Está **muy vivo** en academia, ciencia de datos, bioestadística y análisis (r-project.org (https://www.r-project.org/)).

🧉 Familia JVM

Kotlin

> 🗂️ ***Ficha completa de Kotlin***

JetBrains anunció Kotlin en 2011 y lanzó la versión 1.0 en 2016 como un lenguaje moderno para la **JVM** que corrigiera las asperezas de Java manteniendo interoperabilidad total con él. Google lo adoptó como lenguaje preferente para Android en 2017. Es **multiparadigma** (OO y funcional), de **tipado estático con inferencia**, y destaca por su **seguridad frente a nulos** integrada en el sistema de tipos. Corre sobre la **JVM** con **recolector de basura**, pero también compila a **JavaScript** y a **binario nativo** (Kotlin/Native), lo que habilita proyectos multiplataforma. Se compila con **kotlinc** y se construye con **Gradle** o **Maven**, reutilizando todo el ecosistema de librerías de Java. Si ya sabes **Java**, Kotlin se siente como su versión concisa y segura; la migración es gradual porque conviven en el mismo proyecto. Está **muy vivo**, dominante en Android y creciente en backend (kotlinlang.org (https://kotlinlang.org/)).

Scala

> 🗂️ ***Ficha completa de Scala***

Martin Odersky creó Scala en 2004 en la EPFL para **fusionar programación orientada a objetos y funcional** de forma elegante sobre la JVM. Su nombre viene de "scalable language". Es **multiparadigma** con un sistema de tipos muy potente (tipos algebraicos, inferencia, *traits*, implicits/ `given`), **estático**, ejecutado sobre la **JVM** con **recolector de basura** (con backends a JS y nativo). Popularizó en el mundo JVM las funciones de orden superior, la coincidencia de patrones y la inmutabilidad por defecto; es el lenguaje de **Apache Spark**. Se compila con **scalac** y se construye con **sbt** o Maven. Si ya sabes **Java**, Scala te abre la puerta a la programación funcional tipada sin salir de la JVM, aunque su curva es más pronunciada. Su estado es **vivo pero maduro**: fuerte en big data, backend financiero y sistemas distribuidos (scala-lang.org (https://www.scala-lang.org/)).

Groovy

> 🗂️ ***Ficha completa de Groovy***

James Strachan propuso Groovy en 2003 (versión 1.0 en 2007) como un lenguaje **dinámico para la JVM** con sintaxis cercana a Java pero mucho más ágil, inspirado en Ruby y Python. Es **multiparadigma**, con **tipado dinámico opcionalmente estático**, ejecutado sobre la **JVM** con **recolector de basura**. Reduce la ceremonia de Java (menos punto y coma, menos tipos explícitos) y brilla creando **DSLs**, lo que lo hizo el lenguaje de scripting de **Gradle** y de la automatización en **Jenkins**. Es casi un superconjunto de Java: pegar código Java suele funcionar. Se ejecuta con `groovy` (o se compila con `groovyc`) y aprovecha dependencias vía **Grape** o Gradle/Maven. Si ya sabes **Java**, Groovy es la manera más rápida de scriptear sobre la misma plataforma sin recompilar todo. Su estado es **vivo de nicho**: build scripts, pipelines de CI y testing (Spock) (groovy-lang.org (https://groovy-lang.org/)).

Clojure

> 🗂️ ***Ficha completa de Clojure***

Rich Hickey creó Clojure en 2007 como un **Lisp moderno para la JVM**, centrado en la **inmutabilidad** y la programación funcional práctica. Es un dialecto de Lisp: código **homoicónico** escrito con paréntesis, con macros potentes. Su tipado es **dinámico**, corre sobre la **JVM** con **recolector de basura** (también existe ClojureScript hacia JavaScript), y sus estructuras de datos son **persistentes e inmutables**. Su modelo de concurrencia —basado en STM, átomos y agentes— busca eludir los problemas del estado mutable compartido. Se ejecuta como código sobre la JVM y se gestiona con **Leiningen** o **tools.deps** (`deps.edn`), con acceso a todo el ecosistema Java. Si vienes de **Java**, el salto conceptual es grande (de OO mutable a datos funcionales), pero comparten plataforma y librerías. Está **vivo de nicho**: backend de datos, fintech y equipos que valoran el enfoque funcional (clojure.org (https://clojure.org/)).

🟦 Familia .NET

F#

> 🗂️ ***Ficha completa de F#***

Don Syme y Microsoft Research desarrollaron F#, lanzado en 2005, para llevar la tradición **ML** al ecosistema **.NET**. Es un lenguaje **funcional primero** (aunque multiparadigma) con **tipado estático e inferencia fuerte**, ejecutado sobre el **CLR** con **recolector de basura**. Ofrece tipos algebraicos, coincidencia de patrones, inmutabilidad por defecto y *computation expressions*, manteniendo interoperabilidad total con las librerías de .NET. Su indentación es significativa, al estilo de OCaml, del que hereda directamente. Se compila con `fsc` dentro de las herramientas `dotnet` y usa **NuGet** como gestor de paquetes. Si ya sabes **C#**, F# es la puerta a la programación funcional tipada sin cambiar de plataforma: comparten runtime y bibliotecas, pero el estilo es muy distinto (expresiones e inmutabilidad frente a OO imperativo). Está **vivo de nicho**: finanzas, análisis de datos y equipos .NET que buscan robustez funcional (fsharp.org (https://fsharp.org/)).

VB.NET

> 🗂️ ***Ficha completa de VB.NET***

Microsoft lanzó Visual Basic .NET en 2002 como la reencarnación del clásico Visual Basic (1991) sobre la plataforma **.NET**, rompiendo la compatibilidad con el VB6 anterior a cambio de un modelo OO completo. Su seña es una **sintaxis verbosa en inglés** (`If … Then … End If`) pensada para ser legible y accesible a principiantes. Es **multiparadigma** con **tipado estático** (con opción de enlace tardío), ejecutado sobre el **CLR** con **recolector de basura**; es funcionalmente equivalente a C# porque comparten el mismo runtime y biblioteca base. Se compila con `vbc` vía las herramientas `dotnet` y usa **NuGet**. Si ya sabes **C#**, VB.NET es literalmente el mismo modelo con otra sintaxis más prosaica. Su estado es **legado en mantenimiento**: Microsoft ya no le añade características de lenguaje nuevas, pero sigue soportado para las muchas aplicaciones empresariales existentes (documentación de Visual Basic (https://learn.microsoft.com/dotnet/visual-basic/)).

λ Familia Funcional tipada (ML)

Haskell

> 🗂️ ***Ficha completa de Haskell***

Haskell nació en 1990 del trabajo de un comité académico que buscaba unificar los lenguajes funcionales perezosos en un estándar común; lleva el nombre del lógico **Haskell Curry**. Es el lenguaje funcional puro por excelencia: **evaluación perezosa** por defecto, **funciones puras** y efectos secundarios controlados mediante **mónadas**. Su tipado es **estático con inferencia Hindley-Milner** y un sistema de clases de tipos muy expresivo; se **compila a nativo** con **recolector de basura**. Es célebre por permitir razonamiento matemático sobre los programas y por su influencia en casi todos los lenguajes modernos (incluidas las características funcionales de Rust, Scala y Swift). Se compila con **GHC** y se gestiona con **Cabal** o **Stack**. Si ya sabes las partes funcionales de **Rust** o de otro lenguaje tipado, Haskell las lleva al extremo purista. Su estado es **vivo de nicho**: investigación, compiladores, fintech y verificación (haskell.org (https://www.haskell.org/)).

OCaml

> 🗂️ ***Ficha completa de OCaml***

OCaml procede del linaje **ML** desarrollado en el INRIA francés; la versión con objetos data de 1996 (Caml existía desde finales de los 80). Combina programación **funcional, imperativa y orientada a objetos** con un énfasis en la practicidad y la velocidad de compilación. Su tipado es **estático con inferencia** y tipos algebraicos con coincidencia de patrones; **compila a código nativo** muy eficiente (o a bytecode) y usa **recolector de basura**. Es conocido por su solidez y por ser la base de proyectos como el compilador de Rust en sus inicios, la herramienta de facto en verificación y el lenguaje de trading de Jane Street. Se compila con **ocamlopt/ocamlc** y se gestiona con **opam** (paquetes) y **dune** (build). Si ya sabes **Rust**, reconocerás las enumeraciones, el pattern matching y la inferencia: Rust bebió directamente de aquí. Está **vivo de nicho**: finanzas, compiladores y herramientas formales (ocaml.org (https://ocaml.org/)).

Elm

> 🗂️ ***Ficha completa de Elm***

Evan Czaplicki presentó Elm en 2012 como su tesis: un lenguaje **funcional puro dedicado a interfaces web** que compila a JavaScript y promete **cero excepciones en tiempo de ejecución**. Es funcional al estilo ML —**tipado estático con inferencia**, tipos algebraicos, inmutabilidad total— pero con un diseño deliberadamente pequeño y amable con los principiantes. Introdujo **la Arquitectura Elm** (modelo, actualización, vista), un patrón unidireccional que inspiró a Redux en el mundo React. No hay efectos secundarios sueltos: todo pasa por un runtime controlado, y su compilador es famoso por mensajes de error extraordinariamente claros. Se usa el **compilador** `elm`, que trae su propio gestor de paquetes (`elm install`). Si ya sabes **JavaScript** o **TypeScript**, Elm te ofrece garantías mucho más fuertes a cambio de un ecosistema más cerrado. Su estado es **vivo de nicho**, con desarrollo pausado pero comunidad fiel (elm-lang.org (https://elm-lang.org/)).

🔬 Familia Lisp

Scheme

> 🗂️ ***Ficha completa de Scheme***

Guy Steele y Gerald Sussman crearon Scheme en 1975 en el MIT como un dialecto **minimalista y elegante de Lisp**, diseñado para ser conceptualmente limpio y fácil de razonar. Su filosofía es dar pocas primitivas muy poderosas en lugar de muchas características. Es **funcional primero** (con soporte imperativo), de **tipado dinámico**, **homoicónico** (código y datos comparten forma) y con **recolección de basura**. Fue pionero en tratar las funciones como ciudadanos de primera clase con alcance léxico y en formalizar las **continuaciones** y la **recursión de cola**. Durante décadas fue el lenguaje del célebre libro y curso *SICP*. Tiene muchas implementaciones (Chez Scheme, Guile, Racket, MIT/GNU Scheme), cada una con su propio empaquetado. Si ya sabes cualquier lenguaje del núcleo, Scheme te enseña los fundamentos desde la raíz. Su estado es **vivo de nicho**: enseñanza, investigación y scripting embebido (GNU Guile) (sitio de R7RS (https://standards.scheme.org/)).

Racket

> 🗂️ ***Ficha completa de Racket***

Racket comenzó en 1995 como *PLT Scheme* y se renombró en 2010; es un descendiente de Scheme convertido en **plataforma para crear lenguajes**. Su gran idea es que construir lenguajes de dominio específico es una actividad de primera clase: incluye herramientas para definir tu propia sintaxis y semántica. Es **multiparadigma** (funcional, OO, lógico según el módulo), de **tipado dinámico** (con un dialecto tipado, Typed Racket), **homoicónico** y con **recolección de basura**. Trae un entorno completo, **DrRacket**, muy usado en educación. Se ejecuta con `racket` y se gestiona con `raco pkg`. Si ya sabes **Python** como lenguaje de propósito general, Racket ofrece una experiencia comparable de "baterías incluidas" pero con la potencia de macros de Lisp. Está **vivo de nicho**: enseñanza de programación (libro *How to Design Programs*), investigación en lenguajes y prototipado (racket-lang.org (https://racket-lang.org/)).

Common Lisp

> 🗂️ ***Ficha completa de Common Lisp***

Common Lisp surgió a mediados de los 80 (estandarizado por ANSI en 1994) para **unificar la dispersa familia de dialectos Lisp** en un estándar industrial común. Es un Lisp grande y pragmático, **multiparadigma**, con un potente sistema de objetos (**CLOS**) que incluye herencia múltiple y métodos multi-despacho. Su tipado es **dinámico** (con anotaciones opcionales), **homoicónico** con un sistema de macros muy potente, y usa **recolección de basura**. Su rasgo distintivo es la **imagen viva**: se desarrolla modificando un sistema en ejecución, con recompilación incremental de funciones. Implementaciones comunes son **SBCL** (compila a nativo) y **CCL**, y las librerías se obtienen con **Quicklisp**. Si vienes de cualquier lenguaje del núcleo, Common Lisp representa un paradigma interactivo y maleable distinto a todos. Su estado es **legado vivo de nicho**: sistemas expertos, aviación (antes), y una comunidad pequeña muy productiva (common-lisp.net (https://common-lisp.net/)).

Emacs Lisp

> 🗂️ ***Ficha completa de Emacs Lisp***

Richard Stallman escribió Emacs Lisp en 1985 como el lenguaje de extensión del editor **GNU Emacs**. Es un dialecto de Lisp diseñado no para programas autónomos, sino para **configurar y programar el propio editor**: casi toda la funcionalidad de Emacs está escrita en él y puede redefinirse en caliente. Es **multiparadigma**, de **tipado dinámico**, **interpretado** (con compilación a bytecode y, desde 2021, compilación nativa vía libgccjit) y con **recolección de basura**. No tiene gestor de paquetes independiente: los paquetes se instalan **dentro de Emacs** mediante `package.el` desde archivos como **MELPA**. Si ya sabes algo de **Scheme** o Lisp, reconocerás el modelo; si no, es el ejemplo vivo de una aplicación completamente programable por su usuario. Su estado es **vivo** mientras Emacs lo esté: sigue siendo el corazón de uno de los editores más longevos y extensibles (manual de Emacs Lisp (https://www.gnu.org/software/emacs/manual/eintr.html)).

🔗 Familia Lógica y declarativa

Prolog

> 🗂️ ***Ficha completa de Prolog***

Alain Colmerauer y Philippe Roussel crearon Prolog en 1972 en Marsella como el lenguaje insignia de la **programación lógica**. En lugar de describir pasos, el programador declara **hechos y reglas**, y el motor busca soluciones mediante **unificación** y **backtracking** automático. Es **declarativo**, de **tipado dinámico**, **interpretado** (o compilado a bytecode) y con memoria automática. Un programa es esencialmente una base de conocimiento sobre la que se hacen consultas; el orden de ejecución lo decide el resolutor, no una secuencia de instrucciones. Fue central en la IA simbólica y en el proyecto japonés de "quinta generación". Implementaciones habituales son **SWI-Prolog** y **GNU Prolog**, con un sistema de paquetes propio (`pack` en SWI). Si ya sabes **SQL**, reconocerás el enfoque declarativo de "describir el qué"; Prolog lo extiende a inferencia lógica general. Está **vivo de nicho**: procesamiento de lenguaje, razonamiento y enseñanza de IA (swi-prolog.org (https://www.swi-prolog.org/)).

Datalog

> 🗂️ ***Ficha completa de Datalog***

Datalog surgió en la comunidad de bases de datos deductivas a finales de los años 70 y 80 como un **subconjunto de Prolog** deliberadamente restringido para garantizar **terminación y consultas decidibles**. Elimina las funciones y limita la recursión de forma que toda consulta finaliza, lo que lo hace ideal como lenguaje de consulta declarativo sobre grandes conjuntos de hechos. Es **declarativo** y lógico: se definen relaciones mediante reglas, y el motor calcula el cierre de todo lo derivable (evaluación *bottom-up*). No es un único producto sino una familia de dialectos integrados en herramientas: **Soufflé** (análisis estático de programas), motores de bases de datos como **Datomic** y sistemas de análisis. Si ya sabes **SQL**, Datalog te resultará familiar como lenguaje declarativo, pero maneja con naturalidad la **recursión** (grafos, jerarquías) que en SQL es engorrosa. Su estado es **vivo de nicho** y en auge dentro de análisis de código y razonamiento sobre datos (Soufflé, un motor Datalog (https://souffle-lang.github.io/)).

📨 Familia Concurrente / actor

Erlang

> 🗂️ ***Ficha completa de Erlang***

Joe Armstrong, Robert Virding y Mike Williams crearon Erlang en Ericsson en 1986 (liberado como software libre en 1998) para construir **sistemas de telecomunicaciones** con alta disponibilidad y tolerancia a fallos. Su modelo es el de **actores**: procesos ligerísimos y aislados que solo se comunican por **paso de mensajes**, sin memoria compartida. Es **funcional**, de **tipado dinámico**, ejecutado sobre la máquina virtual **BEAM** con **recolección de basura por proceso**. Su filosofía distintiva es "**deja que falle**": los procesos que fallan son supervisados y reiniciados en vez de programarse a la defensiva, y el código puede actualizarse **en caliente** sin detener el sistema. Se ejecuta con `erl` (BEAM) y se construye con **rebar3**. Si ya conoces la concurrencia CSP de **Go**, Erlang ofrece un modelo emparentado pero con actores aislados y supervisión. Está **vivo**: mensajería (WhatsApp), telecomunicaciones y sistemas de alta disponibilidad (erlang.org (https://www.erlang.org/)).

Elixir

> 🗂️ ***Ficha completa de Elixir***

José Valim creó Elixir en 2011 para llevar la **robustez de la plataforma Erlang/BEAM** a una sintaxis moderna y productiva inspirada en Ruby. Compila a la máquina virtual **BEAM** y hereda todo su modelo: **actores** ligeros, paso de mensajes, supervisión y tolerancia a fallos. Es **funcional**, de **tipado dinámico**, con **recolección de basura** e inmutabilidad por defecto. Añade sobre Erlang herramientas de primera clase: **macros** para metaprogramación, tuberías (`|>`) y una excelente experiencia de desarrollo. Su framework web **Phoenix** (con LiveView) lo hizo popular para aplicaciones en tiempo real. Se ejecuta con `elixir` sobre BEAM y se gestiona con **Mix** y el repositorio de paquetes **Hex**. Si ya sabes **Ruby**, la sintaxis te encantará; si vienes de **Go**, reconocerás el foco en concurrencia. Está **vivo y en crecimiento**: web en tiempo real, sistemas distribuidos y streaming (elixir-lang.org (https://elixir-lang.org/)).

🔢 Familia Array / científica

APL

> 🗂️ ***Ficha completa de APL***

Kenneth Iverson desarrolló APL (*A Programming Language*) en IBM; su notación es de 1962 y la implementación ejecutable llegó hacia 1966. Es el pionero de la **programación de arrays**: las operaciones se aplican a **matrices y vectores completos** de una vez, sin bucles explícitos. Su rasgo más famoso es un **conjunto de símbolos especiales** (`⍴`, `⍳`, `≠`…) que expresan operaciones complejas en muy pocos caracteres, lo que produce programas extraordinariamente concisos. Es **funcional y de arrays**, de **tipado dinámico**, **interpretado** y con memoria automática. Requiere teclado o mapeo especial para sus glifos. La implementación de referencia moderna es **Dyalog APL** (comercial), y existe **GNU APL** libre. Si ya sabes trabajar vectorizado (como con numpy en **Python**), APL es la raíz filosófica de todo eso, llevada al extremo. Su estado es **legado vivo de nicho**: finanzas cuantitativas y actuariales (dyalog.com (https://www.dyalog.com/)).

J

> 🗂️ ***Ficha completa de J***

Kenneth Iverson (autor de APL) y Roger Hui diseñaron J en 1990 como sucesor de APL que usara únicamente **caracteres ASCII** en lugar de los símbolos especiales, para portabilidad. Conserva la esencia de la **programación de arrays** —operar sobre datos multidimensionales completos— pero reemplaza los glifos por combinaciones de signos de puntuación (`+/`, `i.`, `#`). Es **funcional y de arrays**, de **tipado dinámico**, **interpretado** y con memoria automática, y añade una potente **programación tácita** (definir funciones combinando otras sin nombrar los argumentos). Se ejecuta con el intérprete `jconsole` y tiene su propio sistema de addons. Si ya sabes el estilo vectorizado de **Python** con numpy, J te muestra hasta dónde llega esa idea, aunque su sintaxis es críptica al principio. Su estado es **nicho vivo**: análisis matemático, finanzas y programación recreativa, con una comunidad pequeña y entusiasta (jsoftware.com (https://www.jsoftware.com/)).

Julia

> 🗂️ ***Ficha completa de Julia***

Jeff Bezanson, Stefan Karpinski, Viral Shah y Alan Edelman presentaron Julia en 2012 (desde el MIT) para resolver el "problema de los dos lenguajes": tener la **facilidad de Python** y la **velocidad de C** en uno solo, orientado a la **computación científica**. Es **multiparadigma** con un modelo central de **despacho múltiple**, **tipado dinámico con inferencia** y **compilación JIT** vía LLVM, lo que le da rendimiento cercano al nativo. Trae operaciones vectorizadas, aritmética de alto nivel y excelente interoperabilidad con C, Fortran y Python. Se ejecuta con `julia` y su gestor de paquetes integrado, **Pkg**, muy cuidado. Si ya sabes **Python** para ciencia de datos, Julia ofrece una sintaxis parecida pero sin el cuello de botella de rendimiento que obliga a reescribir en C. Está **vivo y en crecimiento**: computación numérica, machine learning, investigación y HPC (julialang.org (https://julialang.org/)).

Fortran

> 🗂️ ***Ficha completa de Fortran***

John Backus dirigió en IBM la creación de Fortran (*Formula Translation*) en 1957, el **primer lenguaje de alto nivel** ampliamente usado y compilado. Nació para expresar **fórmulas matemáticas** de cálculo científico de forma legible en vez de en ensamblador. Es **imperativo y procedimental** (con módulos y programación orientada a objetos añadidos en estándares modernos), de **tipado estático**, **compilado a nativo** y con **gestión de memoria** manual/estática pensada para rendimiento máximo. Sigue siendo referencia en **álgebra lineal y cálculo numérico** (BLAS, LAPACK están escritos en Fortran) por sus arreglos multidimensionales eficientes y su optimización madura. Se compila con **gfortran** o **ifx**, y ya cuenta con un gestor de paquetes moderno, **fpm**. Si ya sabes **C**, reconocerás la compilación a nativo, pero Fortran es más orientado a arreglos y menos a punteros. Su estado es **legado muy vivo**: supercomputación, clima, física e ingeniería (fortran-lang.org (https://fortran-lang.org/)).

MATLAB

> 🗂️ ***Ficha completa de MATLAB***

Cleve Moler creó MATLAB (*Matrix Laboratory*) a finales de los años 70 como interfaz sencilla a las librerías de álgebra lineal LINPACK/EISPACK, y en 1984 se fundó **MathWorks** para comercializarlo. Su unidad fundamental es la **matriz**, y todo el lenguaje gira en torno a operar sobre arreglos completos. Es **imperativo y vectorizado**, de **tipado dinámico**, **interpretado** (con JIT interno) y memoria automática. Su fuerza no es solo el lenguaje sino el **entorno integrado**: editor, depurador, visualización y sobre todo los **toolboxes** especializados (control, señales, imágenes) que dominan la ingeniería. Es **software comercial y propietario**, con extensiones desde su Add-On Explorer; la alternativa libre más cercana es **GNU Octave**. Si ya sabes **Python** con numpy, verás un competidor directo enfocado a ingeniería. Su estado es **muy vivo** en la industria y la academia de ingeniería, biomédica y control (mathworks.com (https://www.mathworks.com/products/matlab.html)).

📱 Familia Móvil / moderno

Swift

> 🗂️ ***Ficha completa de Swift***

Apple presentó Swift en 2014, liderado por Chris Lattner (creador de LLVM), como el **sucesor de Objective-C** para desarrollar en todo su ecosistema, con una sintaxis moderna, segura y agradable. Es **multiparadigma** (OO, funcional y protocolar), de **tipado estático con inferencia**, con **seguridad frente a nulos** mediante *optionals* y coincidencia de patrones. **Compila a nativo** vía LLVM y gestiona memoria con **conteo automático de referencias (ARC)**, sin recolector de basura en pausa. Se volvió de **código abierto** en 2015 y hoy corre también en Linux y Windows. Se compila con **swiftc** y se gestiona con el **Swift Package Manager**. Si ya sabes **TypeScript** o **Rust**, reconocerás los optionals, la inferencia y los enums con datos asociados; Swift mezcla esa seguridad con una ergonomía muy pulida. Está **muy vivo**: es el lenguaje principal de iOS, macOS y demás plataformas Apple, además de algo de backend (swift.org (https://www.swift.org/)).

Dart

> 🗂️ ***Ficha completa de Dart***

Lars Bak y Kasper Lund crearon Dart en Google en 2011, inicialmente como alternativa a JavaScript para la web. Encontró su verdadero propósito con **Flutter**, el framework de UI multiplataforma que lo convirtió en la elección para apps móviles, web y escritorio desde una sola base de código. Es **orientado a objetos** con clases, de **tipado estático con inferencia** (y seguridad frente a nulos), y tiene un modelo de ejecución dual: **compilación JIT** durante el desarrollo (recarga en caliente) y **compilación AOT a nativo** para producción, además de transpilar a **JavaScript**. Usa **recolección de basura**. Se ejecuta con la herramienta `dart` y el gestor de paquetes **pub** (pub.dev). Si ya sabes **JavaScript** o **TypeScript**, Dart te resultará muy familiar en sintaxis pero más estructurado y con tipos sólidos. Está **muy vivo**, impulsado por la adopción de Flutter (dart.dev (https://dart.dev/)).

🗄️ Familia Históricos / shell

COBOL

> 🗂️ ***Ficha completa de COBOL***

COBOL (*Common Business-Oriented Language*) fue definido en 1959 por el comité CODASYL, con fuerte influencia del trabajo de **Grace Hopper**, para estandarizar el **procesamiento de datos de negocio** en un lenguaje legible por gestores. Su sintaxis es **verbosa y parecida al inglés** (`ADD TAX TO PRICE GIVING TOTAL`), pensada para claridad administrativa más que para concisión. Es **imperativo y procedimental**, de **tipado estático** con precisión decimal exacta (crucial en finanzas), **compilado** y con memoria estática. Domina la lógica de **bancos, seguros y gobiernos** desde hace más de sesenta años, procesando volúmenes enormes de transacciones en mainframes. Se compila con **IBM Enterprise COBOL** en entornos corporativos o con el libre **GnuCOBOL**. No se parece a ningún lenguaje del núcleo en estilo, pero conceptualmente su tratamiento de registros y datos anticipa a las estructuras y a **SQL**. Su estado es **legado crítico y muy vivo**: sostiene infraestructura financiera esencial y hay demanda de mantenimiento (sitio de GnuCOBOL (https://gnucobol.sourceforge.io/)).

Pascal

> 🗂️ ***Ficha completa de Pascal***

Niklaus Wirth diseñó Pascal en 1970 como un lenguaje **para enseñar buena programación estructurada**, claro y disciplinado, en reacción a la permisividad de otros lenguajes de la época. Es **imperativo y procedimental**, de **tipado estático fuerte**, **compilado a nativo** y con gestión de memoria manual mediante punteros. Introdujo o popularizó estructuras limpias (`begin` / `end`, tipos definidos por el usuario, records) que influyeron en generaciones de lenguajes. Su descendiente comercial **Object Pascal / Delphi** añadió orientación a objetos y dominó el desarrollo rápido de aplicaciones Windows en los 90. Hoy se compila sobre todo con el libre **Free Pascal (`fpc`)**, con Delphi vivo en el ámbito comercial. Si ya sabes **C**, reconocerás el modelo compilado y los punteros, pero Pascal es más estricto y legible. Su estado es **legado de nicho**: enseñanza histórica, mantenimiento de aplicaciones Delphi y algunos proyectos nuevos con Free Pascal/Lazarus (freepascal.org (https://www.freepascal.org/)).

BASIC

> 🗂️ ***Ficha completa de BASIC***

John Kemeny y Thomas Kurtz crearon BASIC en el Dartmouth College en 1964 con un objetivo explícito: un lenguaje **fácil para principiantes** de cualquier disciplina, no solo para ingenieros. Es **imperativo y procedimental**, de **tipado dinámico o débil** según el dialecto, e históricamente **interpretado** línea a línea, lo que lo hacía inmediato para experimentar. Con la microinformática de los 80 (con sus versiones con números de línea y `GOTO`) fue el primer contacto con la programación para millones de personas en los ordenadores domésticos. Evolucionó en muchos dialectos: **Visual Basic**, **VBA** (dentro de Office), **FreeBASIC** y **QB64**. No es un único producto ni tiene un gestor de paquetes común. Si ya sabes **Python**, reconocerás la vocación de accesibilidad, pero BASIC es mucho más antiguo y fragmentado. Su estado es **legado**: histórico en su forma clásica, pero vivo en VBA para automatización de ofimática (documentación de VBA (https://learn.microsoft.com/office/vba/api/overview/)).

Bash

> 🗂️ ***Ficha completa de Bash***

Brian Fox escribió Bash (*Bourne Again SHell*) en 1989 para el proyecto GNU como reemplazo libre del shell **Bourne** (`sh`) de Unix. Es a la vez el **intérprete de comandos interactivo** por defecto en la mayoría de sistemas Linux y macOS (durante años) y un **lenguaje de scripting** para automatizar tareas del sistema. Es **imperativo**, de **tipado dinámico basado en cadenas** (todo es texto), **interpretado** y orientado a **encadenar programas** mediante tuberías y redirecciones. No compila ni tiene gestor de paquetes propio: su fuerza es orquestar las utilidades del sistema operativo. Se ejecuta con el intérprete `bash`. Si ya sabes cualquier lenguaje del núcleo, Bash es la pegatina que conecta procesos, archivos y programas en la línea de comandos, con una sintaxis peculiar y llena de trampas de comillas. Su estado es **omnipresente y muy vivo**: automatización, CI/CD, despliegues y administración de sistemas (manual de GNU Bash (https://www.gnu.org/software/bash/manual/)).

PowerShell

> 🗂️ ***Ficha completa de PowerShell***

Microsoft lanzó PowerShell en 2006, diseñado por Jeffrey Snover, como un shell de automatización moderno para Windows y, desde 2016 en su versión **multiplataforma y de código abierto**, también para Linux y macOS. Su idea rompedora frente a los shells Unix es que la tubería transporta **objetos .NET**, no texto plano: los comandos (*cmdlets*, con nombres `Verbo-Sustantivo`) pasan datos estructurados que se pueden filtrar por propiedades sin analizar cadenas. Es **imperativo y orientado a objetos**, de **tipado dinámico** apoyado en el sistema de tipos del **CLR**, **interpretado**, con acceso completo a la plataforma .NET. Se ejecuta con `pwsh` (o `powershell` en Windows heredado) y sus módulos se instalan desde la **PowerShell Gallery** con `Install-Module`. Si ya sabes **Bash**, reconocerás la vocación de automatización pero con un paradigma de objetos en lugar de texto; y si sabes **C#**, verás la plataforma .NET por debajo. Está **muy vivo** en administración de Windows, la nube y DevOps (documentación de PowerShell (https://learn.microsoft.com/powershell/)).

Por qué esta capa existe

Meter 40 lenguajes con implementación completa multiplicaría el mantenimiento sin multiplicar el aprendizaje: **muchos son casi el mismo lenguaje con otra piel**. El Atlas captura esa amplitud por comprensión; el núcleo captura la profundidad por práctica verificada.

Mantenimiento de las fichas

Estas cápsulas se **revisan periódicamente** (última revisión: 2026-07). Los hechos históricos son estables, pero las herramientas, versiones y gestores de paquetes cambian: ante cualquier detalle operativo, la documentación oficial enlazada manda. El **núcleo** de 10 lenguajes —Python, JavaScript, TypeScript, Java, C#, Go, Rust, C, SQL y PHP— no se resume aquí porque **se estudia e implementa a fondo en las clases** (índice completo).

Fichas de lenguaje

🎬 ActionScript — 1998

ActionScript es el único lenguaje de este Atlas que **está muerto de verdad**, y por eso merece una ficha: **su historia es la mejor lección sobre dependencia de plataforma que este curso puede ofrecer**. Durante una década movió la mitad de la web interactiva; el 31 de diciembre de 2020 dejó de ejecutarse en cualquier navegador del mundo, en un mismo día.

> 🎯 ***Por qué está en este programa***
>
> *ActionScript es un **primo de la familia JavaScript / web** (Atlas) —de hecho, **es un dialecto de ECMAScript**, hermano directo de JavaScript—.*
>
> *Está aquí por una razón que no es técnica: **es el caso de estudio de qué pasa cuando el lenguaje depende de un tiempo de ejecución propietario** (clases 162 y 164). PL/I y RPG enseñan lo mismo en su versión lenta; ActionScript lo enseña en su versión brusca. Y ActionScript 3 es, además, **la implementación más completa que existió del abandonado ECMAScript 4**.*

Año	1998 (AS1); **AS2** en 2003; **AS3** en 2006; **fin del Flash Player** el 31-12-2020
Autoría	**Gary Grossman**, Macromedia; después Adobe
Familia	JavaScript / web — implementación de **ECMAScript 4**, que nunca se aprobó
Paradigma	Orientado a objetos con clases, dirigido por eventos
Tipado	AS1/AS2: dinámico. **AS3: estático opcional**, con comprobación en compilación
Memoria	Recolección de basura, con conteo de referencias y marcado
Ejecución	Bytecode sobre la **AVM2** del Flash Player, con JIT
Estado	🔴 **Extinto en la web**; sobrevive en Adobe AIR y en emuladores

---

📜 Historia

Flash empezó en 1996 como una herramienta de animación vectorial. En **1998**, Macromedia le añadió un lenguaje de guion —**ActionScript**— para que los botones hicieran algo.

Y entonces pasó lo que nadie previó: **Flash se convirtió en la plataforma de aplicaciones de la web**. Entre 2000 y 2010, **cuando el navegador no podía hacer casi nada** —sin `canvas`, sin vídeo nativo, sin audio, con un JavaScript lento y con diferencias enormes entre navegadores—, Flash sí podía. Y por eso llevaba dentro:

- **El vídeo de YouTube**, en sus primeros años.
- **Los juegos web**: Newgrounds, Kongregate, Miniclip, FarmVille — un ecosistema enorme.
- **Aplicaciones de empresa** con Flex, que competían con las de escritorio.
- **Y la publicidad y las animaciones** de media Internet.

ActionScript 3 (2006) fue un salto técnico serio: **tipado estático opcional, clases de verdad, paquetes, excepciones y una máquina virtual nueva con JIT** que multiplicó el rendimiento por diez. Estaba basado en la propuesta **ECMAScript 4**, la modernización de JavaScript que se estaba negociando entonces.

Y ES4 se abandonó en 2008. JavaScript siguió por otro camino —el que llevó a ES5 y a ES6—, así que **ActionScript 3 quedó como la única implementación completa de un estándar que nunca existió**: un lenguaje huérfano, primo de JavaScript pero incompatible.

El final tiene fecha y carta: en **abril de 2010, Steve Jobs publicó *Thoughts on Flash***, explicando por qué el iPhone nunca lo admitiría —batería, rendimiento, seguridad y, sobre todo, **no depender de una capa propietaria de terceros**—. La web abierta alcanzó a Flash con HTML5, `canvas`, vídeo nativo y WebGL.

Adobe anunció el fin en 2017, y **el 31 de diciembre de 2020 el reproductor dejó de funcionar**, con un bloqueo programado en el propio software.

🏭 Dónde vive hoy

- **Adobe AIR**, hoy mantenido por HARMAN: aplicaciones de escritorio y móviles heredadas siguen ejecutándose.
- **Apache Royale**: el sucesor libre de Flex, que **compila ActionScript a JavaScript**.
- **Ruffle**: un emulador de Flash **escrito en Rust y compilado a WebAssembly**, que permite volver a ejecutar juegos y animaciones antiguos en el navegador.
- **Proyectos de preservación**: Flashpoint ha archivado más de 100.000 juegos y animaciones.

> ***Ruffle merece la mención como cierre de la historia: la tecnología que sustituyó a Flash es la que hoy lo resucita.*** *Y esa es una lección de la clase 162 —* ***un objetivo de compilación abierto y estándar sobrevive a las plataformas propietarias.***

🧠 Lo que enseña: la dependencia de plataforma

El día que Flash murió, murió todo lo escrito en ActionScript. No hubo migración, ni capa de compatibilidad, ni versión de mantenimiento: **el intérprete se apagó**.

Y merece contrastarlo con los demás lenguajes de este Atlas:

Situación	Qué pasó
COBOL, 66 años	sigue ejecutándose; hay compiladores libres y comerciales
Pascal, 55 años	Free Pascal es libre y llega a plataformas nuevas (clase 162)
Smalltalk, 46 años	SqueakJS ejecuta imágenes de 1978 en el navegador
PL/I, 62 años	vivo, y **sin implementación libre**: no llega a plataformas nuevas
ActionScript, 22 años	**extinto**: dependía de un reproductor de una sola empresa

Y el patrón es el de la clase 164: la supervivencia depende de que exista una implementación libre y de que el formato sea abierto. ActionScript no tenía ninguna de las dos.

Y la segunda lección, más fina, es sobre estándares: **AS3 implementó una propuesta que no llegó a ser estándar**. Apostar a una especificación en discusión es un riesgo real, y aquí costó la compatibilidad con el lenguaje del que venía.

🔁 Lo que quedó

- **La AVM2 y su bytecode** influyeron en el diseño de máquinas virtuales posteriores.
- **El modelo de eventos con burbujeo y captura** de Flash está en el DOM actual (clase 120).
- **Las corrutinas y la línea de tiempo** de Flash anticiparon patrones de animación que hoy están en CSS y en los marcos declarativos.
- **Y la lección de gobernanza**, que es lo que esta ficha aporta al curso.

⚙️ Cómo se ejecuta hoy

```
# Apache Royale: ActionScript compilado a JavaScript
mxmlc Venta.as -output venta.swf         # el compilador clásico (Flex SDK)
asjsc Venta.as                            # Royale: a JavaScript

# Y para ver contenido antiguo:
#   Ruffle (WebAssembly), en el navegador o como aplicación de escritorio
```

🍼 El programa de la clase 041 en ActionScript

Como en SQL y en Elm, aquí el contrato **está adaptado** (clase 040): **el reproductor Flash no tiene entrada estándar**, así que se ilustra el cálculo.

```
// ActionScript corre en el reproductor Flash, sin stdin: se ilustra el cálculo.
package {
    public class Venta {
        public static function total(precio:Number, cantidad:Number, descuento:Number):String {
            var t:Number = precio * cantidad * (1 - descuento);
            return "Total: " + t.toFixed(2);
        }
    }
}
```

Lo que hay que ver.

- `precio:Number` **es la anotación de tipo**, con **dos puntos**, igual que TypeScript — que llegó **catorce años después**. No es coincidencia: los dos descienden de la misma propuesta ES4.
- `Number` **es el mismo tipo único de coma flotante** de JavaScript, y `toFixed(2)` es literalmente la misma función. **La familia se reconoce sin esfuerzo**, que es la tesis del Atlas.
- `package` **y** `public class` vienen de Java, y son lo que ES4 quería añadir a JavaScript y no se aprobó.
- **Y la ausencia de entrada estándar no es un detalle del ejemplo**: es la marca de un lenguaje atado a un entorno concreto, que es exactamente lo que esta ficha cuenta.

📚 Fuentes y bibliografía

- Apache Royale (https://royale.apache.org/) — el sucesor libre; documentación de migración desde Flex.
- Ruffle (https://ruffle.rs/) — el emulador en Rust y WebAssembly; su código es interesante como caso de la clase 162.
- **Steve Jobs**, *Thoughts on Flash* (2010) — archivado; una página que conviene leer como documento de decisión tecnológica (clase 175).
- **Colin Moock**, *Essential ActionScript 3.0*, O'Reilly — la referencia del lenguaje en su momento.
- Flashpoint Archive (https://flashpointarchive.org/) — el proyecto de preservación; un ejemplo de qué hace falta para que el software sobreviva a su plataforma (clase 154).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: JavaScript · TypeScript · Dart · Delphi

✈️ Ada — 1983

El lenguaje que no puede fallar. Si vuelas, si tomas un metro automático o si un satélite mantiene su órbita, hay una probabilidad razonable de que Ada esté ejecutándose. Es un nicho pequeño y es, seguramente, el nicho con las consecuencias más graves.

> 🎯 ***Por qué está en este programa***

> ***Criterio de inclusión: Ada se ejecuta hoy en aviónica, espacio, ferrocarril y defensa**, con revisión del estándar en 2022 y un ecosistema activo (GNAT, Alire, SPARK). Es un nicho pequeño, pero es el nicho donde un fallo se cuenta en vidas.*
>
> *Entra porque **deja a la vista un concepto que el núcleo esconde**: que **el tipo puede llevar encima una regla del dominio**, no solo un tamaño de máquina. En casi todo el núcleo, un descuento es un `float` y "entre 0 y 1" es un comentario o un `if` que alguien recordó escribir. En Ada, `range 0.0 .. 1.0` forma parte del tipo y el programa se niega a continuar si se viola. Es la versión llevada al extremo de lo que Rust hace con la nulabilidad y TypeScript con los tipos literales — y verla en Ada aclara de dónde viene esa idea.*

Año	1983 (MIL-STD-1815); revisiones 95, 2005, 2012, 2022
Autoría	Equipo de **Jean Ichbiah** en CII Honeywell Bull, por encargo del DoD de EE. UU.
Familia	Pascal / ALGOL, rama de sistemas críticos
Paradigma	Imperativo, modular, concurrente y OO (desde Ada 95)
Tipado	Estático, **fuerte y nominal**, con subtipos y rangos comprobados
Memoria	Pila y montículo con control explícito; sin recolector obligatorio
Ejecución	Compilado a nativo, con comprobaciones en tiempo de ejecución activables
Estado	🟢 **Nicho pequeño, importancia extrema** — aviónica, defensa, espacio, ferrocarril

---

📜 Historia

A mediados de los 70 el Departamento de Defensa de Estados Unidos hizo un inventario y encontró que sus sistemas embebidos usaban **más de 450 lenguajes y dialectos distintos**. Cada proveedor traía el suyo; nada era reutilizable, nada era auditable y formar gente era carísimo. En 1975 se creó un grupo de trabajo para definir un lenguaje único, y en 1977 se convocó un concurso internacional con propuestas anónimas identificadas por colores. En 1979 ganó la propuesta **Verde**, del equipo de **Jean Ichbiah**.

El lenguaje se llamó **Ada** en honor a **Ada Lovelace**, y su estándar militar recibió el número **MIL-STD-1815**: 1815 es el año de nacimiento de Lovelace. Es probablemente el número de norma más deliberado de la historia de la informática.

La evolución fue quitando rigidez sin quitar garantías. **Ada 95** fue el primer lenguaje orientado a objetos con estándar ISO. **Ada 2005** añadió interfaces y mejoras de tiempo real. **Ada 2012** introdujo lo que hoy es su sello: los **contratos** —precondiciones, postcondiciones e invariantes de tipo— escritos como parte de la declaración y comprobables por el compilador o en ejecución. **Ada 2022** es la revisión vigente.

En paralelo creció **SPARK**, un subconjunto de Ada con anotaciones que permite **demostrar matemáticamente** la ausencia de errores en tiempo de ejecución: no que los tests pasen, sino que ciertos fallos son imposibles. Es de las pocas tecnologías de verificación formal que se usa de verdad en industria.

🏭 Dónde sobrevive hoy

- **Aviónica**: sistemas de control de vuelo y gestión de misión, certificados bajo **DO-178C**.
- **Espacio**: software de a bordo de satélites y lanzadores; la Agencia Espacial Europea lo ha usado extensamente.
- **Ferrocarril y metro**: señalización y control automático de trenes bajo **EN 50128**.
- **Defensa**: sistemas de armas, radar, mando y control.

- **Gestión del tráfico aéreo** y sistemas industriales de alta integridad.

🧠 Por qué no ha muerto

1. El sistema de tipos convierte errores de dominio en errores de compilación. Ada permite declarar tipos con **rango**, y comprobarlos. Esto no es un comentario ni un `assert`:

```
type Altitud_Pies  is range 0 .. 60_000;
type Velocidad_Nudos is range 0 .. 900;

A : Altitud_Pies  := 35_000;
V : Velocidad_Nudos := 450;
-- A := A + V;  --  ERROR DE COMPILACIÓN: son tipos distintos
```

Dos enteros con la misma representación pero distinto significado **no son intercambiables**. Sumar pies a nudos no compila. En C, en Python o en JavaScript, esa suma es perfectamente válida y perfectamente absurda. La pérdida de la sonda **Mars Climate Orbiter** en 1999 fue exactamente ese error —libras-fuerza contra newtons— en un sistema que no era Ada.

2. Las comprobaciones son parte del lenguaje, no de la disciplina del equipo. Índices fuera de rango, desbordamiento, desreferencia nula: Ada los detecta y lanza una excepción en lugar de producir comportamiento indefinido. Se pueden desactivar (`pragma Suppress`) cuando el rendimiento lo exija, pero **la decisión es explícita y queda escrita**.

3. La concurrencia está en el lenguaje. Las **tareas** (`task`), las **citas** (*rendezvous*) y los **objetos protegidos** son construcciones del lenguaje, no de una biblioteca. El perfil **Ravenscar** define un subconjunto de concurrencia con comportamiento temporal analizable, que es lo que permite certificar un sistema de tiempo real: no basta con que funcione, hay que poder demostrar que siempre cumple sus plazos.

4. Legibilidad como requisito de certificación. El diseño prioriza al lector sobre el escritor —`begin` / `end` con nombre, sin abreviaturas crípticas— porque el código de un avión lo audita gente que no lo escribió, años después.

5. Reescribirlo requeriría recertificar. Y la certificación de un sistema crítico cuesta más que el desarrollo.

> ***La lección de Ariane 5.*** *El fallo del vuelo 501 en 1996 se cita a veces como un fallo de Ada. Fue lo contrario: una conversión de un real de 64 bits a un entero de 16 desbordó, Ada* ***detectó*** *el desbordamiento y lanzó la excepción prevista. El defecto fue de ingeniería de sistemas —se reutilizó código de Ariane 4 con las comprobaciones desactivadas por rendimiento, en un cohete con una trayectoria distinta— y de gestión del fallo. El lenguaje avisó; el diseño no supo qué hacer con el aviso.*

🔄 Lo que se ha modernizado

- **Contratos en el lenguaje** (Ada 2012, ampliados en 2022): `with Pre =>`, `with Post =>`, `Type_Invariant` y `Subtype_Predicate`. La especificación de una función deja de ser un comentario y se convierte en algo que el compilador comprueba —o que **GNATprove demuestra**— antes de ejecutar nada.
- **SPARK como herramienta industrial.** La verificación formal dejó de ser un ejercicio académico: hoy se aplica a componentes concretos de sistemas reales, con niveles graduales (desde "no hay errores de ejecución" hasta "cumple la especificación funcional"). Es de las pocas tecnologías de demostración formal con uso comercial sostenido.

- **Alire (`alr`)**, un gestor de paquetes moderno al estilo de `cargo`, que resolvió el mayor problema práctico del lenguaje: la dificultad de compartir y reutilizar bibliotecas.
- **Ada 2022** añadió literales definidos por el usuario, expresiones `declare`, imágenes de tipo personalizables y paralelismo ligero.
- **Objetivos actuales**: **RISC-V**, ARM Cortex-M y microcontroladores; **GNAT-LLVM** amplía las arquitecturas soportadas; existen bibliotecas para programar placas de bajo consumo.
- **La conversación con Rust.** La discusión pública sobre *memory safety* —las recomendaciones de agencias de ciberseguridad sobre lenguajes seguros por diseño— ha traído a Ada/SPARK de vuelta a la mesa, porque llevaba cuarenta años resolviendo el mismo problema con otras herramientas.

⚙️ Cómo se ejecuta hoy

```
# GNAT, el compilador Ada de GCC
sudo apt-get install -y gnat

# El nombre del fichero DEBE coincidir con el de la unidad: total_venta.adb
gnatmake total_venta.adb
echo "15000 2 0.10" | ./total_venta
# Total: 27000.00
```

Herramientas. GNAT (AdaCore, basado en GCC) es el compilador dominante, con edición libre (GNAT FSF, en las distribuciones) y comercial (GNAT Pro, con soporte para certificación). `gprbuild` es el sistema de construcción por proyectos, y **Alire** (https://alire.ada.dev/) (`alr`) es el gestor de paquetes moderno, equivalente a `cargo`. Para verificación formal, **GNATprove** sobre **SPARK**.

🍼 El programa de la clase 041 en Ada

```
with Ada.Text_IO;            use Ada.Text_IO;
with Ada.Long_Float_Text_IO; use Ada.Long_Float_Text_IO;

procedure Total_Venta is

   --  El tipo dice lo que el negocio permite, no solo lo que la máquina guarda.
   subtype Descuento_T is Long_Float range 0.0 .. 1.0;

   Precio, Cantidad : Long_Float;
   Descuento        : Descuento_T;
   Total            : Long_Float;

begin
   Get (Precio);
   Get (Cantidad);
   Get (Descuento);          --  un 1.5 aquí levanta Constraint_Error

   Total := Precio * Cantidad * (1.0 - Descuento);

   Put ("Total: ");
   Put (Total, Fore => 1, Aft => 2, Exp => 0);
   New_Line;
end Total_Venta;
```

Recorrido, línea a línea.

- `with` importa una unidad; `use` hace visibles sus nombres sin cualificar. Se separan a propósito: puedes importar sin contaminar el espacio de nombres, y en código de alta integridad es común escribir `Ada.Text_IO.Put_Line (...)` completo y prescindir del `use`.

- `subtype Descuento_T is Long_Float range 0.0 .. 1.0` es la línea que resume el lenguaje. No define un tipo nuevo incompatible, sino un **subtipo** con una restricción: cualquier asignación fuera de `[0.0, 1.0]` levanta `Constraint_Error` en el punto exacto en que se produce. La regla de negocio "un descuento es una fracción" deja de ser un comentario y pasa a ser comprobable.
- `Get` está sobrecargado por tipo: al declarar `Descuento` como `Descuento_T`, la lectura ya valida. No hay un `if descuento > 1` en ninguna parte, y aun así el caso está cubierto.
- `Long_Float` es el real de doble precisión. `Float` sería simple precisión y, con siete dígitos significativos, empezaría a dar sorpresas en totales grandes.
- `Put (Total, Fore => 1, Aft => 2, Exp => 0)` formatea sin `printf`: `Fore` son los dígitos mínimos antes del punto, `Aft` los de después, y `Exp => 0` suprime la notación exponencial. La sintaxis `Nombre => Valor` es la **asociación por nombre**, disponible en cualquier llamada de Ada; hace que el sitio de llamada se lea solo, sin ir a buscar la firma.
- El `procedure` de nivel superior es el punto de entrada, y el fichero debe llamarse igual en minúsculas: `total_venta.adb`. GNAT usa esa convención para localizar unidades.

Compáralo con la versión en C de la misma clase: el mismo cálculo, pero aquí el rango del descuento está **en el tipo**, y ni el compilador ni el lector tienen que confiar en que alguien lo validó.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En Ada es…
`int x = 5;`	`X : Integer := 5;` — el nombre primero, `:=` para asignar
`==`	`=` (comparación); `:=` es la asignación, como en Pascal
`struct`	`record`
`enum`	`type Color is (Rojo, Verde, Azul);` — un tipo de verdad, no un entero
`typedef` de C	`subtype` (compatible) frente a `type` (incompatible a propósito)
Genéricos / plantillas	`generic` — instanciados explícitamente, sin sorpresas
`throw` / `try`	`raise` / `exception when ... =>`
Hilos y `mutex`	`task` y `protected` — en el lenguaje, no en una librería
`assert`	`with Pre => ...`, `with Post => ...`, `with Type_Invariant => ...`
`1000000`	`1_000_000` — el subrayado como separador, idea que Ada popularizó

⚠️ Errores comunes al leerlo

- **Confundir `type` y `subtype`.** `type Metros is new Float` crea un tipo **incompatible** con `Float`: no se pueden mezclar sin conversión explícita, y eso es la mitad del valor de Ada. `subtype` solo restringe y sigue siendo compatible.
- **Creer que las comprobaciones cuestan siempre.** El compilador elimina las que puede demostrar innecesarias. Lo que queda se paga solo donde hace falta, y se puede suprimir explícitamente.
- **Buscar recolector de basura.** No lo hay por defecto. La memoria dinámica se gestiona con tipos de acceso y, en sistemas críticos, muchas veces se **prohíbe** por completo: si no hay asignación dinámica, no hay fragmentación ni fallos de memoria imprevisibles.
- **Leer la verbosidad como burocracia.** `end Total_Venta;` repite el nombre porque, en un fichero de 2000 líneas que revisa un auditor, saber qué se está cerrando importa más que ahorrar teclas.
- **Ignorar mayúsculas y minúsculas.** Ada **no** distingue entre ellas; `Total`, `total` y `TOTAL` son el mismo identificador. La convención `Palabra_Con_Guiones` es estilo, no sintaxis.

📚 Fuentes y bibliografía

- Ada Resource Association (https://www.adaic.org/) — el punto de entrada oficial, con el estándar y su fundamento (*Rationale*).
- AdaCore Learn (https://learn.adacore.com/) — cursos interactivos gratuitos de Ada y SPARK, con compilador en el navegador.
- Alire (https://alire.ada.dev/) — el gestor de paquetes y el ecosistema moderno.
- **John Barnes**, *Programming in Ada 2012 with a Preview of Ada 2022*, Cambridge University Press — la referencia canónica; Barnes lleva desde Ada 83 explicando el lenguaje.
- **John McCormick, Frank Singhoff, Jérôme Hugues**, *Building Parallel, Embedded, and Real-Time Applications with Ada*, Cambridge — el libro para la parte de concurrencia y tiempo real.
- **John McCormick, Peter Chapin**, *Building High Integrity Applications with SPARK*, Cambridge — verificación formal aplicada, no teórica.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Pascal · Delphi / Object Pascal · Fortran

🔣 APL — 1966

APL es el lenguaje más raro de este Atlas y uno de los más influyentes: **usa un alfabeto propio de símbolos matemáticos**, necesitaba un teclado especial, y **cada símbolo es una operación sobre arreglos completos**. Un programa APL de una línea puede hacer lo que en otro lenguaje son treinta —y esa densidad es a la vez su virtud y la razón de su marginalidad.

> 🎯 ***Por qué está en este programa***
>
> *APL es un **primo de la familia array / científica** (Atlas), que no tiene representante en el núcleo — su influencia llega al curso por R, NumPy, Julia, MATLAB y el Fortran moderno.*
>
> *Aporta al programa **la programación por arreglos en su forma original y más pura** (clase 089): el pensamiento sin bucles, con operadores que se combinan. Es el antepasado directo de todo lo vectorizado que se hace hoy.*

Año	1966 (implementado); la notación es de **1957-1962**
Autoría	**Kenneth E. Iverson**, Harvard e IBM — **Premio Turing 1979**
Familia	Array / científica
Paradigma	**Funcional y por arreglos**, con operadores de orden superior
Tipado	Dinámico; **todo es un arreglo** —un escalar es un arreglo de rango 0—
Memoria	Recolección de basura
Ejecución	Interpretado, con primitivas muy optimizadas
Estado	🟡 **Vivo en nichos**: finanzas, actuarial y seguros; comunidad pequeña y activa

---

📜 Historia

Kenneth Iverson no diseñó APL como lenguaje de programación: lo diseñó **como notación matemática para describir algoritmos**, y lo publicó en 1962 en un libro cuyo título es literalmente *A Programming Language* — de donde sale el acrónimo.

Durante años fue **una notación en papel** para enseñar y para describir arquitecturas. **IBM la usó para especificar formalmente el System/360**, antes de que existiera un intérprete.

En **1966**, IBM implementó **APL\360**, y ocurrió algo inesperado: el sistema de tiempo compartido con APL se convirtió en una herramienta de análisis de negocio. **Actuarios, financieros y analistas —no programadores— empezaron a usarlo directamente**, porque una operación sobre una tabla entera se escribía en una línea.

Iverson recibió el Premio Turing en 1979, y su conferencia —*Notation as a Tool of Thought*— es uno de los mejores textos que se han escrito sobre por qué la notación importa: **una buena notación no solo expresa las ideas, ayuda a tenerlas**.

Después Iverson diseñó **J** (1990), que conserva la semántica y **sustituye los símbolos por ASCII** — resolviendo el problema del teclado a costa de la legibilidad.

🏭 Dónde vive hoy

- **Servicios financieros y seguros**: modelos actuariales, valoración de carteras y sistemas de riesgo, sobre todo con **Dyalog APL**.
- **Análisis de datos** en organizaciones que lo adoptaron y nunca se fueron.
- **Kdb+/q** (https://kx.com/), un descendiente de APL que es **el estándar de facto en datos de series temporales financieras de alta frecuencia** — probablemente el uso comercial más valioso de la familia.
- **Y como comunidad viva**: Dyalog, GNU APL y APL de código abierto tienen usuarios activos y competiciones anuales.

🧠 Lo que enseña: notación como herramienta de pensamiento

El ejemplo clásico. **Esto calcula la media de un vector:**

```
(+/x) ÷ ≢x
```

Y se lee: **la suma acumulada (`+/`) de `x`, dividida entre la cuenta (`≢`) de `x`.**

Y este es el ejemplo más famoso — **la criba de Eratóstenes en una línea**:

```
(~R∈R∘.×R)/R←1↓⍳N
```

Dos ideas hacen posible esa densidad, y las dos están en todos los lenguajes vectorizados de hoy:

Una, los operadores actúan sobre estructuras completas. No hay bucles porque no hacen falta: `+/` es la reducción, `∘.×` es el producto exterior, `⍳` genera un rango.

Y dos, los operadores se combinan. `/` no es una función: es un **operador de orden superior** que recibe una función y devuelve otra. `+/` es "reducir con suma", `×/` es "reducir con producto", `⌈/` es "el máximo". **Es `fold` de la clase 115**, veinte años antes de que la programación funcional lo popularizara.

> ***Y el precio hay que decirlo, porque define su historia**: **APL es de escritura rápida y de lectura lentísima**. Un programa denso es difícil de revisar (clase 146), y su fama de "lenguaje de solo escritura" —merecida a medias— junto con **la necesidad de un teclado especial** lo dejaron fuera de la corriente principal.*
>
> ***Y aun así ganó**: la idea sobrevivió sin los símbolos. **Cuando alguien escribe** `np.sum(a * b, axis=0)` **o** `datos |> filtrar |> resumir`**, está pensando como Iverson** (clase 089).*

🔄 Lo que se ha modernizado

- **Dyalog APL**: la implementación comercial de referencia, con objetos, interfaz con .NET y Python, y hoy con licencia gratuita para uso personal y educativo.
- **Símbolos sin teclado especial**: los editores modernos insertan los caracteres con prefijos, y el teclado dejó de ser una barrera.
- **J** y **q/kdb+** (https://kx.com/) como descendientes en ASCII.
- **BQN y Uiua**: lenguajes de arreglos de diseño reciente, que revisan la notación con lo aprendido en sesenta años.
- **Y la vuelta de las ideas**: `einsum`, la difusión de formas (*broadcasting*) de NumPy y las operaciones de arreglo de Fortran y Julia son APL con otra ropa.

⚙️ Cómo se ejecuta hoy

```
apl --script main.apl          # GNU APL
dyalog -script main.apls        # Dyalog
# Y en el navegador: tryapl.org, sin instalar nada
```

🧪 El programa de la clase 041 en APL

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
⍝ Leer la línea, convertirla en tres números y calcular el total
v ← ⍎⍞                          ⍝ ⍞ lee una línea; ⍎ la evalúa como expresión APL
total ← (v[1] × v[2]) × 1 - v[3]
⍝ Y con la reducción, sin índices:
total ← (×/ 2↑v) × 1 - ⊃⌽v
⎕ ← 'Total: ', ⍕2 ⍕ total
```

Lo que hay que ver, y es lo más didáctico de la ficha.

- `×/ 2↑v` dice: **toma los dos primeros (`2↑`) y redúcelos con el producto (`×/`)**. Es `precio × cantidad` **sin nombrar ninguno de los dos** — la operación se define sobre la estructura, no sobre los elementos.
- `⊃⌽v` es "el primero (`⊃`) de la inversa (`⌽`)", es decir, **el último elemento**. En APL se combinan operaciones simples en lugar de tener una función `last`.
- **Los índices empiezan en 1** por defecto —y es configurable con `⎕IO`—, como R, Fortran, Julia y MATLAB.
- `⍝` **es el comentario**, y `⎕ ←` es la salida.
- **Y la comparación con las otras diecinueve versiones de la clase 041 es el contenido**: todas declaran tres variables; **APL declara un vector y opera sobre él**. Es exactamente el mismo reflejo que delata a R en su ficha, y viene del mismo sitio.

📚 Fuentes y bibliografía

- **Kenneth Iverson**, *Notation as a Tool of Thought* (Turing Award Lecture, 1979) — libre en línea; **léelo aunque no vayas a tocar APL nunca**: es de los mejores textos sobre notación y pensamiento.
- TryAPL (https://tryapl.org/) — probarlo en el navegador, con teclado en pantalla.
- Dyalog Documentation Centre (https://docs.dyalog.com/) — manuales y el libro *Mastering Dyalog APL* (libre en PDF).
- APL Wiki (https://aplwiki.com/) — historia, dialectos y comparativas.
- **Kenneth Iverson**, *A Programming Language* (1962) — el libro original, que era una notación antes que un lenguaje.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: J · R · Julia · MATLAB · Fortran

⚙️ Assembler — desde 1949

Nunca desapareció; se concentró. El ensamblador dejó de usarse para escribir aplicaciones y se replegó a las capas donde alguien tiene que hablarle a la máquina sin intermediarios: el arranque del sistema, el cambio de contexto del planificador, el firmware, la criptografía de tiempo constante y los núcleos vectoriales que hacen rápido a todo lo demás.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: se escribe ensamblador hoy, todos los días.*** *Cada núcleo de sistema operativo tiene su capa de arranque y de conmutación de tareas en ensamblador; cada biblioteca criptográfica seria (OpenSSL, BoringSSL) lleva rutinas escritas a mano por seguridad y rendimiento; cada microcontrolador tiene su vector de interrupciones; y en el mundo mainframe,* ***HLASM*** *sigue siendo un lenguaje de desarrollo activo en IBM Z.*
>
> *Entra porque es* ***el suelo de todo el programa****. La Parte 8 trata de cómo funcionan los lenguajes por dentro: qué es una llamada a función, qué es la pila, qué significa "paso por valor", por qué una variable local es más barata que una del montículo, qué hace realmente un* `for`*. Todas esas preguntas tienen* ***una sola respuesta definitiva****, y está aquí. Leer el ensamblador que genera tu compilador no es nostalgia: es la única forma de dejar de creer y empezar a saber.*

Año	Finales de los 40 — el primer ensamblador simbólico se atribuye al EDSAC (1949)
Autoría	No hay una: cada arquitectura define el suyo
Familia	Lenguajes de bajo nivel, correspondencia casi 1:1 con el código máquina
Paradigma	Imperativo puro: mover datos, operar, saltar
Tipado	**Ninguno.** Solo hay bytes; el significado lo pone la instrucción que los usa
Memoria	Totalmente manual: registros, pila y direcciones
Ejecución	Ensamblado a código máquina; ejecución directa por el procesador
Estado	🟢 **Vivo donde importa** — firmware, núcleos, criptografía, SIMD, embebidos, seguridad

---

📜 Historia

Los primeros ordenadores se programaban en **código máquina**: números escritos a mano, con las direcciones calculadas por el programador. Cambiar una instrucción en medio de un programa obligaba a recalcular todos los saltos.

El **ensamblador** resolvió eso con dos ideas simples y decisivas: **nombres simbólicos para las instrucciones** (`MOV` en vez de un opcode numérico) y, sobre todo, **etiquetas para las direcciones**, de modo que el programa se reubica solo. **Kathleen Booth** es reconocida como autora del primer lenguaje ensamblador junto a su trabajo en el ARC2 a finales de los 40, y **Maurice Wilkes** y su equipo desarrollaron el *Initial Orders* del EDSAC en 1949, uno de los primeros ensambladores funcionales.

Durante los años 50 y 60, el ensamblador **era** la programación. FORTRAN tuvo que demostrar que un compilador podía generar código comparable, y COBOL tuvo que demostrar que la legibilidad valía la pena. Cuando ambas cosas quedaron probadas, el ensamblador empezó a retirarse de la escritura de aplicaciones — sin desaparecer nunca de las capas bajas.

Un punto que conviene fijar: "ensamblador" **no es un lenguaje**, es una **categoría**. Hay tantos como arquitecturas, y no se parecen entre sí:

Arquitectura	Dónde vive	Rasgo
x86-64	PC, servidores, nube	Muy complejo (CISC), instrucciones de longitud variable, dos sintaxis rivales (Intel y AT&T)
ARM / AArch64	Móviles, Apple Silicon, embebidos, servidores	RISC, instrucciones de tamaño fijo, muchos registros
RISC-V	Embebidos, académico, industria creciente	RISC abierto y modular, sin licencia
z/Architecture (HLASM)	Mainframe IBM Z	Ensamblador empresarial con macros muy potentes
AVR / PIC / MSP430	Microcontroladores	Recursos mínimos, control directo de patillas
PTX / SASS	GPU NVIDIA	Ensamblador paralelo para miles de hilos

🏭 Dónde sobrevive hoy

- **Arranque y núcleo**: el código que se ejecuta antes de que exista una pila utilizable, la conmutación de contexto entre procesos, los manejadores de interrupción, las barreras de memoria. El núcleo de Linux contiene decenas de miles de líneas de ensamblador por arquitectura, y no se pueden escribir en C.
- **Criptografía**: las implementaciones de AES, ChaCha20, curvas elípticas o SHA en OpenSSL llevan rutinas en ensamblador, por dos razones: aprovechar instrucciones específicas del procesador (AES-NI) y garantizar **tiempo constante**, porque un compilador puede introducir un salto condicional que filtre información por canal lateral.
- **Núcleos vectoriales (SIMD)**: códecs de vídeo, procesamiento de imagen, álgebra lineal. **FFmpeg** y **dav1d** son ejemplos públicos con enormes cantidades de ensamblador optimizado a mano, con ganancias que el compilador no alcanza.
- **Sistemas embebidos y tiempo real**: microcontroladores con kilobytes de memoria, donde cada ciclo y cada byte cuentan.
- **Seguridad ofensiva y defensiva**: ingeniería inversa, análisis de malware, desarrollo de exploits, CTF. Aquí no se escribe ensamblador tanto como **se lee**, porque es lo único que hay.
- **Mainframe**: **HLASM** sobre z/OS, para rutinas de sistema y salidas de personalización.

- **Compiladores y máquinas virtuales**: los generadores de código de LLVM y de la JVM emiten ensamblador, y sus autores lo leen a diario.

🧠 Por qué no ha muerto

1. Hay cosas que ningún lenguaje de alto nivel puede expresar. No existe forma en C de decir "escribe este valor en el registro de control del procesador", "invalida la TLB" o "ejecuta esta instrucción atómica concreta". El código de arranque, el cambio de contexto y los manejadores de interrupción **tienen** que ser ensamblador.

2. El compilador es muy bueno, pero no lo sabe todo. En el 99 % de los casos, escribir C bien es más rápido que escribir ensamblador a mano. En el 1 % restante —transformaciones vectoriales con patrones de acceso concretos, bucles internos de un códec— la persona que conoce el procesador gana, y gana por factores, no por porcentajes.

3. Garantías que el compilador no ofrece. El tiempo constante en criptografía es el ejemplo canónico: el compilador tiene derecho a convertir tu operación sin ramas en un salto condicional, y eso abre un canal lateral. La única forma de garantizarlo es escribir las instrucciones.

4. Leerlo es una destreza imprescindible. Depurar un fallo de optimización, entender un volcado de memoria, analizar un binario sin fuentes, verificar que una mitigación de seguridad se aplicó de verdad: todo eso es lectura de ensamblador.

5. Es el único lugar donde las abstracciones se pueden comprobar. Cuando en la Parte 5 se afirma que "una llamada a función guarda la dirección de retorno en la pila", eso es una afirmación verificable: se compila, se mira el ensamblador y se ve.

🔄 Lo que se ha modernizado

- **Extensiones vectoriales continuas**: AVX-512 en x86-64, **SVE/SVE2** en ARM (de longitud vectorial variable), la extensión **V** de RISC-V. Cada generación de procesador añade instrucciones nuevas que alguien tiene que usar primero a mano.
- **Instrucciones para acelerar cosas concretas**: cifrado (AES-NI, SHA-NI), multiplicación de matrices para IA (AMX en Intel, las extensiones matriciales de ARM), operaciones atómicas más finas.
- **RISC-V**, una arquitectura **abierta y modular** diseñada en los 2010, con adopción industrial creciente. Es la primera ISA importante en décadas que se puede estudiar e implementar sin licencia, y ha revitalizado la enseñanza de la arquitectura de computadores.
- **Herramientas incomparablemente mejores**: **Compiler Explorer** (https://godbolt.org/) muestra en el navegador, en tiempo real y con colores, qué ensamblador genera tu código en decenas de compiladores y arquitecturas. Lo que antes exigía `objdump` y paciencia hoy es inmediato.
- **Ensamblador en línea con restricciones** (`asm` de GCC/Clang, `asm!` de Rust): permite insertar instrucciones concretas dentro de código de alto nivel, describiendo al compilador qué registros y qué memoria se tocan, para que siga optimizando alrededor con seguridad.
- **Verificación formal de rutinas críticas**: proyectos como HACL*/EverCrypt generan e incluso demuestran la corrección de código criptográfico de bajo nivel.

⚙️ Cómo se ejecuta hoy

```
# Ensamblar y enlazar (GNU as, sintaxis AT&T, x86-64 Linux)
gcc -no-pie total.s -o total
echo "15000 2 0.10" | ./total
# Total: 27000.00

# Ver el ensamblador que genera TU código C — el uso más frecuente hoy:
gcc -O2 -S -masm=intel programa.c -o programa.s

# Desensamblar un binario existente:
objdump -d -M intel ./programa | less
```

Herramientas: GAS (`as`, el ensamblador de GNU, sintaxis AT&T por defecto), **NASM** y **YASM** (sintaxis Intel, muy usados en proyectos multimedia), **MASM** (Microsoft), **LLVM-MC**. Para leer y analizar: `objdump`, **Ghidra** (libre, de la NSA), **IDA Pro**, **radare2/rizin** y **Compiler Explorer**.

Las dos sintaxis de x86, que confunden a todo el mundo:

```
AT&T (GNU as):   movq  $5, %rax     # destino a la DERECHA, $ para literales, % para registros
Intel (NASM):    mov   rax, 5       # destino a la IZQUIERDA, sin prefijos
```

Es el mismo código máquina escrito de dos maneras. Al leer documentación conviene comprobar siempre cuál se está usando.

🪠 El programa de la clase 041 en ensamblador x86-64

> ⚠️ ***Específico de una arquitectura, y declarado.*** *Este código es x86-64 con la ABI System V de Linux y sintaxis AT&T.* ***No es portable****: en ARM64, en RISC-V o en Windows sería otro programa. Esa dependencia es, precisamente, la característica que define al ensamblador.*

```
        .section .rodata
fmt_in:  .asciz  "%lf %lf %lf"
fmt_out: .asciz  "Total: %.2f\n"
uno:     .double 1.0

        .text
        .globl  main
main:
        pushq   %rbp                    # prólogo: guardar el marco del llamante
        movq    %rsp, %rbp              # establecer el marco propio
        subq    $32, %rsp               # reservar 32 bytes en la pila (3 doubles + alineación)

        # --- scanf("%lf %lf %lf", &precio, &cantidad, &descuento) ---
        leaq    fmt_in(%rip), %rdi      # 1.er argumento: el formato
        leaq    -8(%rbp),  %rsi         # 2.º: &precio
        leaq    -16(%rbp), %rdx         # 3.º: &cantidad
        leaq    -24(%rbp), %rcx         # 4.º: &descuento
        xorl    %eax, %eax              # 0 argumentos en registros vectoriales
        call    scanf@PLT

        # --- total = precio * cantidad * (1 - descuento) ---
        movsd   uno(%rip), %xmm0        # xmm0 = 1.0
        subsd   -24(%rbp), %xmm0        # xmm0 = 1.0 - descuento
        mulsd   -8(%rbp),  %xmm0        # xmm0 *= precio
        mulsd   -16(%rbp), %xmm0        # xmm0 *= cantidad

        # --- printf("Total: %.2f\n", total) ---
        leaq    fmt_out(%rip), %rdi     # 1.er argumento: el formato
        movl    $1, %eax                # 1 argumento en registros vectoriales (xmm0)
        call    printf@PLT

        xorl    %eax, %eax              # return 0
        leave                           # epílogo: deshacer el marco
        ret
```

Recorrido, y lo que enseña cada bloque.

- **No hay variables.** `precio`, `cantidad` y `descuento` no existen: hay tres huecos de 8 bytes en la pila, en `-8(%rbp)`, `-16(%rbp)` y `-24(%rbp)`. El nombre era una comodidad del compilador. Esto es lo que **es** una variable local: un desplazamiento respecto al puntero de marco.
- **`pushq %rbp` / `movq %rsp, %rbp` / `leave` es el marco de pila**, y aparece idéntico en el código generado por cualquier compilador. Cuando en la Parte 5 se habla de "el ámbito de una función" o de "la pila de llamadas", **esto** es la pila de llamadas.
- **`subq $32, %rsp` es reservar memoria local**, y por eso es gratis comparado con el montículo: una resta. Liberarla es `leave`. Ahí está, en dos instrucciones, la respuesta a por qué una variable local es más barata que una asignación dinámica.
- **La convención de llamada es un contrato.** En la ABI System V de x86-64, los argumentos enteros y punteros van en `%rdi, %rsi, %rdx, %rcx, %r8, %r9`, y los reales en `%xmm0` – `%xmm7`. Nadie lo comprueba: si te equivocas de registro, el programa lee basura. Esa convención es lo que hace posible que Rust llame a C, que Python cargue una biblioteca `.so` y que la Parte 10 tenga sentido. **La interoperabilidad entre lenguajes se define en este nivel, no en el sintáctico.**
- **`movl $1, %eax` antes de `printf` no es adorno.** En una función variádica, `%eax` debe contener **cuántos argumentos viajan en registros vectoriales**, porque `printf` lo necesita para recorrerlos. Olvidarlo suele producir un fallo de segmentación desconcertante. Es un detalle de la ABI que casi nadie conoce hasta que lo sufre.

- `subsd`, `mulsd`: **aritmética de doble precisión en registros SSE.** La `s` final es *scalar* (un valor por registro); las variantes `subpd` / `mulpd` operan sobre **varios valores a la vez**, y en esa diferencia de una letra está todo el paralelismo SIMD.
- `(%rip)` **es direccionamiento relativo al contador de programa**, lo que permite que el código sea reubicable. Es la base técnica de las bibliotecas compartidas y de las mitigaciones de seguridad como ASLR.
- `@PLT` indica que el símbolo se resuelve en tiempo de enlace dinámico, a través de la tabla de enlace de procedimientos. Es cómo funciona realmente cargar una biblioteca.
- **No hay tipos.** Los mismos 8 bytes se tratan como puntero con `leaq` y como real con `movsd`. El procesador no sabe qué son: lo decide la instrucción. Cuando en la Parte 3 se dice que "el tipo es una interpretación de los bits", esta es la demostración.

El uso real de esto no es escribirlo, es leerlo. Compila la versión en C de la clase 041 con `gcc -O2 -S` y compárala con este código: verás que el compilador elimina el marco de pila, mantiene los valores en registros y reordena las multiplicaciones. Ese ejercicio —tres minutos en Compiler Explorer (https://godbolt.org/)— enseña más sobre optimización que cualquier explicación.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En ensamblador es…
Variable local	Un desplazamiento en la pila: `-8(%rbp)`
Variable global	Una etiqueta en `.data` o `.bss`
Constante	Una etiqueta en `.rodata`
`x = y`	`mov` — copiar bytes de un sitio a otro
`x + y`	`add` (enteros) o `addsd` (reales de doble precisión)
`if (a > b)`	`cmp` seguido de un salto condicional (`jg`, `jle`, …)
`while` / `for`	Una etiqueta y un salto hacia atrás
Llamada a función	`call` — apila la dirección de retorno y salta
`return`	`ret` — desapila la dirección y salta a ella
Argumentos	Registros según la ABI, y la pila a partir del séptimo
Valor devuelto	`%rax` (entero) o `%xmm0` (real)
`struct`	Un bloque de bytes; los campos son desplazamientos
Puntero	Una dirección; `leaq` la calcula, `movq (%reg)` la sigue

⚠️ Errores comunes al leerlo

- **Confundir las sintaxis.** `mov %rax, %rbx` mueve rax→rbx en AT&T y rbx→rax en Intel. La misma línea, sentido contrario. Comprueba siempre cuál estás leyendo.
- **Romper la alineación de la pila.** La ABI exige que `%rsp` esté alineado a 16 bytes en el momento del `call`. Si no lo está, una instrucción SSE dentro de `printf` provoca un fallo de segmentación aparentemente aleatorio. Es el error más frustrante para quien empieza.
- **Olvidar qué registros preserva quién.** La ABI divide los registros entre los que el llamado debe conservar (`%rbx`, `%rbp`, `%r12`–`%r15`) y los que puede destruir. Suponer lo contrario produce errores que solo aparecen a veces.
- **Leer código optimizado como si fuera el original.** Con `-O2`, el compilador reordena, fusiona y elimina. La correspondencia línea a línea con el fuente desaparece. Para estudiar, empieza con `-O0`.

- **Suponer portabilidad.** Este programa no funciona en un Mac con Apple Silicon ni en una Raspberry Pi. Es la naturaleza del lenguaje.
- **Creer que a mano siempre es más rápido.** Casi nunca lo es. Antes de escribir una sola instrucción, mide, y comprueba qué generó ya el compilador.

📚 Fuentes y bibliografía

- Compiler Explorer (godbolt.org) (https://godbolt.org/) — la herramienta con la que aprender a leer ensamblador. Empieza aquí.
- Intel 64 and IA-32 Architectures Software Developer's Manuals (https://www.intel.com/sdm) — la referencia completa de x86-64.
- Arm Architecture Reference Manual (https://developer.arm.com/documentation/ddi0487/latest/) — el equivalente para AArch64.
- The RISC-V Instruction Set Manual (https://riscv.org/technical/specifications/) — abierta y notablemente más legible que las anteriores; buena para aprender.
- IBM High Level Assembler (HLASM) (https://www.ibm.com/docs/en/hla-and-tf) — el ensamblador del mainframe, con su sistema de macros.
- **Randal Bryant, David O'Hallaron**, *Computer Systems: A Programmer's Perspective*, 3.ª ed., Pearson — **el libro** para entender qué hay debajo de tu código. Si solo lees uno de esta lista, que sea este.
- **Jonathan Bartlett**, *Programming from the Ground Up* — gratis en línea (https://savannah.nongnu.org/projects/pgubook/); enseña a programar empezando por el ensamblador.
- **Randall Hyde**, *The Art of 64-bit Assembly*, No Starch Press — el tratado moderno y exhaustivo.
- **David Patterson, John Hennessy**, *Computer Organization and Design (RISC-V Edition)*, Morgan Kaufmann — la arquitectura de computadores desde la ISA hacia arriba.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: C · C++ · Fortran

📐 AutoLISP — 1986

Probablemente el Lisp con más usuarios del mundo, y casi ninguno se considera programador. Arquitectos, ingenieros civiles, delineantes y proyectistas llevan cuarenta años automatizando AutoCAD con AutoLISP. Muchos estudios tienen miles de líneas de `.lsp` acumuladas que son, en la práctica, su ventaja competitiva.

> ***🎯 Por qué está en este programa***
>
> ***Criterio de inclusión: AutoLISP viene incluido en AutoCAD y Autodesk lo mantiene explícitamente*** *como el mecanismo de automatización del producto, con documentación actualizada en cada versión anual. No es un lenguaje que sobreviva en un rincón: se distribuye con uno de los programas de ingeniería más instalados del planeta.*
>
> *Entra porque* ***es el mejor ejemplo vivo de un concepto que el núcleo no toca****: el* ***lenguaje incrustado en una aplicación anfitriona****. AutoLISP no tiene* `main`*, no tiene fichero ejecutable y no sirve para nada fuera de AutoCAD: su "biblioteca estándar" son los objetos del dibujo. Es la misma relación que VBA tiene con Excel y que Lua tiene con un motor de videojuegos, y es el modelo de la Parte 10:* ***el lenguaje como capa de guion sobre un programa que ya existe****. Entenderlo cambia cómo se lee cualquier sistema de plugins.*

Año	1986, en AutoCAD Release 2.18
Autoría	**Autodesk**, sobre el intérprete **XLISP** de David Betz
Familia	Lisp — dialecto reducido y especializado
Paradigma	Funcional e imperativo, orientado a la manipulación del dibujo
Tipado	**Dinámico**, con los tipos del dibujo (puntos, nombres de entidad, conjuntos)
Memoria	Gestionada por el intérprete de AutoCAD
Ejecución	**Interpretado dentro de AutoCAD**; compilable a `.fas` / `.vlx`
Estado	🟢 **Muy vivo dentro de su nicho** — arquitectura, ingeniería, construcción, CAD

📜 Historia

En 1986 Autodesk necesitaba que AutoCAD fuera personalizable sin recompilarlo. La solución fue incrustar un intérprete de Lisp: se partió de **XLISP**, una implementación pequeña y libre de David Betz, y se le añadieron funciones para hablar con el dibujo. La elección de Lisp no fue casual — un intérprete de Lisp es notablemente pequeño y fácil de empotrar, la misma razón por la que Lua triunfó después en los videojuegos.

En 1997 Autodesk adquirió Vital LISP, de Basis Software, y lo integró como **Visual LISP**: un entorno de desarrollo dentro de AutoCAD (`VLIDE`), un compilador a ficheros `.fas` / `.vlx` para proteger el código, y —lo más importante— el acceso al **modelo de objetos ActiveX** de AutoCAD a través de las funciones `vla-*`. Eso duplicó de golpe lo que se podía hacer.

Durante casi cuarenta años Autodesk ha añadido lenguajes alternativos —VBA, ObjectARX en C++, .NET, más recientemente extensiones en JavaScript— y AutoLISP ha sobrevivido a todos. La razón es sociológica más que técnica: es el único que un ingeniero sin formación en programación aprende en una tarde y usa el mismo día.

🏭 Dónde sobrevive hoy

- **Estudios de arquitectura e ingeniería**: rutinas de acotado automático, generación de leyendas, numeración de elementos, cumplimiento de normas de capas y estilos.
- **Ingeniería civil y topografía**: importación de puntos, perfiles, cubicaciones.
- **Instalaciones y fabricación**: generación paramétrica de piezas, listas de materiales extraídas del dibujo.
- **Estandarización interna**: casi cualquier oficina técnica con muchos planos tiene un `acaddoc.lsp` que impone la plantilla de la casa.

Existen además dialectos compatibles en clones de AutoCAD (BricsCAD, ZWCAD, GstarCAD), lo que amplía su alcance más allá de Autodesk.

🧠 Por qué no ha muerto

1. Está donde está el dibujo. Una rutina de AutoLISP accede a la geometría, a las capas, a los bloques y a los atributos **directamente**, sin API externa, sin instalación y sin permisos de administrador. Ese último punto no es menor: en muchas empresas, un ingeniero puede cargar un `.lsp` pero no puede instalar software.

2. La curva de entrada es diminuta. Tres funciones — `setq`, `command` y `getpoint` — bastan para ser útil. Y como el intérprete está vivo dentro de AutoCAD, se prueba escribiendo en la línea de comandos y viendo el resultado en pantalla al instante. Es un REPL con salida gráfica.

3. Cuarenta años de código acumulado. Miles de estudios dependen de rutinas escritas por alguien que ya no está, que funcionan y que nadie va a reescribir.

4. Autodesk lo mantiene deliberadamente. Romper AutoLISP rompería a una parte sustancial de su base instalada.

🔄 Lo que se ha modernizado

Autodesk ha invertido en AutoLISP más recientemente de lo que la gente supone:

- **Extensión oficial para Visual Studio Code**, publicada y mantenida por Autodesk: resaltado, autocompletado, y **depuración paso a paso conectada a una sesión de AutoCAD en marcha**. Sustituye al viejo VLIDE, que apenas cambió en veinte años.
- **Unicode completo**, imprescindible para trabajar con planos en cualquier idioma.
- **Ampliación de plataforma**: Autodesk ha ido llevando el soporte de AutoLISP más allá de la versión completa de AutoCAD para escritorio, respondiendo a la presión de una base de usuarios que dependía de sus rutinas. Consulta la documentación de tu versión concreta, porque este punto ha cambiado varias veces.
- **Funciones `vl-*` y `vla-*`** de Visual LISP para manipulación de listas, cadenas, ficheros, registro del sistema y el modelo de objetos ActiveX completo — muy por encima del AutoLISP de 1986.
- **Convive con las alternativas**: .NET, ObjectARX (C++) y las extensiones en JavaScript cubren lo que AutoLISP no alcanza, y se llaman entre sí. Ese es el patrón realista: AutoLISP para la automatización cotidiana del delineante, .NET para el producto empaquetado.

⚙️ Cómo se ejecuta hoy

AutoLISP **solo** existe dentro de AutoCAD (o de un clon compatible). No hay intérprete de línea de comandos, no hay ejecutable, no hay CI posible.

```
; Cargar desde la línea de comandos de AutoCAD:
(load "C:/rutinas/totalventa.lsp")

; O con el cuadro de diálogo:  APPLOAD
; Y para que se cargue en cada dibujo: incluirlo en acaddoc.lsp
```

Ficheros del ecosistema: `.lsp` (fuente), `.fas` (compilado con Visual LISP), `.vlx` (aplicación empaquetada), `.mnl` (se carga automáticamente con el menú del mismo nombre), `acad.lsp` y `acaddoc.lsp` (carga automática al arrancar y por cada dibujo, respectivamente).

Herramientas: el editor integrado **VLIDE** (`VLISP` en la línea de comandos), y hoy también la extensión **AutoCAD AutoLISP Extension** para **Visual Studio Code**, que Autodesk publica y que permite editar y depurar contra una sesión de AutoCAD abierta.

🧪 El programa de la clase 041 en AutoLISP

> ⚠️ ***Contrato adaptado, y declarado.*** *AutoLISP no tiene `stdin` ni `stdout`: sus entradas son la línea de comandos de AutoCAD y el usuario, y su salida es el dibujo o la propia línea de comandos. El cálculo es el mismo; la forma de entrar y salir es la del anfitrión.* ***No se verifica en CI****: requiere una instalación de AutoCAD.*

```
;;; totalventa.lsp — clase 041 adaptada a AutoCAD
;;; El prefijo c: convierte la función en un COMANDO de AutoCAD.

(defun c:TOTALVENTA ( / precio cantidad descuento total )

  (setq precio    (getreal "\nPrecio unitario: "))
  (setq cantidad  (getreal "\nCantidad: "))
  (setq descuento (getreal "\nDescuento (0 a 1): "))

  (setq total (* precio cantidad (- 1.0 descuento)))

  (princ (strcat "\nTotal: " (rtos total 2 2)))
  (princ)
)
```

Recorrido, línea a línea.

- `(defun c:TOTALVENTA ...)` — el prefijo `c:` es la convención que convierte una función en un **comando de AutoCAD**: a partir de que se carga el fichero, el usuario escribe `TOTALVENTA` en la línea de comandos y se ejecuta. Sin ese prefijo sería una función normal, invocable solo desde otro código Lisp.
- **La barra en la lista de argumentos es la rareza que hay que conocer.** `( / precio cantidad ... )` significa: cero parámetros, y todo lo que va **detrás de la barra** son **variables locales**. Si se olvida la barra, esas variables quedan **globales** y contaminan la sesión de AutoCAD entera hasta que se cierre el dibujo. Es la primera causa de errores fantasma en AutoLISP: una rutina que funciona sola y falla cuando se ejecuta después de otra.
- `getreal` pide un número al usuario por la línea de comandos, con su mensaje. La familia completa — `getpoint`, `getdist`, `getangle`, `getstring`, `getkword`, `entsel` — es la interfaz de entrada, y toda ella entiende que el usuario puede responder **haciendo clic en el dibujo**. `getpoint` devuelve una lista `(x y z)`: los puntos son listas de números, sin tipo especial.
- `rtos` es *real to string*: `(rtos total 2 2)` significa modo **2** (decimal) con **2** decimales. Los modos son 1 científico, 2 decimal, 3 pies y pulgadas, 4 arquitectónico, 5 fraccionario — la lista delata para qué se diseñó el lenguaje.
- `princ` imprime. **El `(princ)` final y sin argumentos no es un descuido**: es el idioma obligatorio para que la función devuelva "nada" y AutoCAD no eche un eco del último valor en la línea de comandos. Todo el código AutoLISP del mundo termina así.

Y ahora lo que hace de AutoLISP lo que es: el acceso al dibujo.

```
(defun c:RADIO ( / ent datos )
  (setq ent   (car (entsel "\nSelecciona un círculo: ")))
  (setq datos (entget ent))
  (princ (strcat "\nTipo: "  (cdr (assoc 0  datos))))
  (princ (strcat "\nRadio: " (rtos (cdr (assoc 40 datos)) 2 3)))
  (princ)
)
```

`entsel` deja al usuario **señalar un objeto con el ratón**; `entget` devuelve ese objeto como una **lista de asociaciones**, y `assoc` extrae un campo por su número. Esos números son los **códigos de grupo DXF**: el `0` es el tipo de entidad, el `8` la capa, el `10` el punto de inserción, el `40` el radio. No son arbitrarios: son el formato de intercambio de AutoCAD, y aprenderlos es aprender el modelo de datos del programa.

Ahí está la lección: **la biblioteca estándar de un lenguaje incrustado es el modelo de objetos del anfitrión.** Aprender AutoLISP es, en un 20 %, aprender Lisp, y en un 80 %, aprender AutoCAD.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En AutoLISP es…
`x = 5`	`(setq x 5)`
`def f(a, b):`	`(defun f (a b / locales) ...)`
Variable local	Detrás de la `/` en la lista de argumentos — **no hay `let` clásico**
`print(x)`	`(princ x)` / `(prompt x)` / `(alert x)`
`str(x)`	`(rtos x 2 2)` para reales, `(itoa x)` para enteros
`int(s)` / `float(s)`	`(atoi s)` / `(atof s)`
`a + b`	`(+ a b)`
`if / else`	`(if condicion entonces si-no)`
`for x in lista`	`(foreach x lista ...)`
`dict["clave"]`	`(cdr (assoc clave lista))` — listas de asociación
Llamar a la app anfitriona	`(command "_LINE" p1 p2 "")` — ejecuta un comando de AutoCAD
API orientada a objetos	Las funciones `vla-*` de Visual LISP sobre ActiveX

⚠️ Errores comunes al leerlo

- **Olvidar la `/` de variables locales.** Efectos entre rutinas imposibles de reproducir.
- **Olvidar el `(princ)` final.** La función devuelve el último valor y AutoCAD lo imprime, ensuciando la línea de comandos.
- **Usar `command` con nombres de comando localizados.** En un AutoCAD en español, `(command "LINEA")` funciona y `(command "LINE")` no. El prefijo de subrayado — `(command "_LINE")` — fuerza el nombre en inglés, y el prefijo de punto — `"._LINE"` — fuerza además el comando original aunque esté redefinido. Escribir `"._LINE"` no es manía: es la única forma de que la rutina funcione en cualquier instalación.
- **Confundir `setq` con `set`.** `setq` no evalúa el nombre de la variable; `set` sí, y se usa con símbolos entrecomillados.
- **Suponer aritmética real.** `(/ 7 2)` da **3**, no 3.5: si todos los argumentos son enteros, la división es entera. Hay que escribir `(/ 7.0 2)`.
- **Esperar Common Lisp.** No hay macros, ni CLOS, ni condiciones, ni paquetes. AutoLISP es un Lisp pequeño; el manejo de errores se hace con `vl-catch-all-apply` y con una función `*error*` propia.

📚 Fuentes y bibliografía

- AutoLISP Developer's Guide (Autodesk) (https://help.autodesk.com/view/OARX/2026/ENU/?guid=GUID-4CEE5072-8817-4920-8A2D-7060F5E16547) — la guía oficial vigente.
- AutoLISP Reference — funciones (https://help.autodesk.com/view/OARX/2026/ENU/?guid=GUID-A1301F70-D9DE-4E31-B0AD-B9E1B24CB0E0) — el catálogo completo de funciones, la página que se tiene siempre abierta.
- DXF Reference (https://help.autodesk.com/view/ACD/2026/ENU/?guid=GUID-235B22E0-A567-4CF6-92D3-38A2306D73F3) — los códigos de grupo; imprescindible para trabajar con `entget`.
- AutoCAD AutoLISP Extension para VS Code (https://marketplace.visualstudio.com/items?itemName=Autodesk.autolispext) — el entorno moderno de edición y depuración.
- **Lee Ambrosius**, *AutoCAD Platform Customization: AutoLISP*, Sybex — el libro de referencia actual; el autor escribe la documentación oficial de Autodesk.
- **Lee Mac**, lee-mac.com (https://www.lee-mac.com/) — colección pública de rutinas y tutoriales que es, de facto, la escuela de la comunidad hispanohablante y anglosajona.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Common Lisp · VBA · Tcl/Tk

🐚 Bash — 1989

Bash es, con casi total seguridad, **el lenguaje que más veces se ejecuta al día en el mundo**: cada arranque de un contenedor, cada canalización de integración continua y cada tarea programada de cualquier servidor Linux pasa por él. Y es también **el que menos gente reconoce como un lenguaje de programación** — hasta que un guion de cuatrocientas líneas falla en producción.

> 🎯 ***Por qué está en este programa***
>
> *Bash es un **primo de la familia históricos / shell** (Atlas).*
>
> *Aporta al programa **la composición de procesos por tuberías** (clase 161) —la idea de McIlroy de 1964 que sostiene el contrato de las 136 clases de este curso— y el **componente de automatización** de la clase 171 en su forma más extendida. Y aporta un caso serio de la clase 153: **la inyección de comandos y por qué la citación importa**.*

Año	1989; **Bourne shell** de 1979; **Bash 5.x** actual
Autoría	**Brian Fox** para GNU; después **Chet Ramey**, que lo mantiene desde 1993
Familia	Históricos / shell; sucesor del **Bourne shell** de Stephen Bourne
Paradigma	Imperativo, **orientado a la composición de procesos**
Tipado	**Todo es texto**; con arreglos y enteros como añadidos
Memoria	No aplica: el intérprete la gestiona
Ejecución	Interpretado, línea a línea
Estado	🟢 **Omnipresente** en Unix, Linux, macOS, contenedores y CI

📜 Historia

Stephen Bourne escribió el **Bourne shell (`sh`)** en los Laboratorios Bell en 1979, sustituyendo al shell de Thompson. Y aportó lo que hoy es el lenguaje: **estructuras de control, funciones, variables, sustitución de comandos y redirección**.

Su predecesor, el shell de Thompson (1971), había implementado **las tuberías** que **Doug McIlroy** había propuesto en un memorando de 1964 con una imagen que se ha citado mil veces:

> *Deberíamos tener alguna forma de acoplar programas como **mangueras de jardín**.*

Y de ahí salió **la filosofía de Unix** (clase 161): programas pequeños que hacen una cosa, que leen de la entrada estándar y escriben en la salida estándar, y que se componen.

Bash —*Bourne Again SHell*, un juego de palabras— lo escribió **Brian Fox** en **1989** para el proyecto GNU, como reemplazo libre del `sh` de AT&T, añadiendo lo bueno del **C shell** y del **Korn shell**: historial, edición de línea, `[[ ]]`, arreglos.

Se convirtió en el shell por defecto de **Linux**, y con Linux llegó a todas partes. **macOS lo usó hasta 2019**, cuando cambió a **zsh** por la licencia GPLv3 de las versiones nuevas de Bash — un detalle de licencias con consecuencias reales.

Y en **2014** tuvo su momento más incómodo: **Shellshock**, una vulnerabilidad en el manejo de funciones exportadas por variables de entorno que estuvo presente **veinticinco años** y afectó a millones de servidores.

🏭 Dónde vive hoy

- **Todo servidor Linux**: guiones de arranque, tareas de `cron`, mantenimiento (clase 171).
- **Contenedores**: cada `RUN` de un `Dockerfile` es un comando de shell (clase 174).
- **Integración continua**: los pasos de GitHub Actions, GitLab CI y Jenkins son, casi siempre, shell (clase 147).
- **Herramientas de desarrollo**: `./configure`, los guiones de instalación, los ganchos de git (clase 145).
- **Y como pegamento** entre cualquier par de programas que hablen texto.

🧠 Lo que enseña: componer procesos, y las trampas de la citación

Uno, la tubería, que es el contenido de la clase 161:

```
grep ERROR registro.log | awk '{print $4}' | sort | uniq -c | sort -rn | head
```

Cinco programas, ninguno de los cuales sabe de los otros, encadenados por texto. Cada uno se escribió por separado, en años distintos, y componen porque **todos respetan el contrato: entrada estándar, salida estándar** — que es exactamente el contrato de este curso (clase 040).

Dos, y es la parte que hay que aprender bien: la citación (clase 153).

```
# ✗ TODOS estos están mal
rm $fichero                  # si el nombre tiene espacios, borra varias cosas
if [ $var = "x" ]             # si $var está vacío, error de sintaxis
for f in $(ls *.txt)           # se rompe con espacios y con saltos de línea
eval "comando $entrada"         # inyección de comandos

# ✓ Y así se escriben
rm -- "$fichero"
if [[ "$var" == "x" ]]
for f in *.txt
```

La regla es simple y casi nadie la sigue: entrecomillar SIEMPRE las variables. Sin comillas, Bash **divide por espacios y expande comodines** sobre el valor — que es la causa de la mayoría de los guiones que fallan con un nombre de fichero raro, y de bastantes incidentes de seguridad.

Y tres, las tres líneas que deberían encabezar todo guion serio:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

- `-e`: parar al primer comando que falle. **Sin esto, un guion sigue adelante tras un error** —el problema que `MONMSG` resuelve en CL (clase 171)—.
- `-u`: fallar al usar una variable no definida. **Es el `implicit none` de Bash** (clase 137), y evita el desastre clásico de `rm -rf "$DIR/"` con `$DIR` vacío.
- `-o pipefail`: que una tubería falle si falla **cualquiera** de sus partes, no solo la última.

> ***Y la advertencia de la clase 171 aplica aquí más que en ningún sitio: el guion de shell es código de producción.*** *Despliega, borra y mueve datos.* ***Merece control de versiones, revisión (clase 146) y*** `shellcheck` *— que es un analizador estático excelente y gratuito, y que detecta la mayoría de los errores de esta sección.*

🔄 Lo que se ha modernizado

- **Bash 5**: arreglos asociativos maduros, `${var@Q}` para citar de forma segura, mejoras de rendimiento.
- **ShellCheck**: análisis estático que **debería estar en la integración continua de cualquier repositorio con guiones** (clases 146 y 147).
- `shfmt`: formateador, como `gofmt` (clase 146).
- `bats`: marco de pruebas para guiones de shell — sí, se pueden probar (clase 139).
- **Y la competencia sana**: **zsh** (por defecto en macOS), **fish** (interactivo y amable) y **nushell** (con datos estructurados en lugar de texto plano), que replantea la premisa de McIlroy.

⚙️ Cómo se ejecuta hoy

```
bash main.sh < entrada.txt          # ejecutar
./main.sh                            # con el shebang y permiso de ejecución

shellcheck main.sh                    # ← análisis estático (clase 146)
shfmt -d main.sh                      # formato
bats pruebas/                          # pruebas (clase 139)
```

🧪 El programa de la clase 041 en Bash

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
#!/usr/bin/env bash
set -euo pipefail

read -r precio cantidad descuento

# Bash NO tiene aritmética de coma flotante: hay que delegar en bc o en awk.
total=$(awk -v p="$precio" -v c="$cantidad" -v d="$descuento" \
            'BEGIN { printf "%.2f", p * c * (1 - d) }')

printf 'Total: %s\n' "$total"
```

Lo que hay que ver, y es lo más honesto que puede decir esta ficha.

- **Bash no sabe multiplicar números con decimales.** `$(( ))` es **solo aritmética entera**, así que **hay que llamar a otro programa** — `awk`, `bc` o `python3` —. **Y esa limitación es la ficha entera**: Bash no es un lenguaje de cálculo, **es un lenguaje para componer programas que sí calculan** (clase 155).
- `read -r` con `-r` es obligatorio: sin él, la barra invertida se interpreta como escape.
- **Cada variable va entrecomillada**, incluidas las de `awk -v`.
- `printf` **en lugar de** `echo`: `echo` **interpreta las barras invertidas de forma distinta según el shell y las opciones**, y no es portable. `printf` sí lo es, y es la recomendación universal.
- **Y** `set -euo pipefail` **en la segunda línea**, que es lo que convierte un guion frágil en uno que falla cuando debe.

📚 Fuentes y bibliografía

- Manual de Bash (GNU) (https://www.gnu.org/software/bash/manual/) — la referencia; densa y completa.

- BashFAQ y BashPitfalls (https://mywiki.wooledge.org/BashPitfalls) — **la lectura más útil de esta ficha**: una lista de errores comunes con la explicación de por qué fallan. Cambia cómo se escriben los guiones.
- ShellCheck (https://www.shellcheck.net/) — pegar un guion y ver qué está mal, en el navegador.
- Google Shell Style Guide (https://google.github.io/styleguide/shellguide.html) — incluida la recomendación de **cuándo NO usar shell** (a partir de unas cien líneas o cuando hay estructuras de datos).
- **Doug McIlroy**, el memorando de 1964 y la filosofía de Unix — contexto de la clase 161.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Perl · Tcl · PowerShell · JCL · Python

🔤 BASIC — 1964

BASIC se diseñó con un objetivo que ningún otro lenguaje de este Atlas tuvo: **que lo pudiera usar un estudiante de historia**. Y lo consiguió tanto que durante los años ochenta **venía grabado en la ROM de casi todos los ordenadores domésticos del mundo** — al encenderlos, lo primero que aparecía era un intérprete de BASIC esperando.

> 🎯 ***Por qué está en este programa***
>
> *BASIC es un **primo de la familia históricos / shell** (Atlas), y es el antepasado de dos fichas de este Atlas: VBA y VB.NET.*
>
> *Aporta al programa una idea que el curso defiende desde varios ángulos: **la accesibilidad es una decisión de diseño con consecuencias**. Y aporta el contraste histórico con la clase 083 —**el `GOTO` y por qué la programación estructurada tuvo que imponerse**—, que es difícil de entender sin ver de dónde veníamos.*

Año	**1 de mayo de 1964**, a las 4 de la madrugada, en Dartmouth
Autoría	**John G. Kemeny** y **Thomas E. Kurtz**, Dartmouth College
Familia	Históricos; con influencia de FORTRAN y ALGOL
Paradigma	Imperativo; estructurado a partir de los años ochenta
Tipado	Dinámico o estático según el dialecto; con sufijos (`$`, `%`, `!`)
Memoria	Automática; sin punteros en los dialectos clásicos
Ejecución	Interpretado —lo habitual— o compilado (QuickBASIC, PowerBASIC)
Estado	🟡 **Histórico**, con descendientes muy vivos: VBA y VB.NET

📜 Historia

En 1963, **John Kemeny** y **Thomas Kurtz**, matemáticos de Dartmouth, tenían una convicción poco compartida: **que todos los estudiantes —no solo los de ciencias— deberían saber usar un ordenador**.

El problema era que programar entonces significaba **FORTRAN o ensamblador**, con tarjetas perforadas y un turno de días. Así que hicieron dos cosas a la vez:

1. **El Sistema de Tiempo Compartido de Dartmouth**, para que varias personas usaran el ordenador simultáneamente desde teletipos.

2. **BASIC** —*Beginner's All-purpose Symbolic Instruction Code*—, un lenguaje que se pudiera aprender en una tarde.

El primer programa BASIC se ejecutó el 1 de mayo de 1964, y en pocos años **más del 80 % de los estudiantes de Dartmouth sabía programar** — una cifra que no se ha vuelto a alcanzar en ninguna universidad generalista.

Y los autores tomaron una decisión que merece recordarse: **lo pusieron en el dominio público**, sin licencia ni restricciones, porque su objetivo era que se extendiera.

Se extendió. En 1975, **Bill Gates y Paul Allen** escribieron **Altair BASIC** — el primer producto de Microsoft. Y en los años ochenta, **BASIC venía en la ROM** del Commodore 64, el ZX Spectrum, el Apple II, el Amstrad CPC y el MSX.

> ***Eso significa que una generación entera aprendió a programar encendiendo el ordenador y escribiendo.*** *No había que instalar nada, ni configurar nada, ni elegir nada: el ordenador arrancaba en un lenguaje de programación. Es una propiedad que se perdió y que muchos consideran una pérdida real.*

Y también recibió la crítica más famosa que ha recibido un lenguaje: **Edsger Dijkstra** escribió en 1975 que *"es prácticamente imposible enseñar buena programación a estudiantes que han estado expuestos a BASIC: como programadores potenciales, están mentalmente mutilados sin esperanza de regeneración."*

Era una exageración polémica y señalaba algo real: el BASIC de la época **tenía números de línea y `GOTO`, y no tenía funciones con parámetros locales ni bloques**. Se programaba con saltos, y eso producía lo que se llamó **código espagueti** (clase 083).

Y el propio Kurtz estuvo de acuerdo: en 1984 diseñó **True BASIC**, estructurado y sin números de línea. **QuickBASIC (1985)** hizo lo mismo en el mundo Microsoft, con procedimientos, tipos definidos por el usuario y un compilador de verdad.

🏭 Dónde vive hoy

- **Como VBA**: en Excel, Access y Office — probablemente **el lenguaje con más usuarios activos del mundo** que no se consideran programadores.
- **Como VB.NET**: aplicaciones de empresa en mantenimiento.
- **FreeBASIC, QB64 y PureBasic**: dialectos modernos, con comunidades pequeñas y activas.
- **Retroinformática y educación**: emuladores del C64 y del Spectrum, y proyectos como el BBC micro:bit que recuperan la idea del ordenador que arranca programable.
- **Y en sistemas industriales heredados**, con dialectos propios de cada fabricante.

🧠 Lo que enseña: la programación estructurada, vista desde antes

Este es el BASIC que Dijkstra criticaba, y **merece verlo para entender la clase 083**:

```
10 INPUT "Precio, cantidad, descuento"; P, C, D
20 IF P <= 0 THEN GOTO 60
30 LET T = P * C * (1 - D)
40 PRINT "Total: "; T
50 GOTO 70
60 PRINT "Precio no válido"
70 END
```

Los números de línea y el `GOTO` son el problema (clase 083): con veinte líneas se lee; con dos mil, **el flujo de control es un grafo que nadie puede seguir**, y **no hay ámbitos**: todas las variables son globales (clase 087).

Y la respuesta fue la programación estructurada, que Böhm y Jacopini habían demostrado suficiente en 1966 —**secuencia, selección e iteración bastan para cualquier programa**— y que Dijkstra defendió en *Go To Statement Considered Harmful*.

Y el mismo programa en BASIC moderno **es otro lenguaje**:

```
Dim As Double p, c, d
Input "", p, c, d

If p <= 0 Then
    Print "Precio no valido"
Else
    Print Using "Total: ##.##"; p * c * (1 - d)
End If
```

Sin números de línea, con bloques, con tipos declarados y con `End If`. Es exactamente la transición que Fortran hizo con el formato libre y que RPG hizo con el formato totalmente libre (clase 146) — **el mismo lenguaje, después de aprender**.

Y la lección que queda es de la clase 175: **casi ninguna decisión antigua fue estúpida cuando se tomó**. Los números de línea eran **el editor**: sin pantalla ni ratón, escribir `25 PRINT X` insertaba una línea entre la 20 y la 30. **Era una solución a un problema real que dejó de existir.**

🔁 Lo que quedó

- **La idea de que el ordenador debe ser programable por su usuario**, que sobrevive en las hojas de cálculo (clase 163) y en proyectos educativos.
- **El REPL como forma de aprender**: escribir una línea y ver el resultado (clase 124).
- **VBA y VB.NET**, que son descendientes directos.
- **Y `Print Using`**, cuya idea de formato por plantilla llegó hasta COBOL y las imágenes de edición.

⚙️ Cómo se ejecuta hoy

```
fbc main.bas && ./main            # FreeBASIC: compila a nativo
qb64pe -x main.bas                 # QB64 Phoenix Edition
bwbasic main.bas                    # Bywater BASIC, el clásico de GNU
# Y en el navegador: emuladores de C64 y de ZX Spectrum
```

🧪 El programa de la clase 041 en BASIC

Esta versión se escribe aquí y **no está verificada en CI** (clase 040). Está en **FreeBASIC**, estructurado.

```
Dim As Double precio, cantidad, descuento, total

Input "", precio, cantidad, descuento

total = precio * cantidad * (1 - descuento)

Print "Total: " & Format(total, "0.00")
```

Lo que hay que ver.

- `Dim As Double` declara el tipo — algo que el BASIC clásico no exigía, y que los dialectos modernos recomiendan siempre (`Option Explicit`), por la misma razón que `implicit none` en Fortran y `use strict` en Perl (clase 137): **sin declaración, un nombre mal escrito crea una variable nueva**.
- `&` **concatena**, como en VB.NET — y por el mismo motivo: **evitar la ambigüedad de** `+` entre suma y concatenación (clase 100).
- **Las palabras en lugar de símbolos** — `Dim`, `Print`, `Input`, `End If` — son la herencia de la ficha: **legibilidad para quien empieza**, y la razón por la que se eligió el nombre "Beginner's".
- **Y no hay números de línea**, que es la diferencia entera entre el BASIC que Dijkstra criticó y este.

📚 Fuentes y bibliografía

- **Kemeny y Kurtz**, *Back to BASIC: The History, Corruption, and Future of the Language* (1985) — los autores explicando qué querían y qué creen que se estropeó por el camino.
- **Edsger Dijkstra**, *Go To Statement Considered Harmful* (1968) y *How do we tell truths that might hurt?* (1975) — la crítica; corta y polémica.
- **Böhm y Jacopini** (1966) — el teorema que demuestra que secuencia, selección e iteración bastan (clase 083).
- FreeBASIC (https://www.freebasic.net/) y QB64 Phoenix (https://qb64phoenix.com/) — los dialectos vivos.
- **Dartmouth**, *BASIC at 50* (https://www.dartmouth.edu/basicfifty/) — archivo histórico con el contexto del sistema de tiempo compartido.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: VBA · VB.NET · Pascal · Fortran

🔨 C — 1972

Está en esta lista, pero no es legacy — y esa distinción es el contenido de la ficha. C tiene la edad de COBOL menos trece años y una diferencia decisiva: no sobrevive por el coste de migrar, sino porque **sigue siendo la mejor herramienta para lo que hace**. Es, además, el idioma en el que todos los demás lenguajes se hablan entre sí.

> 🎯 ***Por qué está en este programa***
>
> ***C es uno de los diez lenguajes del núcleo****: se implementa y se verifica en CI en las 136 clases de código. Esta ficha no repite eso — cuenta su historia y responde a la pregunta que motiva esta sección: ¿por qué un lenguaje de 1972 no es un lenguaje viejo?*
>
> *La respuesta corta: porque* ***C no es solo un lenguaje, es una interfaz****. La ABI de C es el único terreno común donde Python llama a NumPy, Rust llama al sistema operativo, Java llama a una biblioteca nativa y Go llama a SQLite. Cuando en la Parte 10 se estudia la interoperabilidad, se estudia la ABI de C, porque* ***no hay otra****. Un lenguaje puede sustituir a C escribiendo código nuevo; no puede sustituir a C como punto de encuentro.*

Año	1972; **K&R** en 1978; ANSI **C89**; **C99**, **C11**, **C17**, **C23**
Autoría	**Dennis Ritchie**, Bell Labs, a partir de **B** (Thompson) y **BCPL** (Richards)
Familia	C / llaves — la raíz sintáctica de C++, Java, C#, Go, Rust, JavaScript, PHP…
Paradigma	Imperativo y procedimental
Tipado	Estático y **débil**: las conversiones son fáciles y a menudo silenciosas
Memoria	**Manual**: `malloc` / `free`, sin red de seguridad
Ejecución	Compilado a nativo
Estado	🟢 **Tecnología fundamental** — no debería llamarse legacy

---

📜 Historia

A finales de los 60, Ken Thompson escribió **B** en los Laboratorios Bell —una simplificación de **BCPL**, de Martin Richards— para el PDP-7. B no tenía tipos: todo era una palabra de máquina. Al llegar el PDP-11, con bytes y palabras de distinto tamaño, esa carencia se volvió insostenible, y **Dennis Ritchie** extendió B con un sistema de tipos. El resultado se llamó **C** por continuación alfabética.

El movimiento que lo cambió todo llegó en **1973**: Thompson y Ritchie **reescribieron el núcleo de Unix en C**. Hasta entonces, un sistema operativo era ensamblador por definición. Al escribirlo en un lenguaje de alto nivel, Unix se volvió **portable**: llevarlo a una máquina nueva pasó de ser un proyecto de años a ser un proyecto de meses. Unix se extendió, y C viajó con él a todas partes.

En **1978**, Brian Kernighan y Dennis Ritchie publicaron *The C Programming Language*: 228 páginas que sirvieron a la vez de tutorial, de manual y de estándar de facto durante una década. Es probablemente el libro de programación más influyente jamás escrito, y su ejemplo de apertura fijó para siempre el ritual del `hello, world`.

La estandarización llegó con **ANSI C (C89/C90)**, y después **C99** (comentarios `//`, `bool`, declaraciones en medio del bloque, arrays de longitud variable), **C11** (hilos, atómicos, `_Generic`), **C17** (correcciones) y **C23** (`constexpr`, `nullptr`, `typeof`, `bool` de verdad, literales binarios, atributos).

El diseño se resume en una frase de Kernighan y Ritchie que sigue siendo la mejor descripción del lenguaje: **"C no es un lenguaje grande, y no está bien servido por un libro grande."** Confía en el programador, no impide lo que parece peligroso, y a cambio no oculta nada.

🏭 Dónde vive hoy

- **Núcleos de sistemas operativos**: Linux (decenas de millones de líneas), el núcleo de Windows, los BSD, macOS/XNU.
- **Firmware y embebidos**: desde el microcontrolador de un termostato hasta el software de un automóvil, bajo normas como **MISRA C**. Es, con mucho, el mayor volumen de C que se escribe hoy.
- **Infraestructura de Internet**: OpenSSL, nginx, curl, OpenSSH, los servidores DNS.
- **Bases de datos**: SQLite, PostgreSQL, MySQL, Redis.
- **Los intérpretes de otros lenguajes**: **CPython**, **PHP**, **Ruby (MRI)**, **Perl**, **Lua**, el motor de **R**. Cuando escribes Python, ejecutas C.
- **Bibliotecas numéricas y multimedia**: FFmpeg, zlib, libpng, la capa nativa de NumPy.
- **Herramientas**: git, GCC, los coreutils, prácticamente cualquier utilidad de Unix.

🧠 Por qué no es legacy

1. Es la ABI universal. No hay una "ABI de Python" ni una "ABI de Rust" a la que otros lenguajes puedan hablar. Hay la ABI de C. Todo mecanismo de interoperabilidad — `ctypes`, JNI, P/Invoke, cgo, `extern "C"` de Rust, N-API de Node— consiste en **fingir ser C** durante un momento. Mientras exista software escrito en más de un lenguaje, existirá esa necesidad.

2. Es la superficie del sistema operativo. Las llamadas al sistema de Linux, la API de Windows y POSIX están definidas en C. Programar contra el sistema operativo es hablar C, aunque lo escribas con otro nombre.

3. Modelo mental transparente. No hay recolector de basura que se despierte en un momento inconveniente, ni destructores implícitos, ni asignaciones ocultas. En un manejador de interrupción o en un sistema de tiempo real duro, esa previsibilidad no es una preferencia estética: es un requisito.

4. Portabilidad extrema. Hay compilador de C para arquitecturas que ningún otro lenguaje soporta. Un microcontrolador de 8 bits con 2 KB de memoria tiene compilador de C.

5. Cuarenta años de código probado en producción. No como deuda: como activo. SQLite es posiblemente el software más desplegado del mundo y su fiabilidad procede de una suite de pruebas descomunal sobre una base de C estable.

> ***Y la crítica honesta, que también forma parte de la ficha.*** *La gestión manual de memoria de C es el origen de una parte muy grande de las vulnerabilidades graves de la industria: desbordamientos de búfer, uso después de liberar, doble liberación. Los análisis públicos de bases de código grandes han situado repetidamente esa familia de fallos en torno a* ***dos tercios*** *de las vulnerabilidades críticas de memoria, y las agencias de ciberseguridad recomiendan hoy lenguajes seguros por diseño para código nuevo. Eso* ***no*** *convierte a C en obsoleto —Rust y Zig existen porque el problema es difícil, no porque esté resuelto—, pero sí significa que elegir C en 2026 debe ser una decisión argumentada, no una inercia. Esta es la tensión de fondo de todo el debate actual sobre lenguajes de sistemas, y merece la pena entenderla antes de tomar partido.*

🔄 Lo que se ha modernizado

- **C23**, el estándar vigente: `constexpr`, `nullptr` (adiós a la ambigüedad de `NULL`), `typeof`, `bool` / `true` / `false` como palabras del lenguaje, literales binarios `0b`, atributos `[[...]]` y `#embed` para incrustar ficheros binarios en tiempo de compilación.
- **Herramientas de seguridad que cambian el juego**: los *sanitizers* (**ASan**, **UBSan**, **TSan**, **MSan**) detectan en ejecución los errores de memoria y el comportamiento indefinido; **fuzzing** continuo (libFuzzer, AFL++, OSS-Fuzz) encuentra fallos que ninguna revisión humana encontraría; y el análisis estático (clang-tidy, Coverity, el analizador de GCC) atrapa clases enteras de defectos. El C que se escribe hoy con este instrumental es mucho más seguro que el de hace quince años.
- **Sistemas de construcción y dependencias modernos**: CMake, Meson, Ninja, y gestores de paquetes como Conan o vcpkg que también sirven a proyectos C.
- **Normas de codificación para dominios críticos**: **MISRA C** en automoción e industria, **CERT C** en seguridad, con verificación automatizada.
- **Convivencia con Rust**: el núcleo de Linux admite desde 2022 controladores escritos en Rust llamando a las interfaces C existentes. El futuro más probable no es la sustitución, sino la frontera bien definida — que es, otra vez, la ABI de C.

⚙️ Cómo se ejecuta hoy

```
cc total.c -o total          # el comando de la clase 041
gcc -std=c23 -Wall -Wextra -O2 total.c -o total

# Con red de seguridad, que es como conviene desarrollar:
gcc -fsanitize=address,undefined -g total.c -o total
```

🧪 El programa de la clase 041 en C

Es el mismo que se ejecuta y se verifica en la clase 041; aquí interesa por comparación con las demás fichas.

```
#include <stdio.h>

int main(void) {
    double precio, cantidad, descuento;

    if (scanf("%lf %lf %lf", &precio, &cantidad, &descuento) != 3) {
        return 1;
    }

    double total = precio * cantidad * (1 - descuento);
    printf("Total: %.2f\n", total);
    return 0;
}
```

Lo que hay que ver, comparando con las otras fichas de esta sección.

- `&precio` **es la diferencia fundamental.** C solo pasa argumentos **por valor**, así que para que `scanf` pueda escribir en tus variables hay que darle su **dirección**. Fortran pasa por referencia y no lo necesita; Pascal tiene `var`; COBOL no tiene funciones con parámetros en el sentido moderno. Ese `&` es toda una filosofía: **el programador gestiona las direcciones explícitamente.**
- **Comprobar el retorno de** `scanf` **no es opcional.** Devuelve cuántos elementos convirtió. Si la entrada no encaja, las variables quedan **sin inicializar** y leerlas es comportamiento indefinido. Casi todo el código de ejemplo que circula omite esta comprobación, y ese hábito es exactamente el problema que discutimos arriba.
- `%lf` **en** `scanf` **y** `%f` **en** `printf` para el mismo `double`. La asimetría existe porque en una función variádica los `float` se promueven a `double`, así que `printf` no puede distinguirlos, pero `scanf` recibe punteros y sí debe saber el tamaño. Es un detalle histórico que sigue sorprendiendo.
- `printf` **no valida nada.** Si el formato no coincide con los argumentos, el programa lee la pila como si fuera lo que dijiste. Los compiladores modernos avisan con `-Wformat`, pero el lenguaje no lo impide.
- **Todo esto es una elección de diseño coherente.** C te deja hacer lo que quieras y no te vigila. Es la razón de que sea rápido, portable y universal, y también la razón de que haya que escribirlo con disciplina y herramientas.

📚 Fuentes y bibliografía

- cppreference — sección C (https://en.cppreference.com/w/c) — la mejor referencia en línea del lenguaje y su biblioteca estándar.
- Documentación de GCC (https://gcc.gnu.org/onlinedocs/) y Clang (https://clang.llvm.org/docs/) — incluidos los *sanitizers*, que conviene conocer desde el primer día.
- SEI CERT C Coding Standard (https://wiki.sei.cmu.edu/confluence/display/c) — reglas concretas para escribir C seguro.
- **Brian Kernighan, Dennis Ritchie**, *The C Programming Language*, 2.ª ed., Prentice Hall — "el K&R". Anterior a C99, así que no lo uses como referencia de la versión actual, pero léelo: enseña a pensar en C como ningún otro.

- **Ben Klemens**, *21st Century C*, 2.ª ed., O'Reilly — cómo escribir C hoy, con las herramientas y las prácticas actuales. El complemento moderno al K&R.
- **Robert Seacord**, *Effective C*, 2.ª ed., No Starch Press — actualizado a C23 y escrito por uno de los autores de los estándares de seguridad del CERT.
- **Peter van der Linden**, *Expert C Programming: Deep C Secrets* — los rincones oscuros del lenguaje, explicados con humor y precisión.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: C++ · Assembler · Fortran

🍀 Clojure — 2007

Clojure es un **Lisp moderno sobre la JVM**, diseñado por una sola persona con una tesis muy clara: **el problema del software no es la complejidad del dominio, sino la complejidad que añadimos nosotros — y la mayor parte viene del estado mutable**. Todo el lenguaje es la consecuencia de esa idea.

> 🎯 ***Por qué está en este programa***
>
> *Clojure es un **primo de la familia JVM** (Atlas) y también de la **familia Lisp**: comparte máquina virtual con Java y modelo con Common Lisp.*
>
> *Aporta al programa **la inmutabilidad como decisión total** (clase 102) y **las estructuras de datos persistentes** que la hacen viable; y aporta un modelo de concurrencia distinto de todos los del curso: **identidades separadas de valores**, con `atom` y con memoria transaccional (clases 135 y 136).*

Año	2007; **1.0** en 2009; **1.12** actual
Autoría	**Rich Hickey**, tras dos años y medio de trabajo en solitario
Familia	Lisp sobre JVM; con influencia de Common Lisp, Scheme, Haskell y ML
Paradigma	**Funcional, con datos inmutables**; sin orientación a objetos
Tipado	**Dinámico y fuerte**; con `spec` y `malli` para contratos en ejecución
Memoria	La de la JVM
Ejecución	Bytecode JVM; también **ClojureScript** (a JS) y **ClojureCLR**
Estado	🟢 **Nicho sólido y estable**; comunidad pequeña y muy influyente

---

📜 Historia

Rich Hickey llevaba veinte años programando en C++, Java y .NET cuando, en **2005**, se tomó un paréntesis de dos años y medio —financiado con sus ahorros— para diseñar el lenguaje que quería usar.

El resultado, publicado en **2007**, parte de un diagnóstico que expuso en su charla *Simple Made Easy*, una de las más citadas de la historia de la disciplina:

> ***Confundimos "simple" con "fácil".** Simple es no entrelazado; fácil es familiar. **El estado mutable compartido es lo contrario de simple**, aunque nos resulte facilísimo.*

Y de ahí las decisiones:

- **Los datos son inmutables por defecto.** No hay que pedirlo: es lo que hay.
- **Las estructuras son persistentes**: modificar devuelve una versión nueva que **comparte** casi toda la anterior (clase 102), así que la inmutabilidad no cuesta memoria ni tiempo.
- **Estado e identidad se separan**: una identidad (`atom`, `ref`, `agent`) **apunta a un valor inmutable**, y cambiarla es cambiar a qué apunta, de forma controlada.
- **Sintaxis de Lisp**, con literales para mapas, vectores y conjuntos — lo que la hace mucho más legible que la de Common Lisp para datos.

ClojureScript (2011) llevó el lenguaje al navegador, y de ahí salió **Reagent/Re-frame**, cuya arquitectura de estado influyó en el ecosistema React (clase 169).

Y su comunidad tiene una particularidad que merece nombrarse: **es de las más estables de esta lista**. El lenguaje cambia poco y a propósito, y la compatibilidad hacia atrás se cuida — la misma disciplina de Go y Tcl (clase 154).

🏭 Dónde vive hoy

- **Servicios de fondo** en empresas de datos, finanzas y salud: Nubank —el banco digital brasileño, uno de los mayores usuarios del mundo—, Walmart, Cisco, Apple.
- **Análisis y procesamiento de datos**, por la facilidad de componer transformaciones.
- **Front-end con ClojureScript**: Re-frame, y herramientas como Figwheel con recarga en caliente.
- **Datomic**: una base de datos —del mismo autor— cuyo modelo es **inmutable y con historia completa**: nunca se borra, se añaden hechos con su instante (clase 172).

🧠 Lo que enseña: valores, identidades y tiempo

Esta es la idea que hay que llevarse, y no la tiene ningún otro lenguaje del curso:

```
(def cuenta (atom {:saldo 100}))       ; una IDENTIDAD que apunta a un VALOR inmutable

(swap! cuenta update :saldo + 50)       ; ← cambia a QUÉ VALOR apunta, atómicamente
@cuenta                                  ; {:saldo 150}
```

El mapa `{:saldo 100}` nunca cambia. Lo que cambia es a qué valor apunta `cuenta`, y ese cambio es **atómico y sin cerrojos** — con comparación e intercambio por debajo.

Y de ahí sale una propiedad enorme para la clase 136: no hay carreras de datos sobre los valores, porque los valores no se pueden modificar. Solo hay que coordinar las identidades, que son pocas y están declaradas.

Y las estructuras persistentes son lo que lo hace asequible:

```
(def a [1 2 3])
(def b (conj a 4))     ; b es nuevo... y COMPARTE la estructura interna de a
```

No se copia el vector. Por debajo hay un árbol con factor de ramificación 32, y añadir un elemento crea unos pocos nodos nuevos. Es la técnica que la clase 102 describe y que también usan Scala y las bibliotecas inmutables de JavaScript.

Y hay una tercera idea, muy práctica, que Clojure defiende: **usar mapas en lugar de definir clases**.

```
{:nombre "Ana" :edad 30}      ; ← no hace falta declarar un tipo Persona
```

Los datos son datos, y las funciones genéricas de la biblioteca — `get`, `assoc`, `update`, `merge`, `select-keys` — funcionan sobre todos ellos. Es lo contrario del modelado con clases, y `clojure.spec` añade la validación **donde hace falta** en lugar de en todas partes.

🔄 Lo que se ha modernizado

- `clojure.spec` y **malli**: contratos y validación de datos **en ejecución**, con generación de casos de prueba a partir de la especificación (clases 118 y 140).
- **Herramientas oficiales**: `deps.edn` y la CLI de Clojure, con dependencias declarativas y fichero de bloqueo (clase 143).
- `core.async`: canales y procesos ligeros al estilo de Go (clase 134), implementados con macros que transforman el código en una máquina de estados.
- **ClojureScript con compilación avanzada** (Closure Compiler) y **shadow-cljs**.
- **Y** `babashka`: un intérprete de Clojure **con arranque instantáneo** compilado con GraalVM, para guiones de línea de comandos — la respuesta al problema de arranque de la JVM (clase 167).

⚙️ Cómo se ejecuta hoy

```
clojure -M main.clj < entrada.txt      # el comando de la clase 041
bb main.clj                             # babashka: arranque en milisegundos

clj -X:test                             # pruebas
clj-kondo --lint src                    # análisis estático, muy bueno (clase 146)
```

🧪 El programa de la clase 041 en Clojure

```
(require '[clojure.string :as str])

(let [[precio cantidad descuento] (map read-string (str/split (read-line) #" "))
      total (* precio cantidad (- 1 descuento))]
  (println (format "Total: %.2f" (double total))))
```

Lo que hay que ver.

- **La notación prefija** — `(* precio cantidad ...)` — es la de Lisp: **el operador va primero y admite cualquier número de argumentos**. Es lo que permite que el código sea una estructura de datos (clase 123).
- `let` **con desestructuración** `[[a b c] ...]` liga tres nombres de una vez, y **no son variables: son enlaces que no se reasignan**. En todo el programa **no hay una sola mutación** (clase 102).
- `(double total)` **no sobra**: `read-string` puede devolver un entero o una **fracción exacta** —Clojure tiene racionales, como Smalltalk (clase 045)— y `%.2f` necesita un doble. Ese detalle es un buen recordatorio de que **el modelo numérico varía mucho entre lenguajes**.
- **Y compárese con Java, en la misma máquina virtual**: la diferencia no es de sintaxis, es de paradigma. Clojure no crea ningún objeto ni declara ninguna clase.

📚 Fuentes y bibliografía

- clojure.org (https://clojure.org/guides/getting_started) — guías oficiales y la referencia; el apartado sobre **estado e identidad** es lectura obligatoria para la clase 136.
- **Rich Hickey**, charlas *Simple Made Easy*, *The Value of Values* y *Are We There Yet?* — de lo mejor que se ha dicho sobre diseño de software, y aplicable a cualquier lenguaje.
- **Daniel Higginbotham**, *Clojure for the Brave and True* — libre en línea; la mejor introducción.
- **Alex Miller, Stuart Halloway, Aaron Bedra**, *Programming Clojure*, 3.ª ed., Pragmatic.

- **Chas Emerick et al.**, *Clojure Programming*, O'Reilly — completo, con el modelo de concurrencia bien explicado.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Common Lisp · Scheme · Java · Scala · Elixir

🏛️ COBOL — 1959

El lenguaje que mueve el dinero. Si esta madrugada te cobraron una tarjeta, se liquidó una nómina o se calculó una prima de seguro, es probable que una máquina IBM Z ejecutara COBOL para hacerlo. No es una curiosidad de museo: es infraestructura crítica en producción.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: COBOL se ejecuta hoy, en producción, con dinero real encima.*** *No entra por nostalgia ni por capricho histórico; entra porque sigue vivo en banca, seguros, medios de pago y administración pública, e IBM sigue vendiendo y actualizando su compilador.*
>
> *Y entra, sobre todo, porque* ***deja a la vista un concepto que el núcleo esconde****: la* ***aritmética decimal exacta*** *y la* ***declaración explícita de la forma del dato****. En Python escribes* `0.1 + 0.2` *y obtienes* `0.30000000000000004` *sin que nada te avise; en COBOL el tipo* `PIC 9(9)V99 COMP-3` *te obliga a decidir, antes de calcular, cuántos dígitos y cuántos decimales tiene el dinero. Ver el mismo cálculo en los dos sitios es lo que convierte "usa* `Decimal` *para dinero" de receta memorizada en comprensión.*

Año	1959 (especificación); primeros compiladores en 1960
Autoría	Comité **CODASYL**, con Grace Hopper como influencia decisiva vía FLOW-MATIC
Familia	Negocios / procesamiento de datos por lotes
Paradigma	Imperativo y procedimental; OO opcional desde COBOL 2002
Tipado	Estático, declarativo por **plantilla de imagen** (`PICTURE`)
Memoria	Estática por defecto — el `WORKING-STORAGE` se dimensiona al compilar
Ejecución	Compilado a nativo (z/OS, AIX, Linux, Windows)
Estado	🟢 **Muy vivo en legacy empresarial** — banca, seguros, gobierno, pensiones, tarjetas

---

📜 Historia

En 1959 el Departamento de Defensa de Estados Unidos convocó en el Pentágono a fabricantes y usuarios para atacar un problema concreto: cada máquina tenía su propio lenguaje y un programa de nóminas escrito para un UNIVAC no servía en un IBM. De esa reunión nació el comité **CODASYL** (*Conference on Data Systems Languages*) y, en pocos meses, la especificación de **COBOL** (*COmmon Business-Oriented Language*).

La decisión de diseño que lo define se tomó ahí: el lenguaje debía poder ser **leído por alguien que no fuera programador**. De ahí la sintaxis en inglés casi narrativo — `ADD IVA TO TOTAL GIVING TOTAL-CON-IVA` — que hoy nos parece verbosa y que entonces era una apuesta deliberada por la auditabilidad. El trabajo previo de **Grace Hopper** en FLOW-MATIC, el primer lenguaje con instrucciones en inglés, es el antecedente directo.

La otra decisión estructural fue separar el programa en **cuatro divisiones**: `IDENTIFICATION` (quién es), `ENVIRONMENT` (en qué máquina y con qué ficheros corre), `DATA` (la forma exacta de cada dato) y `PROCEDURE` (qué se hace). Esa separación entre **la forma del dato y el proceso** es la razón de que un COBOL de 1975 siga

siendo legible: el `DATA DIVISION` es, en la práctica, un esquema documentado del negocio.

Los estándares posteriores fueron acumulando: **COBOL-68** y **74** consolidaron el lenguaje; **COBOL-85** trajo la programación estructurada de verdad (`END-IF`, `EVALUATE`, `PERFORM ... UNTIL`) y permitió por fin escribir COBOL sin `GO TO`; **COBOL 2002** añadió orientación a objetos y formato libre; **2014** y **2023** siguen puliendo. El lenguaje no se congeló en los 60: se congeló su reputación.

🏭 Dónde sobrevive hoy

- **Banca central y comercial**: sistemas de cuentas, liquidación interbancaria, compensación.
- **Medios de pago**: autorización y liquidación de tarjetas, procesamiento nocturno de lotes.
- **Seguros**: cálculo de primas, siniestros, reservas actuariales.
- **Gobierno y pensiones**: seguridad social, recaudación fiscal, padrones.
- **Logística y retail** de gran escala, sobre mainframe IBM Z con CICS, IMS y Db2.

Lo típico no es un programa COBOL suelto, sino un **ecosistema**: el COBOL contiene la regla de negocio, JCL orquesta la ejecución por lotes, **CICS** le da la cara transaccional en línea y **Db2** guarda los datos. Estudiar COBOL sin ver ese entorno es como estudiar JavaScript sin un navegador.

🧠 Por qué no ha muerto

Hay una respuesta perezosa —"porque migrar cuesta caro"— y una respuesta técnica, que es la interesante.

1. Aritmética decimal exacta. COBOL no calcula dinero en punto flotante binario. Su tipo natural es el **decimal empaquetado** (`COMP-3`): cada dígito decimal se guarda en medio byte, y las operaciones son decimales, no binarias. En Python o JavaScript, `0.1 + 0.2` no da `0.3`; en COBOL con `PIC 9(9)V99` da exactamente lo que un contable espera. Los lenguajes modernos han tenido que **añadir** ese tipo después (`BigDecimal` en Java, `decimal` en C#, `numeric` en SQL). COBOL nació con él porque nació para eso.

2. El `PICTURE` es un contrato de datos. `PIC S9(7)V99 COMP-3` declara a la vez el signo, los dígitos enteros, los decimales y la representación física en disco. Un fichero de 40 años se sigue leyendo porque su forma está escrita en el programa, no en una convención perdida.

3. Décadas de reglas de negocio validadas. El valor no está en el código: está en los casos límite que ese código ya resolvió. El tratamiento de un año bisiesto en un cálculo de intereses, la excepción de un producto retirado en 1994 que aún tiene clientes. Reescribirlo no es traducir sintaxis, es **redescubrir requisitos que nadie documentó**.

4. El coste del fallo es asimétrico. Un error en un microservicio de recomendaciones es una métrica peor. Un error en la liquidación nocturna de un banco es un incidente regulatorio.

> ***Honestidad sobre las cifras.*** *Circulan estimaciones muy citadas —"800 000 millones de líneas de COBOL en producción"— que provienen de informes de la industria y no de un censo verificable. Tómalas como orden de magnitud, no como dato. Lo verificable es que IBM sigue desarrollando y vendiendo compiladores COBOL (https://www.ibm.com/products/cobol-compilers) y publicando documentación actual para z/OS.*

🔄 Lo que se ha modernizado

Conviene decirlo con claridad, porque es lo que casi nadie cuenta: **COBOL no se quedó en 1985.** El estándar y los compiladores han incorporado lo necesario para resolver problemas actuales:

- **JSON y XML nativos.** `JSON GENERATE` y `JSON PARSE` (y sus equivalentes `XML GENERATE` / `XML PARSE`) son **sentencias del lenguaje**. Un programa COBOL serializa una estructura de datos a JSON sin biblioteca externa. Es exactamente lo que hace falta para hablar con una API.
- **Servicios REST en las dos direcciones.** Con **z/OS Connect**, un programa COBOL existente se expone como API REST sin tocarlo, y también puede **consumir** APIs externas. El mainframe dejó de ser una isla por lotes.
- **Unicode.** El tipo `NATIONAL` (`PIC N`) maneja UTF-16; el COBOL de los 80 solo entendía EBCDIC.
- **Orientación a objetos** (COBOL 2002) y **formato libre**, que elimina las columnas heredadas de la tarjeta perforada.
- **COBOL fuera del mainframe.** Micro Focus/OpenText Visual COBOL compila a **.NET** y a la **JVM**, de modo que el mismo código de negocio se ejecuta en un contenedor Linux o en Azure.
- **Herramientas actuales**: depuración desde **VS Code** (extensiones IBM Z Open Editor y COBOL de Micro Focus), Git, pipelines de CI/CD sobre z/OS, y asistentes de IA para traducir COBOL a Java.
- **Pruebas unitarias** con marcos como **zUnit** o **GnuCOBOL + cobol-check**: el COBOL también entró en la disciplina de las pruebas automatizadas.

⚙️ Cómo se ejecuta hoy

En producción: *IBM Enterprise COBOL for z/OS* (compilador de pago, integrado con CICS/Db2/IMS) y *Micro Focus / OpenText Visual COBOL* (COBOL sobre Linux, Windows, .NET y JVM).

Para aprender, gratis: GnuCOBOL, que traduce COBOL a C y lo compila con el compilador de C del sistema.

```
# Debian / Ubuntu
sudo apt-get install -y gnucobol

# Compilar a ejecutable, en formato libre (sin las columnas históricas)
cobc -x -free total.cob -o total

echo "15000 2 0.10" | ./total
# Total: 27000.00
```

Formato fijo vs. formato libre. El COBOL histórico hereda la **tarjeta perforada de 80 columnas**: las columnas 1–6 eran el número de secuencia, la 7 el indicador (un `*` ahí es un comentario, un `-` es continuación), las 8–11 el "Área A" donde van las divisiones y los niveles `01`, las 12–72 el "Área B" con las sentencias, y las 73–80 la identificación del programa. Ese formato sigue siendo el de la mayoría del código en producción. El **formato libre** (`-free`) lo elimina y es lo que verás en material nuevo. Si abres un fuente real y todo parece indentado de forma extraña, no es estilo: son columnas.

🧪 El programa de la clase 041 en COBOL

Mismo contrato que en toda la clase 041: leer `precio cantidad descuento` de una línea y escribir `Total: <total con 2 decimales>`.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TOTAL-VENTA.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  LINEA           PIC X(80).
01  TXT-PRECIO      PIC X(20).
01  TXT-CANTIDAD    PIC X(20).
01  TXT-DESCUENTO   PIC X(20).
01  PRECIO          PIC 9(9)V99   COMP-3.
01  CANTIDAD        PIC 9(9)V99   COMP-3.
01  DESCUENTO       PIC 9V9(4)    COMP-3.
01  TOTAL-CALC      PIC 9(12)V99  COMP-3.
01  TOTAL-EDITADO   PIC ZZZZZZZZZ9.99.

PROCEDURE DIVISION.
    ACCEPT LINEA
    UNSTRING LINEA DELIMITED BY ALL SPACES
        INTO TXT-PRECIO TXT-CANTIDAD TXT-DESCUENTO
    END-UNSTRING
    MOVE FUNCTION NUMVAL(TXT-PRECIO)    TO PRECIO
    MOVE FUNCTION NUMVAL(TXT-CANTIDAD)  TO CANTIDAD
    MOVE FUNCTION NUMVAL(TXT-DESCUENTO) TO DESCUENTO
    COMPUTE TOTAL-CALC ROUNDED = PRECIO * CANTIDAD * (1 - DESCUENTO)
    MOVE TOTAL-CALC TO TOTAL-EDITADO
    DISPLAY "Total: " FUNCTION TRIM(TOTAL-EDITADO)
    STOP RUN.
```

Recorrido, línea a línea.

- `PROGRAM-ID` es la identidad del módulo; en un mainframe ese nombre es también el del miembro en la librería de carga, y por eso suele estar limitado a 8 caracteres.
- Todas las variables se declaran en `WORKING-STORAGE`, **antes** de cualquier sentencia. No existe "declarar donde se usa": el `DATA DIVISION` es el inventario completo de la memoria del programa. Esa rigidez es la que hace que se pueda auditar un COBOL sin ejecutarlo.
- El nivel `01` marca un elemento de primer orden. Los niveles `05`, `10`… anidan campos dentro de una estructura, que es como COBOL representa un registro de fichero.
- `PIC 9(9)V99` se lee literalmente como una plantilla: nueve dígitos, una **coma decimal implícita** (`V` no ocupa espacio, solo dice dónde está el punto) y dos decimales. `COMP-3` pide que se guarde como decimal empaquetado. Nada de esto es float.
- `ACCEPT LINEA` lee una línea de la entrada estándar y la deja rellenada con espacios hasta 80.
- `UNSTRING ... DELIMITED BY ALL SPACES` es el `split` de COBOL: parte la línea por rachas de espacios y reparte los trozos en los tres campos de texto. `ALL` es lo que colapsa espacios consecutivos; sin él, dos espacios seguidos producirían un campo vacío.
- `FUNCTION NUMVAL` convierte el texto a número. Es una **función intrínseca**, incorporada al estándar en COBOL-85; antes había que escribir la conversión a mano.
- `COMPUTE ... ROUNDED` es la única forma de escribir una expresión aritmética completa; las formas verbales (`MULTIPLY A BY B GIVING C`) son las originales. `ROUNDED` aplica redondeo comercial al guardar en los dos decimales del destino, y es explícito **a propósito**: en dinero, redondear en silencio es un defecto.
- `TOTAL-EDITADO` con `PIC ZZZZZZZZZ9.99` es un **campo editado para presentación**: cada `Z` suprime un cero a la izquierda y lo sustituye por espacio, el `9` final garantiza que un total de cero se imprima como `0.00` y no como una cadena vacía. `FUNCTION TRIM` quita el relleno.

Fíjate en lo que **no** hay: ni tipo inferido, ni conversión implícita, ni formateo con `printf`. Cada transformación —texto a número, número a texto presentable— está escrita.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En COBOL es…
`x = 3`	`MOVE 3 TO X` — mover, no asignar; el verbo es deliberado
`total = a * b`	`COMPUTE TOTAL = A * B`
`const IVA = 0.19`	`01 IVA PIC 9V99 VALUE 0.19.` (y `CONSTANT` desde COBOL 2002)
`if/else if/else`	`EVALUATE ... WHEN ... WHEN OTHER ... END-EVALUATE`
`for` / `while`	`PERFORM VARYING I FROM 1 BY 1 UNTIL I > N`
Función	`PERFORM PARRAFO` (sección de código) o `CALL "SUBPROG" USING ...`
`struct` / registro	Niveles anidados `01 / 05 / 10`
`decimal` / `BigDecimal`	`PIC 9(9)V99 COMP-3` — nativo, no una librería
`split(línea)`	`UNSTRING`
`"%.2f" % x`	Un campo `PIC ZZZ9.99` al que se mueve el valor

⚠️ Errores comunes al leerlo

- **Confundir `V` con un punto.** `PIC 9(5)V99` ocupa 7 dígitos, no 8: la coma decimal es implícita y no existe en memoria. Un `PIC 9(5).99` sí imprime el punto, pero ya es un campo de presentación.
- **Creer que la indentación es estilo.** En formato fijo, mover una línea dos columnas puede sacarla del Área B y romper la compilación.
- **Leer `PERFORM` como una llamada a función.** `PERFORM` ejecuta un párrafo y vuelve, pero no crea ámbito ni recibe parámetros: todas las variables son globales al programa. Es la fuente número uno de sorpresas para quien viene de un lenguaje con funciones de verdad.
- **Asumir recursión.** El COBOL clásico no la permite: el `WORKING-STORAGE` es estático. Hace falta declarar el programa `RECURSIVE` (COBOL 2002) o usar `LOCAL-STORAGE`.
- **Ignorar el punto final.** El `.` cierra sentencias y, en el COBOL antiguo, delimita el alcance de un `IF`. Un punto de más dentro de un `IF` cambia el flujo del programa sin dar error.

📚 Fuentes y bibliografía

- IBM COBOL compilers (https://www.ibm.com/products/cobol-compilers) — familia de compiladores vigente para z/OS, AIX y Linux on Z.
- IBM Enterprise COBOL for z/OS — documentación (https://www.ibm.com/docs/en/cobol-zos) — referencia del lenguaje y guía de programación.
- GnuCOBOL (https://gnucobol.sourceforge.io/) — implementación libre, la que puedes instalar hoy.
- **Michael Coughlan**, *Beginning COBOL for Programmers*, Apress, 2014 — la entrada correcta si ya programas en otro lenguaje: asume que sabes programar y explica solo lo que COBOL hace distinto.
- **Mike Murach, Anne Prince, Raul Menendez**, *Murach's Mainframe COBOL* — el manual de referencia del COBOL empresarial con CICS, VSAM y Db2.
- **Gary DeWard Brown**, *z/OS JCL*, Wiley — porque el COBOL de producción no se ejecuta solo; ver también la ficha de JCL.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: PL/I · RPG · JCL

🧠 Lisp / Common Lisp — 1958

El segundo lenguaje de alto nivel de la historia, y el que se adelantó a todos. Recolección de basura, funciones de primera clase, tipado dinámico, REPL, código que se manipula a sí mismo: Lisp tenía todo eso cuando el resto del mundo programaba con tarjetas perforadas. Los demás lenguajes llevan sesenta años alcanzándolo.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Common Lisp se ejecuta hoy.*** *SBCL publica una versión al mes, Quicklisp distribuye miles de bibliotecas, y hay implementaciones comerciales con soporte (LispWorks, Allegro CL) vendiéndose a industria. Su variante* ***AutoLISP*** *viene incluida en AutoCAD, uno de los programas de ingeniería más usados del planeta. Y* ***Emacs Lisp*** *configura el editor de una parte considerable de los programadores del mundo.*
>
> *Entra porque* ***muestra dos conceptos que el núcleo entero no tiene****. El primero es la* ***homoiconicidad****: en Lisp el código y los datos tienen la misma forma —listas—, así que un programa puede construir y transformar programas con las mismas herramientas con que manipula una lista. De ahí salen las* ***macros****, que no son plantillas de texto como en C sino transformaciones reales del árbol sintáctico: en Lisp puedes* ***añadir sintaxis al lenguaje****. El segundo es el* ***sistema de condiciones y reinicios****, un manejo de errores estrictamente más potente que el* `try/catch` *que casi todos heredamos. Ver Lisp es entender qué le falta a los demás.*

Año	1958 (McCarthy); **Common Lisp** estandarizado en ANSI X3.226 en 1994
Autoría	**John McCarthy** (MIT); Common Lisp por un comité de la comunidad
Familia	**Lisp** — la raíz de Scheme, Racket, Clojure, Emacs Lisp y AutoLISP
Paradigma	Multiparadigma: funcional, imperativo y OO (CLOS, con **despacho múltiple**)
Tipado	**Dinámico y fuerte**, con declaraciones de tipo opcionales para optimizar
Memoria	Recolector de basura — Lisp fue donde se inventó
Ejecución	Compilado a nativo en tiempo de ejecución; REPL siempre disponible
Estado	🟡 **Nicho vivo** — IA simbólica, CAD, sistemas con mucho DSL, investigación

---

📜 Historia

En 1958 **John McCarthy** trabajaba en el MIT sobre problemas de inteligencia artificial y necesitaba manipular **expresiones simbólicas**, no números. Su artículo de 1960, *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, define un lenguaje matemático basado en el cálculo lambda y en una sola estructura de datos: la **lista**.

McCarthy había definido, como ejercicio teórico, una función `eval` que interpretaba expresiones del propio lenguaje. Su estudiante **Steve Russell** se dio cuenta de algo que McCarthy no había previsto: esa `eval` se podía **implementar en código máquina**, y entonces habría un intérprete de Lisp de verdad. Lo hizo, y así nació el primer Lisp ejecutable — junto con la idea del REPL y de la programación interactiva.

Lisp introdujo o popularizó, décadas antes que nadie:

- La **recolección de basura automática**.

- Las **funciones como valores de primera clase** y las de orden superior.
- La **recursión** como herramienta central.
- El **tipado dinámico** con comprobación en tiempo de ejecución.
- El **REPL** y el desarrollo interactivo.
- Las **macros** y la metaprogramación sintáctica.
- La **condicional** `cond` — sí, el `if/else if` viene de aquí.

Durante los 70 y 80 proliferaron dialectos: MacLisp, InterLisp, ZetaLisp, Scheme (1975, minimalista y académico). Se llegó a construir hardware específico —las **máquinas Lisp** de Symbolics y LMI— con el lenguaje como sistema operativo. Cuando la financiación de la IA se desplomó a finales de los 80 (el llamado "invierno de la IA"), aquella industria se hundió con ella.

Para frenar la fragmentación, un comité produjo en 1984 *Common Lisp the Language* y en **1994** el estándar **ANSI Common Lisp**: un lenguaje grande, deliberadamente pragmático, que unificó los dialectos e incorporó **CLOS** (*Common Lisp Object System*), probablemente el sistema de objetos más potente jamás estandarizado.

🏭 Dónde sobrevive hoy

- **CAD e ingeniería**: **AutoLISP** dentro de AutoCAD — con diferencia, el Lisp con más usuarios del mundo, aunque casi ninguno se llame a sí mismo "programador Lisp".
- **Configuración y extensión de herramientas**: **Emacs Lisp**; el editor Emacs es, literalmente, un intérprete de Lisp con un editor escrito encima.
- **Sistemas con reglas complejas**: motores de planificación, sistemas expertos, optimización combinatoria. El caso históricamente documentado es **ITA Software**, el motor de búsqueda de vuelos escrito en Common Lisp que compró Google y que está detrás de Google Flights.
- **Investigación**: procesamiento del lenguaje, razonamiento simbólico, demostradores de teoremas, y como lenguaje de implementación de otros lenguajes.
- **Aeroespacial**: el *Remote Agent* de la NASA que controló autónomamente la sonda **Deep Space 1** en 1999 estaba escrito en Common Lisp; el equipo llegó a **depurar el software en vuelo, a 150 millones de kilómetros, conectándose al REPL de la nave**. Es probablemente la mejor anécdota que existe sobre desarrollo interactivo.

🧠 Por qué no ha muerto

1. Las macros permiten construir el lenguaje que el problema necesita. Como el código es una lista, una macro recibe la estructura del código y devuelve otra estructura, antes de compilar. En un dominio complejo puedes crear notación propia y escribir la solución en ella. En otros lenguajes, eso exige un preprocesador, un generador de código o un DSL externo con su parser.

2. El sistema de condiciones es superior al `try/catch`. Cuando una función de Lisp señala un error, el manejador se ejecuta **antes de desenrollar la pila**, y puede elegir entre varios **reinicios** (*restarts*) que la función que falló dejó disponibles: reintentar, usar otro valor, saltar el elemento. En Java o Python, para cuando atrapas la excepción, el contexto ya se destruyó y la única opción es reintentar todo desde fuera. PL/I tuvo una idea parecida en 1964; casi nadie más la siguió.

3. Desarrollo interactivo real. Con SLIME/Sly sobre Emacs puedes recompilar una única función dentro de una imagen en ejecución, con el estado intacto. Es el mismo modelo de trabajo vivo de Smalltalk.

4. CLOS y el despacho múltiple. Un método de CLOS se selecciona según los tipos de **todos** sus argumentos, no solo del receptor. Eso resuelve limpiamente el problema del "doble despacho" que en Java o C# obliga al patrón *Visitor*. Además, el **Protocolo de Metaobjetos (MOP)** permite modificar cómo funciona la propia orientación a objetos.

5. Es rápido. SBCL compila a código nativo. La fama de "lenguaje lento e interpretado" viene de los años 70 y hoy es simplemente falsa.

🔄 Lo que se ha modernizado

El estándar ANSI está congelado desde 1994 —eso es una decisión, no un abandono—, pero **todo lo que rodea al lenguaje se ha renovado**:

- **SBCL publica una versión al mes**, con mejoras continuas de compilador y recolector de basura. Compila a nativo y es competitivo en rendimiento con lenguajes mucho más jóvenes.
- **Gestión de dependencias moderna**: **Quicklisp** para bibliotecas, y **Qlot** o **CLPM** para fijar versiones por proyecto, que es lo que faltaba para reproducibilidad real.
- **Web y servicios**: **Clack** y **Hunchentoot** como servidores, **Jonathan** y **Yason** para JSON, **Dexador** como cliente HTTP. Construir una API REST en Common Lisp es rutinario.
- **Coalton**: un lenguaje con **tipado estático al estilo ML**, con inferencia y tipos algebraicos, implementado **como una biblioteca de macros** sobre Common Lisp e interoperable con él. Es la mejor demostración posible de para qué sirven las macros: añadir un sistema de tipos a un lenguaje dinámico sin cambiar el compilador.
- **Herramientas de edición actuales**: además de Emacs con SLIME/Sly, hay extensiones para **VS Code** (Alive) y para IntelliJ.
- **Uso reciente documentado**: la computación cuántica de Rigetti desarrolló su compilador **Quilc** y su simulador en Common Lisp, precisamente por la manipulación simbólica.

⚙️ Cómo se ejecuta hoy

```
sudo apt-get install -y sbcl

sbcl --script total-venta.lisp < entrada.txt

# El REPL, que es donde de verdad se trabaja:
sbcl
* (+ 1 2)
3
```

Implementaciones: SBCL (*Steel Bank Common Lisp*, libre, la más usada y la más rápida), **CCL** (Clozure), **ECL** (empotrable en C), **ABCL** (sobre la JVM), y las comerciales **LispWorks** y **Allegro CL** (Franz), con IDE, entrega de aplicaciones y soporte.

Ecosistema: Quicklisp (https://www.quicklisp.org/) es el gestor de bibliotecas de facto y **ASDF** el sistema de construcción. Para editar, **Emacs + SLIME/Sly** sigue siendo la experiencia de referencia, aunque hay extensiones para VS Code (Alive) y un plugin para IntelliJ.

🧹 El programa de la clase 041 en Common Lisp

```
;;; total-venta.lisp — clase 041
(setf *read-default-float-format* 'double-float)

(let* ((precio    (read))
       (cantidad  (read))
       (descuento (read))
       (total     (* precio cantidad (- 1 descuento))))
  (format t "Total: ~,2F~%" total))
```

Recorrido, línea a línea.

- **Todo son paréntesis, y la regla es una sola:** `(operador argumento argumento ...)`. El primer elemento de la lista es lo que se aplica; el resto, sus argumentos. `(* precio cantidad)` es multiplicación en notación prefija. Esa uniformidad —conocida como *s-expresiones*— es exactamente lo que hace posible que un programa manipule programas: no hay sintaxis especial que analizar.
- `*read-default-float-format*` es una **variable global especial** (la convención `*asteriscos*` las marca). Le decimos al lector que los literales reales sean de doble precisión. Fíjate en que estamos **configurando el analizador sintáctico del lenguaje desde el propio programa**; en la mayoría de los lenguajes eso ni siquiera es expresable.
- `(read)` es una función asombrosa por lo que hace: lee de la entrada estándar **una expresión de Lisp** y devuelve el objeto correspondiente. `15000` llega como entero, `0.10` como real. No hay `split`, ni `parseFloat`, ni conversión de tipos: el lector del lenguaje es también el analizador de la entrada. Es el mismo `read` que usa el REPL.
- `let*` liga variables locales **en secuencia**, de modo que `total` puede usar `precio`, `cantidad` y `descuento` ya calculados. Su hermano `let` liga en paralelo y no lo permitiría. Que exista esa distinción explícita es típico de Lisp: el orden de evaluación no se deja a la costumbre.
- `(- 1 descuento)` es `1 - descuento`. Cuesta un rato acostumbrarse, y a cambio no hay que memorizar ninguna tabla de precedencia de operadores: la estructura del paréntesis **es** la precedencia.
- `format` es un mini-lenguaje de plantillas dentro del lenguaje. `~,2F` significa "real con 2 decimales" y `~%` es el salto de línea. El primer argumento `t` es el destino (la salida estándar). `format` es notoriamente potente: tiene condicionales, iteración sobre listas y hasta pluralización.

Y ahora el motivo real de estudiarlo. Supón que quieres una notación propia para las reglas de precios de tu negocio. En Lisp la creas:

```
(defmacro con-descuento ((var porcentaje) &body cuerpo)
  "Ejecuta CUERPO con VAR ligada al factor multiplicador del descuento."
  `(let ((,var (- 1 (/ ,porcentaje 100))))
     ,@cuerpo))

(con-descuento (factor 10)
  (format t "Total: ~,2F~%" (* 15000 2 factor)))
;; => Total: 27000.00
```

`con-descuento` **no es una función**: es una macro que se ejecuta en tiempo de compilación y **reescribe el código** antes de compilarlo. El acento grave abre una plantilla, la coma inserta un valor y `,@` inserta una lista de expresiones. El resultado es una construcción sintáctica nueva, indistinguible de las que trae el lenguaje. En Python o Java, para conseguir esto harías falta un generador de código externo; en Lisp es una función más que devuelve listas, porque **el código es una lista**.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces...	En Common Lisp es...
`a + b * c`	`(+ a (* b c))` — prefijo, sin precedencia que memorizar
`x = 5`	`(setf x 5)` — y `setf` funciona sobre cualquier "lugar"
Variable local	`(let ((x 1) (y 2)) ...)`
`def f(x): ...`	`(defun f (x) ...)`
Lambda	`(lambda (x) ...)`
`if / elif / else`	`(if c a b)` o `(cond (c1 ...) (c2 ...) (t ...))`
`switch`	`(case x (1 ...) (2 ...) (otherwise ...))`
Clase con métodos	`defclass` + `defgeneric` + `defmethod` — **los métodos viven fuera de la clase**
`try / catch / finally`	`handler-case`, `handler-bind` + `restart-case`, `unwind-protect`
Anotaciones / decoradores	`defmacro` — pero de verdad
`null`	`nil`, que además es la lista vacía y el falso lógico

⚠️ Errores comunes al leerlo

- **Contar paréntesis a mano.** No se hace: se usa un editor con emparejamiento y edición estructural (paredit, smartparens). Quien lucha con los paréntesis es siempre quien no ha configurado el editor.
- **Confundir `list` con `quote`.** `(list 1 2)` evalúa sus argumentos; `'(1 2)` devuelve la lista literal sin evaluar. La comilla es la puerta entre "código" y "datos".
- **Creer que `nil` y `false` son cosas distintas.** `nil` es a la vez la lista vacía, el falso lógico y el símbolo `nil`. Cualquier otro valor es verdadero, incluido `0`.
- **Escribir macros donde bastaría una función.** El consejo clásico de la comunidad: una macro solo se justifica cuando necesitas controlar **si** o **cuándo** se evalúan los argumentos.
- **Comparar con `=`.** `=` es solo para números. Para el resto están `eq`, `eql`, `equal` y `equalp`, en orden creciente de laxitud; elegir mal es la fuente clásica de errores sutiles.
- **Asumir que es interpretado y lento.** SBCL compila a nativo, y con declaraciones de tipo puede acercarse a C.

📚 Fuentes y bibliografía

- Common Lisp HyperSpec (https://www.lispworks.com/documentation/HyperSpec/Front/) — el estándar ANSI en formato navegable; la referencia definitiva.
- SBCL (https://www.sbcl.org/) y Quicklisp (https://www.quicklisp.org/) — la implementación y el gestor de bibliotecas que usarás.
- Common Lisp Cookbook (https://lispcookbook.github.io/cl-cookbook/) — recetas prácticas y actuales.
- **Peter Seibel**, *Practical Common Lisp*, Apress — gratis en línea (https://gigamonkeys.com/book/); la mejor entrada para quien ya programa.
- **Paul Graham**, *On Lisp* — el libro sobre macros; descarga gratuita del autor (http://www.paulgraham.com/onlisp.html).
- **Peter Norvig**, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann — construye sistemas de IA clásica desde cero; uno de los mejores libros de programación jamás escritos, y liberado por el autor.
- **Sonya Keene**, *Object-Oriented Programming in Common Lisp* — la referencia sobre CLOS.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: AutoLISP · Smalltalk · PL/I

🚀 C++ — 1985

Como C, está en esta lista sin ser legacy. C++ tiene cuarenta años y una revisión del estándar cada tres. Lo que ejecuta el videojuego que juegas, el navegador con el que lees esto, el motor de la base de datos que guarda tus datos y el sistema que ejecuta una orden en bolsa en microsegundos está, con altísima probabilidad, escrito en C++.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: C++ está completamente vigente**, con estándar nuevo cada tres años (C++20, C++23, C++26 en camino) y presencia dominante en varios dominios donde no tiene rival práctico.*
>
> *Entra porque **encarna un concepto que ningún otro lenguaje del núcleo expresa igual**: la **abstracción de coste cero**. La promesa de C++ es que puedes escribir código de alto nivel —plantillas, objetos, algoritmos genéricos— y obtener el mismo código máquina que habrías escrito a mano en C. Y su segunda gran idea, **RAII**, resuelve la gestión de recursos atándola al tiempo de vida de un objeto: es la respuesta que precede al ownership de Rust y al `defer` de Go, y entenderla ilumina a las dos. C++ ya aparece como primo de la familia C en cada `primos.md` del programa; esta ficha es su historia y su porqué.*

Año	1979 como "C con clases"; **1985** como C++; ISO desde 1998
Autoría	**Bjarne Stroustrup**, Bell Labs
Familia	C / llaves
Paradigma	Multiparadigma: procedimental, OO, **genérico** y funcional
Tipado	Estático, fuerte, con inferencia (`auto`) y metaprogramación de tipos
Memoria	**Manual con RAII**: punteros inteligentes, destructores deterministas
Ejecución	Compilado a nativo
Estado	🟢 **Completamente vigente** — videojuegos, navegadores, finanzas, HPC, robótica

---

📜 Historia

Bjarne Stroustrup había trabajado con **Simula 67**, el lenguaje que inventó las clases, y le había convencido su capacidad para modelar problemas. También le había desesperado su lentitud. Cuando en 1979 llegó a Bell Labs, quiso las abstracciones de Simula **con el rendimiento y el acceso al hardware de C**. Empezó con un preprocesador llamado *C con clases*, y en **1985** publicó el lenguaje con el nombre **C++** —el operador de incremento de C, un chiste que se convirtió en marca.

La regla de diseño que ha gobernado el lenguaje desde entonces es explícita y explica casi todo lo demás: **no debes pagar por lo que no usas**. Si no utilizas excepciones, no cuestan; si no usas funciones virtuales, no hay tabla de despacho. Esa promesa es lo que le permitió entrar donde ningún otro lenguaje de alto nivel podía.

La cronología moderna se divide en dos épocas:

- **Hasta C++03**: crecimiento acumulativo. En 1998 llegó el primer estándar ISO, con la **STL** de Alexander Stepanov —contenedores, iteradores y algoritmos genéricos separados entre sí— que fue una aportación conceptual de primer orden y sigue influyendo en el diseño de bibliotecas de todos los lenguajes.

- **Desde C++11**: renacimiento. `auto`, lambdas, **semántica de movimiento**, punteros inteligentes, `nullptr`, bucles `for` sobre rangos, hilos en la biblioteca estándar. Stroustrup lo resumió así: *"C++11 se siente como un lenguaje nuevo."* Desde entonces el comité publica una revisión **cada tres años**: C++14, C++17, C++20, C++23, y C++26 en preparación.

La crítica constante es su tamaño: el estándar supera las mil quinientas páginas y hay más de un "C++" posible según el subconjunto que se elija. Es una crítica justa, y la comunidad responde con guías —las **C++ Core Guidelines** de Stroustrup y Herb Sutter— que definen qué subconjunto usar.

🏭 Dónde vive hoy

- **Videojuegos y gráficos**: **Unreal Engine**, el núcleo de **Unity**, motores propios de estudio, y los controladores gráficos por debajo.
- **Navegadores**: **Chromium/Blink**, **Firefox/Gecko**, **WebKit**, y los motores de JavaScript **V8** y **SpiderMonkey** — es decir, tu JavaScript se ejecuta dentro de C++.
- **Bases de datos**: MySQL, MongoDB, ClickHouse, RocksDB, el motor de SQL Server.
- **Finanzas de baja latencia**: sistemas de negociación donde importan los microsegundos.
- **HPC e IA**: los núcleos de cómputo de **PyTorch** y **TensorFlow** son C++ con CUDA; Python es la interfaz.
- **Robótica y automoción**: **ROS 2**, **AUTOSAR Adaptive**, conducción asistida, visión artificial.
- **Aplicaciones de escritorio**: Adobe Photoshop, la suite de Autodesk, gran parte de Windows y de Office, **Qt** como marco multiplataforma.
- **Infraestructura**: **LLVM** (el compilador que usan Rust, Swift y Clang está escrito en C++).

🧠 Por qué no es legacy

1. Nadie más ocupa ese punto exacto. C++ es el único lenguaje ampliamente disponible que combina control total sobre la memoria y el hardware, abstracción de alto nivel sin sobrecoste, y un ecosistema maduro en gráficos, cálculo científico y sistemas. Rust se acerca por un lado, pero su ecosistema en dominios como los motores gráficos o la instrumentación científica está a décadas de distancia.

2. Rendimiento predecible sin recolector de basura. En un motor de videojuego que debe entregar un fotograma cada 16 milisegundos, o en un sistema de trading, una pausa de recolección es inaceptable. **RAII** —destructores deterministas ligados al ámbito— da liberación automática de recursos **sin** pausas impredecibles.

3. La metaprogramación con plantillas es única. Permite generar código especializado en tiempo de compilación: bibliotecas como **Eigen** producen operaciones con matrices tan rápidas como el ensamblador escrito a mano, a partir de expresiones legibles. Con `constexpr` y `consteval`, hoy se puede ejecutar prácticamente cualquier cálculo durante la compilación.

4. Interoperabilidad total con C y, a través de ella, con todo lo demás.

5. Compatibilidad hacia atrás casi obsesiva. Código de 1998 sigue compilando. Es su mayor lastre —arrastra construcciones que nadie recomienda— y a la vez la razón de que la industria haya podido apostar por él durante treinta años sin reescribir.

🔄 Lo que se ha modernizado

El C++ que se escribe hoy no se parece al de 1998, y esa es la parte que más se ignora desde fuera:

- **Gestión de memoria sin `new` / `delete` a mano.** `std::unique_ptr` y `std::shared_ptr` con `make_unique` / `make_shared` cubren casi todos los casos. La guía moderna es sencilla: **si escribes `delete`, probablemente te has equivocado**.

- **Semántica de movimiento** (C++11): transferir la propiedad de un recurso en lugar de copiarlo. Es el antepasado directo del sistema de *ownership* de Rust.
- `constexpr` y `consteval`: cálculo en tiempo de compilación con sintaxis normal, en lugar de metaprogramación de plantillas ilegible.
- **Conceptos** (C++20): restricciones sobre los parámetros de plantilla que producen mensajes de error comprensibles. Resuelven el problema más doloroso del C++ genérico clásico.
- **Rangos** (C++20): componer algoritmos con `|`, al estilo de las tuberías funcionales, sin iteradores explícitos.
- **Módulos** (C++20): sustituto de `#include`, que elimina la inclusión textual y acelera drásticamente la compilación. Su adopción va despacio, pero es el cambio estructural más grande del lenguaje.
- **Corrutinas** (C++20) y `std::format` (C++20) / `std::print` (C++23), que por fin dan un formateo seguro con tipos, al estilo de Python.
- **Herramientas**: *sanitizers*, `clang-tidy`, análisis estático, **vcpkg** y **Conan** como gestores de paquetes, y CMake como estándar de facto para construir.
- **Y una respuesta explícita al debate de seguridad de memoria**: los perfiles de seguridad propuestos para C++26 y las **C++ Core Guidelines** buscan garantizar por construcción propiedades que Rust obtiene del compilador. Es un trabajo en curso y merece seguirse.

⚙️ Cómo se ejecuta hoy

```
g++ -std=c++23 -Wall -Wextra -O2 total.cpp -o total
clang++ -std=c++23 -fsanitize=address,undefined -g total.cpp -o total

echo "15000 2 0.10" | ./total
# Total: 27000.00
```

Compiladores: GCC, **Clang/LLVM**, **MSVC** e **Intel oneAPI**. **Construcción:** CMake, Meson, Bazel. **Dependencias:** vcpkg, Conan.

🍼 El programa de la clase 041 en C++

```
#include <iomanip>
#include <iostream>

int main() {
    double precio{}, cantidad{}, descuento{};

    if (!(std::cin >> precio >> cantidad >> descuento)) {
        return 1;
    }

    const double total = precio * cantidad * (1 - descuento);

    std::cout << "Total: " << std::fixed << std::setprecision(2) << total << '\n';
    return 0;
}
```

Recorrido, y la comparación que importa.

- `std::cin >> precio` usa **sobrecarga de operadores**: `>>` está definido para cada tipo, así que la misma línea lee un `double`, un `int` o una cadena según lo que reciba. Compara con el `scanf("%lf", &precio)` de C: allí el formato y el tipo se declaran por separado y **nada comprueba que coincidan**; aquí el tipo lo decide el compilador. Es el mismo problema resuelto con seguridad de tipos.

- `double precio{}` es **inicialización uniforme con llaves** (C++11). Además de inicializar a cero, prohíbe las conversiones que pierden información (*narrowing*), que con `=` pasarían en silencio. Es un ejemplo pequeño y muy representativo de la dirección del lenguaje moderno: cerrar puertas que C dejó abiertas, sin coste en ejecución.
- `if (!(std::cin >> ...))` funciona porque el flujo se convierte a booleano según su estado. Igual que en C hay que comprobar `scanf`, aquí hay que comprobar el flujo — con la diferencia de que si no lo haces, las variables ya están inicializadas a `0` gracias a `{}`, en lugar de contener basura.
- `std::fixed` y `std::setprecision(2)` son **manipuladores**: objetos que modifican el estado del flujo. Son notoriamente incómodos —el estado persiste para las siguientes escrituras—, y por eso C++20 introdujo `std::format` y C++23 `std::print`:

```
cpp #include <print> std::print("Total: {:.2f}\n", total); // C++23
```

Esa línea es el C++ que se escribirá a partir de ahora: seguro con los tipos, legible y sin estado global. Merece la pena verla al lado de la anterior para entender hacia dónde va el lenguaje. - **Y lo que no aparece porque no hace falta**: no hay `free`, ni `delete`, ni fugas posibles. El ámbito gestiona todo. Eso es **RAII**, y es la idea que hay que llevarse.

⚠️ Errores comunes al leerlo

- **Confundir "C++" con "C con clases".** Escribir C++ como si fuera C —punteros crudos, `new` / `delete`, arrays de C— es la fuente principal de su mala fama. El C++ moderno es otro lenguaje.
- `using namespace std;` **en una cabecera.** Contamina el espacio de nombres de todo quien la incluya. En un `.cpp` pequeño es discutible; en un `.hpp` es un error.
- **Copias silenciosas.** Pasar un `std::vector` por valor copia todos sus elementos. Se pasa por referencia constante (`const std::vector<T>&`) salvo que quieras la copia.
- **Ignorar la regla de cero/tres/cinco.** Si una clase gestiona un recurso, necesita destructor, constructor de copia, asignación, movimiento y asignación por movimiento — o mejor, ninguno de los cinco, delegando en tipos que ya lo hacen (**la regla de cero**).
- **Suponer que las plantillas son genéricos de Java.** Se instancian en compilación y generan código distinto por cada tipo. Eso da rendimiento y también tiempos de compilación largos y errores descomunales (hasta que llegaron los conceptos).
- **Leer código antiguo como si fuera actual.** Fíjate siempre en el estándar con que se compila: `-std=c++98` y `-std=c++23` son lenguajes distintos.

📚 Fuentes y bibliografía

- cppreference.com (https://en.cppreference.com/w/) — la mejor referencia del lenguaje y la biblioteca; imprescindible.
- isocpp.org (https://isocpp.org/) — sitio oficial del estándar, con las FAQ de Stroustrup.
- C++ Core Guidelines (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines) — Stroustrup y Sutter definiendo qué subconjunto del lenguaje usar y por qué. La respuesta práctica a "C++ es demasiado grande".
- **Bjarne Stroustrup**, *A Tour of C++*, 3.ª ed., Addison-Wesley — el mejor punto de entrada si ya programas: el lenguaje moderno completo en unas doscientas páginas, por su autor.
- **Bjarne Stroustrup**, *The C++ Programming Language*, 4.ª ed. — la referencia extensa.
- **Scott Meyers**, *Effective Modern C++*, O'Reilly — cuarenta y dos consejos sobre C++11/14; sigue siendo el libro que convierte a un programador de C++ en uno bueno.
- **Nicolai Josuttis**, *C++ Move Semantics* y *C++20: The Complete Guide* — para las dos piezas más difíciles del lenguaje moderno.
- **Bjarne Stroustrup**, *The Design and Evolution of C++* — por qué el lenguaje es como es, contado por quien tomó las decisiones. Excelente lectura sobre diseño de lenguajes en general.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: C · Assembler · Delphi / Object Pascal

🟪 C# — 2000

C# nació como la respuesta de Microsoft a Java y acabó adelantándolo en casi todo lo que significa "características del lenguaje". Es, probablemente, **el lenguaje mayoritario que más rápido ha incorporado ideas de la investigación**: genéricos reificados, consultas integradas, `async` / `await` y tipos que distinguen si algo puede ser nulo.

> 🎯 ***Por qué está en este programa***
>
> ***C# es uno de los diez lenguajes del núcleo*** *y el* ***representante de la familia .NET*** *(Atlas): quien entiende la versión C# de una clase reconoce después F# y VB.NET, que comparten con él el mismo tiempo de ejecución, la misma biblioteca y los mismos tipos.*
>
> *Y aporta al programa dos cosas que ningún otro del núcleo enseña igual:* ***los genéricos reificados*** *— los tipos existen en ejecución, a diferencia de Java (clase 108)— y* ***el origen de*** `async` / `await`*, que C# inventó en 2012 y que hoy está en JavaScript, Python, Rust, Kotlin y C++ (clase 134).*

Año	2000; **2.0** con genéricos (2005); **5.0** con `async` (2012); anual desde 2019
Autoría	**Anders Hejlsberg**, Microsoft — antes Turbo Pascal y Delphi
Familia	.NET; sintaxis de C, arquitectura influida por Java y por Delphi
Paradigma	Multiparadigma: OO, funcional, imperativo y —con LINQ— declarativo
Tipado	**Estático, nominal y fuerte**; genéricos **reificados**; nulabilidad comprobada
Memoria	Recolección de basura, con `struct` de valor y `Span<T>` sin copia
Ejecución	Bytecode (**IL**) sobre el CLR, con JIT; también AOT nativo
Estado	🟢 **Muy usado**, multiplataforma y de código abierto desde 2016

📜 Historia

Microsoft tenía en los noventa un problema con Java: quería usarlo y quería extenderlo, Sun se opuso, y de ahí salió **un pleito** y la decisión de hacer una plataforma propia.

En **2000** presentó **.NET** y **C#**, diseñado por **Anders Hejlsberg** — la misma persona que había hecho Turbo Pascal y Delphi, y que después dirigiría TypeScript. Esa continuidad se nota: propiedades, eventos y delegados vienen directamente de Delphi (clase 120).

Y desde el principio hubo una decisión que lo separó de Java y que resultó ser la correcta: **el CLR se diseñó para varios lenguajes**, con un sistema de tipos común. Por eso VB.NET, F# y C# comparten biblioteca e interoperan sin fricción — que es lo que la clase 157 llama **ABI definido por la plataforma**, igual que ILE en IBM i.

Los hitos:

- **C# 2.0 (2005)**: **genéricos reificados** —el CLR los conoce en ejecución, a diferencia del borrado de Java — y métodos anónimos.

- **C# 3.0 (2007)**: **LINQ**, expresiones lambda, tipos anónimos, `var`, métodos de extensión. LINQ llevó las consultas al lenguaje, con comprobación de tipos (clase 170).
- **C# 5.0 (2012)**: `async` / `await`. Es la aportación más influyente de C# a la industria (clase 134).
- **2016**: **.NET Core** — multiplataforma, de código abierto y sin dependencia de Windows. El giro que salvó a la plataforma.
- **C# 8 (2019)**: **tipos de referencia nulables**, que separan `string` de `string?` y hacen comprobable el error del billón de dólares.
- **C# 9-13**: registros, emparejamiento de patrones, `init`, programas de nivel superior, `required`, colecciones literales.

🏭 Dónde vive hoy

- **Software empresarial en Windows y, cada vez más, en Linux**: ASP.NET Core mueve una parte grande de la web corporativa.
- **Videojuegos**: **Unity** usa C# como lenguaje de guion; es probablemente el mayor volumen de C# escrito por número de personas.
- **Aplicaciones de escritorio**: WPF, WinUI y MAUI — el nicho que en su día ocupó Delphi.
- **Servicios en la nube**: Azure Functions, servicios de fondo, integraciones.
- **Herramientas y automatización**: PowerShell está construido sobre .NET (ficha PowerShell).

🧠 Lo que enseña y no enseña ningún otro del núcleo

Uno, genéricos reificados (clase 108). En Java, `List<String>` y `List<Integer>` **son el mismo tipo en ejecución**: los genéricos se borran. En C#, no:

```
var lista = new List<string>();
Console.WriteLine(lista.GetType());        // System.Collections.Generic.List`1[System.String]
```

El CLR conoce el argumento de tipo, así que se puede preguntar por él, serializar con él y especializar el código para `List<int>` sin boxing — lo que tiene consecuencias directas de rendimiento (clase 128).

Dos, el origen de `async` / `await` (clase 134). C# lo introdujo en 2012 sobre la máquina de estados que el compilador genera, y **de ahí lo copiaron casi todos**: TypeScript, Python, Rust, Kotlin, Swift y C++20.

Y tres, LINQ, que es la respuesta más elegante que existe al desajuste de impedancia de la clase 170:

```
var caros = from p in productos
            where p.Precio > 100
            orderby p.Nombre
            select new { p.Nombre, p.Precio };
```

Esa consulta se comprueba en compilación, y **se puede traducir a SQL** por un proveedor —Entity Framework— o ejecutarse en memoria. Es la misma idea que las consultas como estructura de Lisp y que `sqlpp11` en C++: **la consulta es un árbol tipado, no una cadena** (clase 153).

🔄 Lo que se ha modernizado

- **Multiplataforma y de código abierto**: .NET 8/9/10 corren en Linux, macOS, ARM y contenedores.
- **AOT nativo**: binario sin máquina virtual, con arranque de milisegundos y una imagen mínima (clase 174) — la respuesta al problema de arranque de las plataformas gestionadas.
- `Span<T>` y `Memory<T>`: trabajar con memoria **sin copiar y sin recolector**, lo que ha llevado a C# a competir en escenarios de baja latencia (clase 152).

- **Tipos nulables comprobados**, registros y emparejamiento de patrones — la influencia de F# sobre su hermano.
- **Interoperabilidad moderna**: `LibraryImport` con generación de código en vez de reflexión, y punteros de función (clase 156).

⚙️ Cómo se ejecuta hoy

```
dotnet run < entrada.txt                    # el comando de la clase 041
dotnet build -c Release
dotnet publish -c Release -p:PublishAot=true    # binario nativo (clase 174)

dotnet format && dotnet test                 # calidad y pruebas (clases 139 y 146)
```

🪠 El programa de la clase 041 en C#

```
using System;
using System.Globalization;

// C# sobre el CLR: tipado estático con cultura invariante para el formato.
string[] p = Console.In.ReadToEnd()
    .Split(new[] { ' ', '\t', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

double precioUnitario = double.Parse(p[0], CultureInfo.InvariantCulture);
int cantidad = int.Parse(p[1], CultureInfo.InvariantCulture);
double descuento = double.Parse(p[2], CultureInfo.InvariantCulture);

double subtotal = precioUnitario * cantidad;
double total = subtotal * (1 - descuento);

Console.WriteLine("Total: " + total.ToString("F2", CultureInfo.InvariantCulture));
```

Lo que hay que ver.

- **No hay `class Program` ni `static void Main`.** Son los *programas de nivel superior* de C# 9: la misma ceremonia que Java exige, eliminada. La comparación entre los dos fragmentos es la historia de veinte años de evolución.
- **`CultureInfo.InvariantCulture` aparece cuatro veces, y no sobra ninguna.** En una máquina con configuración española, `double.Parse("15000.0")` **falla** —espera coma— y `ToString("F2")` imprimiría `27000,00`. Es la trampa clásica de .NET, y la misma que `Locale.US` en Java.
- **`int` y `double` son alias de `System.Int32` y `System.Double`**: en C#, a diferencia de Java, **los primitivos también son tipos del sistema de tipos**, así que `5.ToString()` es válido.
- **`ReadToEnd` en lugar de leer una línea** es una decisión deliberada del ejemplo: hace el programa robusto ante la forma en que llegue la entrada, que es lo que el contrato de la clase 040 exige.

📚 Fuentes y bibliografía

- Documentación de C# y .NET (https://learn.microsoft.com/dotnet/csharp/) — y las notas de versión del lenguaje (https://learn.microsoft.com/dotnet/csharp/whats-new/), que explican cada característica con su motivación.
- Propuestas del lenguaje (GitHub) (https://github.com/dotnet/csharplang) — el diseño en abierto.
- **Jon Skeet**, *C# in Depth*, 4.ª ed., Manning — el libro que explica **por qué** el lenguaje es como es; imprescindible para genéricos, `async` y LINQ.
- **Joseph Albahari**, *C# 12 in a Nutshell*, O'Reilly — la referencia completa, actualizada cada año.
- **Andrew Troelsen, Phil Japikse**, *Pro C# with .NET* — cobertura amplia de la plataforma.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: F# · VB.NET · Java · Delphi · TypeScript

🅳 D — 2001

D es **el C++ que Walter Bright habría hecho si hubiera podido empezar de cero**, y lo dice con autoridad: Bright escribió el primer compilador nativo de C++ del mundo. Es un lenguaje excelente, técnicamente adelantado a su tiempo en varias cosas, y **el caso de estudio más claro de que la calidad técnica no basta** (clase 164).

> 🎯 ***Por qué está en este programa***
>
> *D es un **primo de la familia C / llaves** (Atlas), cuyo representante en el núcleo es C.*
>
> *Aporta al programa una idea que hoy está en todas partes y que D tuvo primero: **ejecutar código del propio lenguaje en tiempo de compilación** —CTFE, de 2007—, que es lo que Zig llama `comptime`, C++ llama `constexpr` y Rust llama `const fn` (clase 122). Y aporta el ejemplo más didáctico de **cómo una decisión de gobernanza puede hundir un lenguaje**.*

Año	2001; **D2** en 2007, incompatible con D1; estable desde ~2010
Autoría	**Walter Bright** (autor del compilador Zortech C++), con **Andrei Alexandrescu**
Familia	C / llaves; rediseño de C++ con influencia de Java y Python
Paradigma	Multiparadigma: imperativo, OO, funcional, genérico y metaprogramación
Tipado	**Estático y fuerte**, con inferencia, `immutable` y `pure` comprobados
Memoria	**Recolector opcional**: se puede usar, evitar (`@nogc`) o gestionar a mano
Ejecución	Compilado a nativo — tres compiladores: DMD (rápido), LDC (LLVM), GDC (GCC)
Estado	🟡 **Vivo y minoritario**: excelente, con adopción industrial pequeña

---

📜 Historia

Walter Bright había escrito **Zortech C++ (1987)**, el primer compilador de C++ que generaba código nativo directamente en vez de traducir a C. Nadie conocía mejor las esquinas del lenguaje ni sus problemas.

En **1999** empezó **D**: mantener lo bueno de C++ —rendimiento, acceso al sistema, control— y **quitar la carga histórica**: el preprocesador, la compatibilidad con C a nivel de fuente, los ficheros de cabecera, las plantillas ilegibles.

Y trajo cosas que en 2001 eran adelantadas:

- **Módulos de verdad**, en lugar de `#include` (clase 149).
- **Pruebas unitarias en el lenguaje**: bloques `unittest` dentro del propio fichero (clase 139).
- **Contratos**: `in`, `out` e `invariant` — la idea de Ada y Eiffel (clase 118).
- **CTFE (2007)**: ejecutar funciones normales en tiempo de compilación.
- **Arreglos con longitud y rebanadas**, sin la degradación a puntero de C.

Y entonces vino el error que lo marcó: **D2 (2007) rompió la compatibilidad con D1** en medio de la adopción, y **la biblioteca estándar se partió en dos** —Phobos y Tango— con la comunidad dividida. Cuando se resolvió, años después, el hueco lo estaban ocupando otros.

> ***Es la lección más útil de esta ficha**, y la clase 175 la desarrolla: **la compatibilidad y la cohesión del ecosistema pesan más que las características**. Go triunfó con menos características y una promesa de compatibilidad que ha cumplido.*

🏭 Dónde vive hoy

- **Sistemas financieros de baja latencia**: es su nicho más sólido, por el control sobre el recolector.
- **Herramientas internas** en empresas que lo adoptaron pronto — Sociomantic (después Dunnhumby) fue el caso más conocido.
- **El propio compilador**: DMD está escrito en D desde 2017.
- **Y como lenguaje de sustitución de C++ en proyectos concretos**, con interoperabilidad directa (clase 156).

🧠 Lo que enseña: metaprogramación legible y contratos

CTFE — ejecutar el lenguaje al compilar:

```
int factorial(int n) pure { return n <= 1 ? 1 : n * factorial(n - 1); }

enum resultado = factorial(10);   // ← calculado EN COMPILACIÓN, con la MISMA función
```

Y esa es la aportación: no hay un lenguaje de macros aparte. Una función normal, si es `pure` y sus argumentos se conocen, **se ejecuta al compilar**. Es lo que C++ tardó hasta `constexpr` (2011) en tener, y con mucha menos ceremonia.

Los contratos, que son la idea de Ada y de Eiffel (clase 118):

```
int dividir(int a, int b)
in  { assert(b != 0); }
out (r) { assert(r * b <= a); }
do { return a / b; }
```

Las pruebas en el propio fichero, que resuelven un problema real de la clase 139:

```
unittest {
    assert(dividir(10, 2) == 5);
}
```

Se compilan solo con `-unittest`, viven junto al código que prueban y **no hay marco que instalar**. Es una decisión que muy pocos lenguajes han tomado —Rust y Go lo hacen parecido— y que sube muchísimo la probabilidad de que las pruebas existan.

Y el recolector opcional, que es la respuesta de D a la tensión de la clase 131:

```
@nogc void funcionCritica() { /* el compilador PROHÍBE reservar aquí */ }
```

Se puede tener recolector donde no importa y prohibirlo donde sí, comprobado por el compilador.

🔄 Lo que se ha modernizado

- `@safe`, `@trusted`, `@system`: subconjuntos de seguridad de memoria comprobados por el compilador. `@safe` prohíbe la aritmética de punteros y las conversiones inseguras — la idea de la clase 153 sin cambiar de lenguaje.

- **Comprobación de tiempos de vida** (`@live`), en la línea de Rust.
- `betterC`: usar D sin tiempo de ejecución ni recolector, para sistemas embebidos y para integrarse en proyectos C.
- **Interoperabilidad con C++** que va mucho más allá de `extern "C"` — llamar a clases y plantillas (clase 156).
- **DUB** como gestor de paquetes y sistema de construcción (clase 143).

⚙️ Cómo se ejecuta hoy

```
dmd -run main.d < entrada.txt        # DMD: compilación rapidísima, para desarrollar
ldc2 -O3 main.d                       # LDC (LLVM): el que hay que usar en producción
dmd -unittest -main -run main.d       # ← ejecutar las pruebas del propio fichero

dub build && dub test                 # con proyecto y dependencias
```

🧪 El programa de la clase 041 en D

```
import std.stdio, std.array, std.conv, std.algorithm;

void main() {
    auto v = readln().split().map!(to!double).array;
    const total = v[0] * v[1] * (1 - v[2]);
    writefln("Total: %.2f", total);
}
```

Lo que hay que ver.

- `map!(to!double)` usa `!` para los **argumentos de plantilla**: `to!double` es la función de conversión instanciada para `double`. Es la sintaxis de genéricos de D, mucho más ligera que `<...>` de C++, y **se resuelve en compilación** sin coste.
- **La cadena** `readln().split().map!(...).array` es el estilo de rangos de D — los *ranges* son la respuesta de D a los iteradores de C++, y son **perezosos**: nada se calcula hasta el `.array` final (clase 115).
- `const total` es inmutabilidad real, comprobada por el compilador y **transitiva**: en D, `immutable` significa que **nada alcanzable desde ahí puede cambiar** (clase 102), que es más fuerte que el `const` de C++.
- `auto` infiere el tipo, y el programa **no pierde ni una comprobación** por ello: sigue siendo estático y fuerte.
- **Comparado con C++**, hace lo mismo en cuatro líneas y con la mitad de ceremonia. Y esa comparación, junto con la historia de esta ficha, es exactamente la lección de la clase 164: **ser mejor no es suficiente**.

📚 Fuentes y bibliografía

- dlang.org (https://dlang.org/) — especificación, biblioteca Phobos y el tour interactivo.
- D Language Tour (https://tour.dlang.org/) — la mejor introducción, ejecutable en el navegador.
- **Andrei Alexandrescu**, *The D Programming Language*, Addison-Wesley — el libro de referencia, escrito por el codiseñador; también autor de *Modern C++ Design*.
- **Ali Çehreli**, *Programming in D* — libre en línea, muy completo y didáctico.
- Blog de la D Foundation (https://dlang.org/blog/) — artículos técnicos sobre CTFE, rangos y seguridad de memoria.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: C++ · Nim · Zig · Rust · Ada

🎯 Dart — 2011

Dart nació para sustituir a JavaScript en el navegador, fracasó en ese objetivo, y **encontró después su razón de ser en Flutter** — donde hoy mueve aplicaciones móviles, de escritorio y web con un solo código. Es un buen recordatorio de que un lenguaje puede acertar por un camino que no era el previsto.

> 🎯 ***Por qué está en este programa***
>
> *Dart es un **primo de la familia JavaScript / web** (Atlas), cuyos representantes en el núcleo son JavaScript y TypeScript.*
>
> *Aporta al programa algo muy concreto y poco común: **el mismo lenguaje se compila de dos maneras distintas según el momento** — JIT con recarga en caliente mientras se desarrolla, y **AOT a código nativo** al publicar (clases 126 y 174). Es la demostración más clara de que **JIT y AOT no son propiedades del lenguaje, sino decisiones del despliegue**.*

Año	2011; **2.0** con tipado sano (2018); **3.0** (2023) con seguridad frente a nulos completa
Autoría	**Lars Bak** y **Kasper Lund**, Google — los autores de la máquina virtual V8
Familia	JavaScript / web; sintaxis de C, con influencia de Java y Smalltalk
Paradigma	Orientado a objetos con clases; con cierres y programación asíncrona
Tipado	**Estático, sano y con nulabilidad comprobada**; inferencia amplia
Memoria	Recolección de basura generacional, optimizada para objetos de vida corta
Ejecución	**JIT** en desarrollo, **AOT nativo** al publicar, y **a JavaScript o WebAssembly**
Estado	🟢 **Muy vivo** gracias a Flutter

📜 Historia

En **2011**, Google presentó Dart con una ambición explícita: **sustituir a JavaScript**. El plan incluía una máquina virtual de Dart dentro de Chrome. Los demás navegadores no lo aceptaron —con razón: nadie quería un segundo lenguaje web controlado por una empresa— y en **2015 Google retiró el plan**.

Dart quedó como un lenguaje que compilaba a JavaScript y que casi nadie usaba. Y entonces, en **2017**, apareció **Flutter**, y con él la razón de existir: un kit de interfaz que **dibuja cada píxel él mismo** —sin usar los controles nativos— y que necesitaba un lenguaje con **recarga en caliente** para desarrollar y **compilación nativa** para publicar.

Dart 2.0 (2018) hizo el tipado **sano** —hasta entonces era opcional y no garantizaba nada— y **Dart 3.0 (2023)** completó la **seguridad frente a nulos**, además de añadir registros, emparejamiento de patrones y clases selladas.

🏭 Dónde vive hoy

- **Flutter**: aplicaciones móviles (Android e iOS), de escritorio y web con un solo código. Es, con diferencia, el uso principal.
- **Google**: partes de Google Ads, Google Pay y varias herramientas internas.
- **Servidores**: con `dart:io` y marcos como Serverpod, aunque es un nicho pequeño.
- **Herramientas de línea de comandos**: por la compilación AOT y el binario autocontenido (clase 167).

🧠 Lo que enseña: dos compiladores para un lenguaje

Esta es la idea que hay que llevarse de la ficha:

```
Durante el DESARROLLO:
  JIT + recarga en caliente → se cambia una línea y la aplicación se actualiza
  en menos de un segundo, CONSERVANDO su estado.

Al PUBLICAR:
  AOT a código nativo ARM o x86 → arranque instantáneo, sin JIT y sin sobresaltos
  de rendimiento en la primera ejecución de cada función.
```

La recarga en caliente con conservación de estado es lo que hace productivo a Flutter, y es exactamente lo que la clase 124 describe en Smalltalk y Lisp —**el ciclo corto como capacidad, no como comodidad**— llegando al desarrollo móvil.

Y la compilación AOT resuelve el problema contrario: en un móvil no se puede pagar el calentamiento del JIT ni las pausas del recolector durante una animación (clase 152).

Y hay una segunda cosa que Dart hace bien y merece señalarse: **la seguridad frente a nulos aplicada a un lenguaje ya existente**.

```
String nombre = null;      // ✗ no compila
String? apodo = null;       // ✓ el ? declara que puede faltar
print(apodo!.length);        // ! afirma que no es nulo, y falla si lo es
```

La migración se hizo gradual, mezclando código migrado y sin migrar durante la transición — el mismo problema que TypeScript resolvió con el tipado gradual (clase 143).

🔄 Lo que se ha modernizado

- **Registros y emparejamiento de patrones** (3.0): tipos de datos algebraicos y desestructuración (clase 100).
- **Clases selladas** y `switch` exhaustivo comprobado por el compilador.
- **Compilación a WebAssembly** (`dart2wasm`) usando **WasmGC**, además del clásico `dart2js` (clase 162).
- `dart format` **sin opciones**, como `gofmt` (clase 146), y `dart analyze` integrado.
- **Interoperabilidad** con Java/Kotlin, Swift/Objective-C y C mediante `dart:ffi` (clase 156).

⚙️ Cómo se ejecuta hoy

```
dart run main.dart < entrada.txt       # el comando de la clase 041
dart compile exe main.dart -o venta     # ← binario nativo AOT (clase 174)
dart compile js main.dart                # a JavaScript

dart format . && dart analyze            # calidad (clase 146)
dart test                                 # pruebas (clase 139)
flutter run                               # con recarga en caliente
```

🧪 El programa de la clase 041 en Dart

```
import 'dart:io';

void main() {
  final v = stdin.readLineSync()!.split(' ').map(double.parse).toList();
  final total = v[0] * v[1] * (1 - v[2]);
  print('Total: ${total.toStringAsFixed(2)}');
}
```

Lo que hay que ver.

- **El `!` después de `readLineSync()`** es la seguridad frente a nulos en acción: el método devuelve `String?` — puede no haber línea— y `!` **afirma** que la hay. **En JavaScript esa posibilidad no aparece en ningún sitio**, y ahí está la diferencia.
- **`toStringAsFixed(2)` es el `toFixed` de JavaScript** con otro nombre: la herencia de familia se ve en la biblioteca, no solo en la sintaxis.
- **`final` fija el nombre, no el contenido** (clase 102), igual que `const` de JavaScript. Para inmutabilidad real en tiempo de compilación, Dart tiene `const`.
- **`double.parse` como función de primera clase** pasada a `map` — el estilo funcional que comparte con toda la familia.

📚 Fuentes y bibliografía

- dart.dev (https://dart.dev/guides) — el tour del lenguaje y la guía de estilo, muy bien escritos.
- Effective Dart (https://dart.dev/effective-dart) — la guía oficial de estilo, diseño y documentación (clases 146 y 154).
- docs.flutter.dev (https://docs.flutter.dev/) — para el contexto en el que se usa de verdad.
- **Randal Schwartz, Kathy Walrath et al.**, *Dart: Up and Running*, O'Reilly — histórico pero útil para el porqué del diseño.
- Blog de Dart en Medium (https://medium.com/dartlang) — las notas de cada versión, con la motivación de cada característica.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: JavaScript · TypeScript · Swift · Kotlin · Java

🗒️ Datalog — 1977

Datalog es Prolog **al que se le han quitado cosas a propósito** — sin funciones, sin negación sin restricciones, sin orden de cláusulas significativo — y **de esa renuncia salen tres garantías** que Prolog no puede dar: **termina siempre, el resultado no depende del orden, y se puede optimizar como una consulta**.

> 🎯 ***Por qué está en este programa***
>
> *Datalog es un **primo de la familia lógica y declarativa** (Atlas), cuyo representante en el núcleo es SQL.*
>
> *Aporta al programa la demostración más limpia de la tesis que atraviesa el curso (clases 118, 146 y 164): **lo que un lenguaje prohíbe es lo que sus herramientas pueden prometer**. Y aporta el concepto de **recursión declarativa** — consultar una jerarquía o un grafo sin escribir el recorrido (clase 099).*

Año	Formalizado hacia 1977; base teórica en bases de datos deductivas de los ochenta
Autoría	Comunidad académica de bases de datos; el nombre, de Hervé Gallaire y Jack Minker
Familia	Lógica y declarativa; **subconjunto decidible de Prolog**
Paradigma	**Lógico declarativo**, sobre relaciones
Tipado	Según la implementación; los datos son tuplas de constantes
Memoria	Gestionada por el motor
Ejecución	Evaluación **ascendente** con punto fijo (semi-naïve), o mágica
Estado	🟢 **En auge**: análisis de programas, seguridad, bases de datos y grafos

---

📜 Historia

Datalog surgió a finales de los setenta en la intersección de dos comunidades: **la lógica** —de donde venía Prolog— y **las bases de datos** —donde el modelo relacional de Codd (ficha de SQL) acababa de ganar—.

La pregunta era: **¿qué subconjunto de la lógica se puede evaluar como una consulta a una base de datos, con garantías?** Y la respuesta fue quitar tres cosas de Prolog:

1. **Los términos compuestos y las funciones**: solo hay constantes y variables, así que **el universo de valores es finito**.
2. **El corte y el control explícito**: no hay forma de guiar la búsqueda.
3. **El orden de las cláusulas como semántica**: da igual cómo se escriban.

Y de ahí salen las garantías: **todo programa Datalog termina**, y **el resultado es único** —el menor punto fijo—, independientemente de la estrategia de evaluación. Eso es lo que Prolog no puede prometer.

Durante los ochenta y noventa fue sobre todo académico. **Y desde 2010 ha vuelto con fuerza**, por una razón muy concreta: **es el lenguaje ideal para el análisis de programas** — donde las preguntas son relaciones recursivas sobre grafos enormes.

🏭 Dónde vive hoy

- **Análisis estático de programas**: **Soufflé** (Oracle Labs) y **Doop** analizan millones de líneas de Java o C++ expresando el análisis de punteros y de flujo **como reglas Datalog** (clase 150).
- **Seguridad**: **CodeQL** de GitHub —el motor de análisis de vulnerabilidades— usa un lenguaje de consulta con raíces en Datalog, y **Semmle**, su antecesor, era Datalog puro.
- **Bases de datos**: **Datomic** —del autor de Clojure— usa Datalog como lenguaje de consulta; también **XTDB** y **LogicBlox**.
- **Redes y configuración en la nube**: verificación de reglas de acceso y de topologías.
- **Y en investigación**, como base de sistemas de razonamiento incremental.

🧠 Lo que enseña: recursión sin recorrido

Este es el ejemplo canónico, y explica el paradigma entero:

```
% HECHOS: lo que se sabe
padre(ana, luis).
padre(luis, marta).
padre(marta, jorge).

% REGLAS: lo que se deriva
ancestro(X, Y) :- padre(X, Y).
ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y).    % ← recursiva
```

Y con eso, `ancestro(ana, jorge)` es cierto — sin escribir un bucle, sin una pila y sin decidir si se recorre en anchura o en profundidad.

El motor calcula el punto fijo: aplica las reglas una y otra vez hasta que no se derivan hechos nuevos. Y como no hay funciones, **el conjunto de hechos posibles es finito**, así que **termina siempre**.

Y la comparación con los otros dos lenguajes declarativos del Atlas es lo más instructivo de esta ficha:

	Prolog	**Datalog**	**SQL**
Recursión	sí	**sí, y termina**	sí, con `WITH RECURSIVE`
¿Termina siempre?	**no**	**sí**	sí
¿Importa el orden?	**sí**	**no**	no
Evaluación	descendente, con vuelta atrás	**ascendente, punto fijo**	plan del optimizador
Funciones y términos	sí	**no**	limitado
Optimizable como consulta	difícil	**sí**	sí

Y la fila de la terminación es la que justifica el lenguaje: **en análisis de programas se ejecutan reglas sobre grafos de millones de nodos**, y no poder garantizar que el análisis termina lo haría inservible.

Y hay una segunda propiedad, muy práctica, que explica su vuelta: **el mantenimiento incremental**.

```
Si cambia un hecho, NO hay que recalcularlo todo:
el motor propaga solo lo que se ve afectado.
```

Eso es lo que permite que un análisis de código se actualice al guardar un fichero en lugar de tardar minutos, y es lo que hace viable a CodeQL y a los motores de reglas modernos.

🔄 Lo que se ha modernizado

- **Soufflé**: compila Datalog **a C++ paralelo**, con estructuras de datos especializadas — Datalog con rendimiento industrial.
- **Motores incrementales**: **Differential Dataflow** y **DDlog** recalculan solo lo afectado.
- **Extensiones prácticas**: agregados, negación estratificada y tipos, que amplían el lenguaje sin perder las garantías.
- **Datalog embebido**: en Racket (`#lang datalog`), en Clojure (Datomic, Datascript) y como biblioteca en varios lenguajes (clase 163).
- **Y su influencia en SQL**: `WITH RECURSIVE` (SQL:1999) es, esencialmente, **Datalog dentro de SQL**.

⚙️ Cómo se ejecuta hoy

```
souffle -F hechos/ -D salida/ analisis.dl      # Soufflé: compila y ejecuta
racket -e '(require datalog)'                   # dentro de Racket
# Y en Datomic o XTDB, como lenguaje de consulta de la base de datos
```

🍼 El programa de la clase 041 en Datalog

Es la versión que aparece en el `primos.md` de la clase 041, y es un **contrato adaptado** (clase 040): **Datalog puro no tiene entrada, salida ni aritmética general**.

```
% Datalog puro no tiene E/S: se declaran los hechos y la regla que deriva el total.
venta(15000, 2, 0.10).

total(T) :- venta(P, C, D), T = P * C * (1 - D).
```

Lo que hay que ver, y es lo más interesante de la ficha.

- **No hay programa, hay una base de conocimiento.** El hecho `venta(...)` es un dato; la regla `total(T)` **declara qué significa el total**, y el motor lo deriva. **Nadie llama a nada.**
- `T = P * C * (1 - D)` **es una extensión**, no Datalog puro: **el Datalog original no tiene aritmética**, precisamente porque las funciones romperían la garantía de finitud. Las implementaciones prácticas la añaden con cuidado, y **declararlo es más honesto que fingir** (clase 040).
- **La coma es conjunción**, como en Prolog.
- **Y no hay orden**: da igual escribir la regla antes o después del hecho, y da igual el orden de los literales dentro de la regla. **Esa independencia es exactamente la garantía que Prolog no tiene** — y es la razón de que Datalog se pueda optimizar como una consulta SQL.

📚 Fuentes y bibliografía

- Soufflé (https://souffle-lang.github.io/) — el motor moderno; su documentación es la mejor introducción práctica.
- **Serge Abiteboul, Richard Hull, Victor Vianu**, *Foundations of Databases* — libre en línea; los capítulos de Datalog son la referencia teórica.
- What You Always Wanted to Know About Datalog (And Never Dared to Ask) (https://ieeexplore.ieee.org/document/43410) — Ceri, Gottlob y Tanca (1989); el artículo divulgativo clásico.
- **Yannis Smaragdakis, Martin Bravenboer**, *Using Datalog for Fast and Easy Program Analysis* — por qué el análisis de programas encontró aquí su lenguaje.
- Documentación de Datomic (https://docs.datomic.com/query/query-data-reference.html) — Datalog como lenguaje de consulta de una base de datos real.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Prolog · SQL · Clojure · Racket

⏳ Delphi / Object Pascal — 1995

Está más vivo de lo que parece. Miles de aplicaciones de escritorio que usan empresas todos los días — terminales de punto de venta, gestión de almacén, laboratorios, software industrial, ERP regionales— son ejecutables de Delphi. No se ven porque nadie las publica en GitHub: se venden a clientes que llevan veinte años usándolas.

> 🎯 ***Por qué está en este programa***

> ***Criterio de inclusión: Delphi es un producto comercial en desarrollo activo.*** *Embarcadero publica versiones nuevas con regularidad —**Delphi 13 Community Edition** salió en agosto de 2026— y su alternativa libre, **Free Pascal + Lazarus**, tiene desarrollo continuo. No es un lenguaje que sobreviva por inercia: es uno que se factura.*
>
> *Entra porque **es el mejor ejemplo vivo de un modelo que el núcleo no cubre**: la **construcción visual de aplicaciones (RAD)** sobre un **modelo de componentes con propiedades, eventos y publicación de metadatos**. La VCL de Delphi es el antepasado directo de Windows Forms y de la arquitectura de propiedades de C# — no por parecido, sino por autoría: **Anders Hejlsberg diseñó los dos**. Cuando en C# escribes `public string Nombre { get; set; }` o suscribes un evento con `+=`, estás usando ideas que se probaron primero aquí. Y muestra un segundo concepto que importa: la **compilación a un único ejecutable nativo sin runtime**, que es exactamente el debate que Go reabrió veinte años después.*

Año	**Turbo Pascal** 1983; **Delphi 1** en febrero de 1995
Autoría	**Anders Hejlsberg** y el equipo de **Borland** — después CodeGear, Embarcadero, Idera
Familia	Pascal — Object Pascal es su dialecto orientado a objetos
Paradigma	Imperativo y **orientado a objetos**, con componentes y eventos
Tipado	**Estático y fuerte**, con genéricos e inferencia parcial
Memoria	Manual (`Create` / `Free`) con conteo de referencias para interfaces y cadenas; ARC en móvil
Ejecución	**Compilado a nativo**: un único `.exe` sin máquina virtual
Estado	🟢 **Más vivo de lo que parece** — escritorio empresarial, TPV, industria, ERP

📜 Historia

En 1983, un joven danés llamado **Anders Hejlsberg** vendió a Borland un compilador de Pascal que había escrito para CP/M. Se publicó como **Turbo Pascal** por 49,95 dólares —frente a los cientos que costaban los compiladores serios de la época— y cambió el mercado. Cabía en un disquete, integraba editor y compilador en un solo programa y compilaba en segundos cuando lo normal era esperar minutos.

En **febrero de 1995** Borland lanzó **Delphi 1**, para Windows 3.1. Su aportación fue el **desarrollo rápido de aplicaciones (RAD)** llevado a un lenguaje compilado a nativo: arrastras controles a un formulario, ajustas sus propiedades en un inspector, haces doble clic para escribir el manejador del evento, y compilas un **`.exe` único que no necesita instalar nada**. Visual Basic hacía lo primero pero era interpretado y dependía de runtime; C++ compilaba a nativo pero exigía un esfuerzo enorme para la interfaz. Delphi hizo las dos cosas.

La pieza clave era la **VCL** (*Visual Component Library*), y para que funcionara Hejlsberg tuvo que añadir al lenguaje algo que no existía: **propiedades** (campos con `read` / `write` que parecen datos pero ejecutan métodos), **eventos** como punteros a método asignables, y **RTTI publicada** (metadatos que el diseñador visual lee en tiempo de ejecución para saber qué puede editar de cada componente). Ese trío es el modelo de componentes que después reaparecería en .NET.

En 1996 Microsoft fichó a Hejlsberg. Allí diseñó **J++**, luego **C#** y el CLR, y más tarde **TypeScript**. La genealogía es directa y admitida: quien conoce Delphi reconoce medio C# a primera vista.

Borland pasó por varios dueños —CodeGear, **Embarcadero**, hoy dentro de **Idera**— y el producto siguió. **FireMonkey (FMX)**, introducido en 2011, añadió compilación a macOS, iOS, Android y Linux desde el mismo código. En paralelo, el proyecto libre **Free Pascal** alcanzó una compatibilidad muy alta con el dialecto de Delphi, y su IDE **Lazarus** replicó el entorno visual: hoy es una alternativa gratuita y seria.

🏭 Dónde sobrevive hoy

- **Software empresarial de escritorio para Windows**: ERP regionales, gestión comercial, contabilidad, nóminas — especialmente en España, Latinoamérica, Italia, Alemania y Europa del Este.
- **Punto de venta (TPV) y retail**: terminales de caja, gestión de tiendas.
- **Software industrial y de laboratorio**: control de máquinas, adquisición de datos, instrumentación.
- **Aplicaciones verticales**: clínicas, gestorías, talleres, logística — el software que una empresa compró en 2003, que funciona, y que su proveedor sigue manteniendo.
- **Aplicaciones de escritorio conocidas** construidas históricamente con Delphi: **Total Commander**, **FL Studio** o el cliente de escritorio original de **Skype** son casos citados habitualmente.

🧠 Por qué no ha muerto

1. Un `.exe` y nada más. Sin máquina virtual, sin intérprete, sin `node_modules`, sin instalar un runtime en el equipo del cliente. Se copia el fichero y funciona. En entornos corporativos con equipos bloqueados, eso sigue siendo una ventaja real y difícil de igualar.

2. Productividad en aplicaciones con muchos formularios. Para una aplicación de gestión con ochenta pantallas conectadas a una base de datos, el flujo de Delphi —componentes de datos, controles enlazados, diseñador visual— sigue siendo extraordinariamente rápido. Es un nicho que las herramientas web no han mejorado, solo desplazado.

3. Compatibilidad hacia atrás muy larga. Código de Delphi 7 (2002) se recompila hoy con ajustes moderados. Para una empresa con medio millón de líneas, eso es la diferencia entre mantener y reescribir.

4. El coste de reescribir es el argumento de siempre, con el agravante habitual: el conocimiento del negocio vive en ese código y su autor a menudo ya no está.

5. El lenguaje ha seguido creciendo. Genéricos, métodos anónimos (*closures*), atributos, ayudantes de tipo, `for..in`, inferencia con `var`, ARC en plataformas móviles. El Object Pascal de 2026 no se parece al de 1995.

> ***Y una advertencia honesta:*** *la licencia comercial es cara y la comunidad es pequeña comparada con la de cualquier lenguaje del núcleo. Encontrar programadores Delphi es un problema real para las empresas que dependen de él, y es una de las razones por las que muchas migran. Si te acercas por curiosidad, empieza por* ***Lazarus****, que es libre.*

🔄 Lo que se ha modernizado

Delphi es, de esta lista, el que más se ha transformado sin cambiar de nombre:

- **Multiplataforma real con FireMonkey (FMX)**: del mismo código fuente salen ejecutables para **Windows, macOS, iOS, Android y Linux**, incluidos ARM64 y Apple Silicon.
- **El lenguaje creció**: genéricos, **métodos anónimos** (closures), atributos personalizados y RTTI extendida, ayudantes de tipo, `for..in`, inferencia con `var x := ...`, y operadores nulos.

- **Servicios y nube**: **RAD Server** para publicar APIs REST, clientes HTTP/REST integrados, componentes para servicios en la nube y **FireDAC** para acceso a prácticamente cualquier base de datos.
- **IDE moderno**: motor **LSP** para completado y navegación de código, refactorizaciones, y soporte de Git integrado.
- **Pruebas unitarias y CI**: **DUnitX** y **Delphi-Mocks** dan pruebas automatizadas, y la compilación por línea de comandos (`dcc32` / `dcc64`, MSBuild) permite integrarlo en pipelines.
- **Community Edition gratuita**, que bajó la barrera de entrada para quien quiera probarlo sin licencia comercial.
- Y en paralelo, **Free Pascal + Lazarus** ofrecen la misma capacidad multiplataforma con licencia libre, incluida compilación a **WebAssembly**.

⚙️ Cómo se ejecuta hoy

```
# Free Pascal (libre) — compatible con el dialecto de Delphi
sudo apt-get install -y fpc
fpc -Mdelphi total_venta.pas
echo "15000 2 0.10" | ./total_venta
# Total: 27000.00
```

```
REM Delphi (Embarcadero), desde la línea de comandos
dcc64 TotalVenta.dpr
TotalVenta.exe
```

Herramientas: Delphi / RAD Studio de Embarcadero, con **Community Edition** gratuita para uso individual y pequeñas empresas; **Free Pascal** (`fpc`) más **Lazarus** como alternativa libre y multiplataforma. Gestión de paquetes: **GetIt** en Delphi, **Online Package Manager** en Lazarus.

Ficheros del ecosistema: `.pas` (unidad de código), `.dpr` (proyecto), `.dfm` (formulario, un fichero de texto con las propiedades de los componentes, versionable en Git), `.dproj` (configuración del proyecto).

🧪 El programa de la clase 041 en Object Pascal

```
program TotalVenta;
{$MODE DELPHI}{$H+}

uses
  SysUtils;

type
  TVenta = class
  private
    FPrecio, FCantidad, FDescuento: Double;
    function GetTotal: Double;
  public
    constructor Create(APrecio, ACantidad, ADescuento: Double);
    property Precio    : Double read FPrecio;
    property Cantidad  : Double read FCantidad;
    property Descuento : Double read FDescuento;
    property Total     : Double read GetTotal;      // se calcula al leerla
  end;

constructor TVenta.Create(APrecio, ACantidad, ADescuento: Double);
begin
  inherited Create;
  FPrecio    := APrecio;
  FCantidad  := ACantidad;
  FDescuento := ADescuento;
end;

function TVenta.GetTotal: Double;
begin
  Result := FPrecio * FCantidad * (1 - FDescuento);
end;

var
  Precio, Cantidad, Descuento: Double;
  Venta: TVenta;

begin
  Read(Precio, Cantidad, Descuento);

  Venta := TVenta.Create(Precio, Cantidad, Descuento);
  try
    WriteLn('Total: ', Venta.Total:0:2);
  finally
    Venta.Free;
  end;
end.
```

Recorrido, línea a línea.

- `{$MODE DELPHI}` pone a Free Pascal en dialecto Delphi. Con Delphi de Embarcadero esta directiva no hace falta.
- `uses SysUtils;` es la importación de unidades. El sistema de módulos de Object Pascal —una unidad con secciones `interface` e `implementation`— viene de **Modula-2**, y es una de las razones de la velocidad de compilación: el compilador solo necesita la `interface` de las dependencias.
- `type TVenta = class` declara una clase. La convención `T` inicial para tipos es universal en este ecosistema, igual que `F` para campos privados (*field*) y `A` para argumentos.

- `property Total: Double read GetTotal;` **es la línea que hay que entender.** Desde fuera, `Venta.Total` se lee **exactamente como un campo**: `WriteLn(Venta.Total)`. Pero por dentro se ejecuta el método `GetTotal`. Eso es una **propiedad**, y en 1995 era una novedad: permitía cambiar un campo público por un cálculo sin romper a ningún cliente, y permitía al **inspector de objetos** del IDE mostrar y editar los atributos de un componente sin conocerlo. Es, literalmente, el mismo concepto que `public double Total => ...` en C#, con veinte años de diferencia y el mismo autor.
- `constructor Create` — el constructor **tiene nombre** y se llama por convención `Create`; no hay sintaxis especial. Y se invoca **sobre la clase**: `TVenta.Create(...)`, no `new TVenta(...)`.
- `try ... finally Venta.Free; end;` **no es opcional.** En Windows, Object Pascal **no tiene recolector de basura**: todo objeto creado con `Create` debe liberarse con `Free`. El bloque `try..finally` garantiza que ocurra aunque haya una excepción, y es el idioma más repetido de todo el código Delphi del mundo. (En las plataformas móviles hubo un periodo con ARC; hoy el modelo se ha reunificado hacia la gestión manual.)
- `Result := ...` es cómo una función devuelve un valor en Object Pascal. `Result` es una variable implícita, no una palabra de retorno: se le puede asignar varias veces y el código continúa.
- `Venta.Total:0:2` reutiliza el formateo integrado de Pascal: ancho 0, dos decimales.

Y el modelo que de verdad define a Delphi, aunque no quepa en un ejemplo de consola: en una aplicación real el mismo cálculo estaría enganchado a un evento.

```
procedure TFormVenta.EditCantidadChange(Sender: TObject);
begin
  LabelTotal.Caption := Format('Total: %.2f', [CalcularTotal]);
end;
```

El IDE generó esa firma al hacer doble clic sobre el control, la asoció al evento `OnChange` en el fichero `.dfm`, y `Sender` identifica quién lo disparó. **Un evento es un puntero a método asignable en tiempo de ejecución** — el mismo concepto que un `delegate` de C#, y por la misma razón: el mismo diseñador.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En Object Pascal es…
`new Clase(args)`	`TClase.Create(args)` — el constructor se llama sobre la clase
`delete` / recolector	`.Free` dentro de `try..finally`
`public int X { get; set; }` (C#)	`property X: Integer read FX write FX;` — el original
`event` / `delegate` (C#)	`property OnAlgo: TNotifyEvent read FOnAlgo write FOnAlgo;`
`interface`	`interface`, con conteo de referencias vía `IInterface`
`List<T>`	`TList<T>` de `System.Generics.Collections`
Lambda	`procedure of object` (método) o método anónimo `procedure begin ... end`
`foreach`	`for X in Coleccion do`
`string.Format(...)`	`Format('%.2f', [x])` — con los argumentos en un array abierto
`try/catch/finally`	`try..except` y `try..finally` — **bloques separados**, no uno solo
`var x = ...` (inferencia)	`var x := ...;` (Delphi 10.3 en adelante)

⚠️ Errores comunes al leerlo

- **Olvidar `Free`.** Fuga de memoria silenciosa. Y liberar dos veces es peor: corrupción.

- **Confundir `try..except` con `try..finally`.** Son dos construcciones distintas y **no** se combinan en un solo bloque como en Java o C#; hay que anidarlas.
- **Creer que una propiedad es un campo.** `Venta.Total` puede estar ejecutando código, abriendo una conexión o disparando un evento. En depuración, esa distinción importa.
- **Suponer que `string` es igual en todas partes.** Sin `{$H+}` es de 255 caracteres. En Delphi moderno es UTF-16 con conteo de referencias; en Free Pascal el detalle depende del modo.
- **Ignorar el `.dfm`.** El formulario no está en el `.pas`: está en un fichero de texto paralelo con las propiedades de cada componente. Quien lee solo el código no ve la mitad de la aplicación.
- **Asumir que Delphi es solo Windows.** Con FireMonkey compila para macOS, iOS, Android y Linux; con Free Pascal/Lazarus, para más plataformas todavía.

📚 Fuentes y bibliografía

- Embarcadero Delphi (https://www.embarcadero.com/products/delphi) — el producto comercial y su **Community Edition** gratuita.
- DocWiki de Embarcadero (https://docwiki.embarcadero.com/RADStudio/en/Main_Page) — la referencia oficial del lenguaje y de las bibliotecas.
- Lazarus IDE (https://www.lazarus-ide.org/) y Free Pascal (https://www.freepascal.org/) — la alternativa libre, por donde conviene empezar.
- **Marco Cantù**, *Object Pascal Handbook* — el libro de referencia del lenguaje moderno; el autor publica versiones actualizadas y hay ediciones de descarga gratuita.
- **Marco Cantù**, *Mastering Delphi* (serie) — la obra clásica sobre la VCL y el desarrollo de aplicaciones.
- **Nick Hodges**, *Coding in Delphi* y *More Coding in Delphi* — Delphi moderno con genéricos, inyección de dependencias y pruebas unitarias; el puente hacia prácticas actuales.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Pascal · VBA · Ada

💧 Elixir — 2011

Elixir es lo que pasa cuando alguien mira Erlang, reconoce que su máquina virtual es extraordinaria y decide que **el problema es todo lo demás**: la sintaxis, las herramientas y la documentación. El resultado conserva el modelo de actores y la tolerancia a fallos, con la ergonomía de Ruby.

> 🎯 ***Por qué está en este programa***
>
> *Elixir es un **primo de la familia concurrente / actor** (Atlas), junto a Erlang, con quien comparte máquina virtual.*
>
> *Aporta al programa una lección de la clase 164 en estado puro: **el ecosistema y la ergonomía pesan más que el modelo técnico**. Erlang tenía el modelo desde 1986 y seguía siendo minoritario; Elixir **no cambió el modelo** —cambió la sintaxis y las herramientas— y multiplicó la adopción.*

Año	2011; **1.0** en 2014; **1.18** actual, con análisis de tipos
Autoría	**José Valim**, que venía del núcleo de Ruby on Rails
Familia	Concurrente / actor; sobre la **BEAM** de Erlang
Paradigma	**Funcional y concurrente**, con datos inmutables
Tipado	**Dinámico y fuerte**; con **inferencia de tipos gradual** en desarrollo
Memoria	La de la BEAM: montón y recolector **por proceso**
Ejecución	Bytecode sobre la BEAM, con JIT
Estado	🟢 **En crecimiento**: web en tiempo real, sistemas distribuidos, IoT

📜 Historia

José Valim era miembro del equipo central de **Ruby on Rails** y se topó con un problema estructural: **Ruby no aprovecha bien los procesadores multinúcleo** por su bloqueo global del intérprete (clase 135). Buscando alternativas, encontró la BEAM.

Su diagnóstico fue el que define esta ficha: **la máquina virtual de Erlang es una obra maestra de ingeniería, y su lenguaje y sus herramientas son la barrera de entrada**.

Así que en **2011** construyó **Elixir**: **el mismo modelo, la misma máquina virtual, la misma interoperabilidad total con Erlang** —se pueden llamar módulos de Erlang directamente, sin capa— y encima:

- **Sintaxis inspirada en Ruby**, mucho más familiar.
- **Macros higiénicas** al estilo Lisp, con las que se construye buena parte del lenguaje.
- **Herramientas modernas**: `mix` para construir, `hex` para dependencias con fichero de bloqueo (clase 143), `ExUnit` para pruebas (clase 139) y **documentación de primera clase**.
- **Y** `iex`, una consola interactiva excelente.

Phoenix (2014) hizo por Elixir lo que Rails hizo por Ruby, y en **2018** llegó **LiveView**: una forma de construir interfaces web interactivas **sin escribir JavaScript**, manteniendo el estado en el servidor con un proceso por usuario — algo que solo es viable porque **los procesos de la BEAM son baratísimos** (clase 168).

🏭 Dónde vive hoy

- **Aplicaciones web en tiempo real**: Discord —millones de usuarios concurrentes—, Pinterest, Bleacher Report, Heroku.
- **Sistemas distribuidos y mensajería**, donde la tolerancia a fallos importa.
- **Internet de las cosas**: **Nerves** permite ejecutar Elixir en dispositivos embebidos, con actualización remota y supervisión.
- **Procesamiento de datos**: **Broadway** para canalizaciones con contrapresión (clase 168).
- **Y aprendizaje automático**: **Nx** y **Livebook** —cuadernos ejecutables, al estilo de Jupyter— son un desarrollo reciente y notable.

🧠 Lo que enseña: la misma potencia con otra puerta

Uno, la canalización, que es la marca de la casa:

```
"1 2 3"
|> String.split()
|> Enum.map(&String.to_float/1)
|> Enum.sum()
```

`|>` pasa el resultado como primer argumento de la siguiente función. Viene de F# y de OCaml, y hace que las transformaciones se lean **en el orden en que ocurren** en lugar de anidadas.

Dos, el emparejamiento de patrones como control de flujo:

```
def procesar({:ok, valor}), do: "todo bien: #{valor}"
def procesar({:error, razon}), do: "falló: #{razon}"
```

Varias cláusulas de la misma función, seleccionadas por la forma del argumento — como en Erlang y en Haskell. Y la convención `{:ok, ...}` / `{:error, ...}` es el manejo de errores idiomático (clase 116): **el resultado lleva el éxito o el fallo en su forma**.

Y tres, las macros, con las que está construido el propio lenguaje:

```
defmodule MiPrueba do
  use ExUnit.Case            # ← una macro que inyecta el comportamiento de prueba
  test "suma" do             # ← 'test' también es una macro
    assert 1 + 1 == 2         # ← y 'assert' MUESTRA los valores al fallar
  end
end
```

`if`, `unless`, `def` y `defmodule` son macros, no palabras clave del lenguaje. Es la homoiconicidad de Lisp (clase 123) aplicada con una sintaxis convencional — y es lo que permite que Phoenix y Ecto tengan las API que tienen.

Y merece señalar lo que **no** cambió: **el modelo de procesos, la supervisión y "déjalo fallar" son exactamente los de Erlang**. Un módulo de Erlang se llama desde Elixir con `:modulo.funcion()`, sin puente ni conversión (clase 155).

🔄 Lo que se ha modernizado

- **Inferencia de tipos gradual** (desde 1.17): un sistema de **conjuntos de tipos** que detecta errores **sin anotaciones**, integrado en el compilador. Es uno de los desarrollos de diseño de lenguajes más interesantes en curso (clase 146).
- **LiveView**: interfaz web con estado en el servidor y actualizaciones por WebSocket — con un proceso por usuario (clases 168 y 169).
- **Nx, Axon, Bumblebee y Livebook**: aprendizaje automático y cuadernos ejecutables sobre la BEAM.
- **Nerves**: sistemas embebidos con la misma tolerancia a fallos.
- **Y `mix format`** sin opciones, como `gofmt` (clase 146).

⚙️ Cómo se ejecuta hoy

```
elixir main.exs < entrada.txt        # como guion
iex                                   # consola interactiva

mix new proyecto && mix test          # construcción y pruebas (clases 139 y 147)
mix format && mix credo                # estilo y análisis (clase 146)
mix dialyzer                            # análisis de tipos con Dialyzer
```

🧪 El programa de la clase 041 en Elixir

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
[precio, cantidad, descuento] =
  IO.gets("")
  |> String.trim()
  |> String.split()
  |> Enum.map(&String.to_float/1)

total = precio * cantidad * (1 - descuento)
IO.puts("Total: #{:erlang.float_to_binary(total, decimals: 2)}")
```

Lo que hay que ver.

- **La canalización se lee de arriba abajo**, en el orden de las operaciones: leer, recortar, partir, convertir. Compárese con la versión anidada que haría falta sin `|>`.
- `[a, b, c] = ...` **es emparejamiento**, igual que en Erlang: **el** `=` **de Elixir es un operador de coincidencia, no de asignación** (clase 041).
- `&String.to_float/1` es la captura de una función con su aridad — la aridad forma parte de la identidad de la función, herencia directa de Erlang.
- `:erlang.float_to_binary(...)` **llama a Erlang directamente**, con `:` para los átomos de módulo. **Esa es la interoperabilidad total**: no hay envoltorio ni conversión (clase 155).
- **Y** `#{...}` **es interpolación de cadenas**, tomada de Ruby — la ergonomía que Valim vino a traer.

📚 Fuentes y bibliografía

- elixir-lang.org (https://elixir-lang.org/getting-started/introduction.html) — la guía oficial; **la documentación de Elixir es de las mejores de cualquier lenguaje**, y eso fue deliberado.
- Elixir School (https://elixirschool.com/es) — en español, comunitario.
- **Dave Thomas**, *Programming Elixir ≥ 1.6*, Pragmatic — la introducción de referencia.
- **Saša Jurić**, *Elixir in Action*, 3.ª ed., Manning — **el mejor libro para entender OTP y la concurrencia**, no solo la sintaxis.
- **Chris McCord et al.**, *Programming Phoenix LiveView* — para la parte web (clases 168 y 169).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Erlang · Ruby · Clojure · Go · F#

🌳 Elm — 2012

Elm hace una promesa que ningún otro lenguaje de esta lista se atreve a hacer: **si compila, no lanza excepciones en ejecución**. Y la cumple. Es un lenguaje pequeño, deliberadamente limitado, cuya influencia real está mucho más allá de su cuota de uso: **Redux, la arquitectura de estado que domina el front-end, salió de aquí**.

> ### 🎯 *Por qué está en este programa*
>
> *Elm es un **primo de la familia JavaScript / web** (Atlas) y también de la **funcional tipada** — está en el cruce de las dos.*
>
> *Aporta al programa la demostración más contundente de la tesis que atraviesa el curso (clases 118, 146 y 164): **lo que un lenguaje prohíbe es lo que sus herramientas pueden prometer**. Elm prohíbe los efectos secundarios, la nulidad, las excepciones y la interoperabilidad directa — y a cambio garantiza que*

> *no habrá errores en ejecución.*

Año	2012; **0.19** (2018) es la versión estable actual — sin 1.0
Autoría	**Evan Czaplicki**, como tesis de fin de grado en Harvard
Familia	Funcional tipada (ML) aplicada a la web
Paradigma	**Funcional puro**: sin efectos secundarios en el lenguaje
Tipado	**Estático, inferido, sin `null` y sin excepciones**; tipos algebraicos
Memoria	La de JavaScript: recolección automática
Ejecución	**Compila a JavaScript**
Estado	🟡 **Estable y de nicho**: poco movimiento, y lo que hay funciona

---

📜 Historia

Evan Czaplicki presentó Elm en **2012** como tesis de grado. La pregunta de partida era buena: **¿por qué las interfaces web son tan propensas a fallar en ejecución?** Y la respuesta que propuso fue radical: **quitar del lenguaje todo lo que puede fallar**.

- **No hay `null` ni `undefined`**: lo que puede faltar se declara con `Maybe`.
- **No hay excepciones**: lo que puede fallar devuelve `Result`.
- **No hay efectos secundarios en las funciones**: los efectos los ejecuta el tiempo de ejecución.
- **No hay interoperabilidad directa con JavaScript**: se habla con él por **puertos**, mensajes con tipo comprobado.

De esa última restricción sale la garantía: **si el código JavaScript no puede entrar en el mundo de Elm sin pasar por un puerto tipado, no puede romperlo**.

Y su influencia fue mucho mayor que su uso: **la arquitectura de Elm** —modelo, mensaje, actualización, vista— **inspiró directamente a Redux** (Dan Abramov lo ha reconocido), y con Redux llegó a millones de aplicaciones React. Es la idea de la clase 169: **interfaz = f(estado)**.

Su evolución es lenta y deliberada, con versiones espaciadas años, lo que ha generado tanto elogios por la estabilidad como críticas por la falta de movimiento.

🏭 Dónde vive hoy

- **Aplicaciones web con mucha lógica de interfaz** en empresas que lo adoptaron: NoRedInk, Rakuten, Vendr, y varias del sector financiero.
- **Formularios y flujos complejos**, donde el coste de un error en producción justifica la restricción.
- **Enseñanza de programación funcional**, por la calidad de sus mensajes de error.
- **Y como influencia**, que es donde más presencia tiene: Redux, Vuex, Pinia, The Elm Architecture en Rust (`iced`) y en otros ecosistemas.

🧠 Lo que enseña: la garantía sale de la renuncia

El mensaje de error como característica de producto. Elm es famoso por esto:

```
-- TYPE MISMATCH ----------------------------------------- src/Main.elm

The 1st argument to `toFloat` is not what I expect:

23|     toFloat modelo.nombre
                ^^^^^^^^^^^^
This `nombre` value is a: String
But `toFloat` needs the 1st argument to be: Int

Hint: Want to convert a String into an Int? Use String.toInt!
```

Ese nivel de mensaje elevó el listón de toda la industria: Rust, TypeScript y varios compiladores mejoraron sus errores explícitamente citando a Elm. Es un caso claro de la clase 137 — **el diagnóstico es parte del lenguaje, no un detalle de implementación**.

Y la arquitectura de Elm, que es lo más transferible:

```
type Msg = Incrementar | Decrementar

update : Msg -> Model -> Model          -- ← una FUNCIÓN PURA: mensaje + estado → estado nuevo
update msg model =
    case msg of
        Incrementar -> { model | contador = model.contador + 1 }
        Decrementar -> { model | contador = model.contador - 1 }

view : Model -> Html Msg                 -- ← la vista es una FUNCIÓN del estado
```

Y las propiedades que eso da son las de la clase 169: el estado está en un sitio, los cambios son explícitos y con nombre, la vista se deriva del estado, y **todo es probable sin navegador** porque `update` es una función pura (clase 139).

> ***Y el coste hay que decirlo, porque es alto*** *(clase 164):* ***no se puede llamar a una biblioteca de JavaScript directamente****. Todo pasa por puertos, con serialización de por medio. En un ecosistema donde la solución a cualquier problema es un paquete de npm, esa restricción es la barrera de adopción — y es, a la vez, exactamente lo que hace posible la garantía.*

🔄 Estado actual

- **0.19 desde 2018**, con un ritmo de publicación muy lento; la comunidad mantiene el ecosistema.
- `elm-review`: análisis estático con reglas propias, en la línea de la clase 146.
- `elm-test` y `elm-program-test`: pruebas de la lógica sin navegador.
- **Y la discusión abierta y honesta** sobre si la estabilidad extrema es virtud o abandono — que es un debate útil para la clase 164.

⚙️ Cómo se ejecuta hoy

```
elm make src/Main.elm --output=main.js       # compilar a JavaScript
elm repl                                      # explorar el lenguaje
elm reactor                                   # servidor de desarrollo

elm-test && elm-review                        # pruebas y análisis
```

🧪 El programa de la clase 041 en Elm

Elm **no tiene entrada estándar**: se ejecuta en el navegador, y la entrada llega por puertos o por eventos. Como en SQL y en JCL, esto es un **contrato adaptado** (clase 040), y declararlo es más honesto que fingirlo.

```
module Venta exposing (total)

-- La lógica pura, que es lo que Elm quiere que sea el 95 % del programa:
total : Float -> Float -> Float -> String
total precio cantidad descuento =
    let
        resultado =
            precio * cantidad * (1 - descuento)
    in
    "Total: " ++ formatear2 resultado


-- Y el tipo de la entrada declara que PUEDE FALLAR:
parsear : String -> Result String Float
parsear texto =
    case String.toFloat texto of
        Just n -> Ok n
        Nothing -> Err ("No es un número: " ++ texto)
```

Lo que hay que ver.

- `String.toFloat` **devuelve** `Maybe Float`**, no un número.** Es la diferencia esencial con JavaScript, donde `Number("abc")` devuelve `NaN` y sigue adelante: **aquí el fallo está en el tipo y el compilador obliga a tratarlo** (clase 116).
- **La firma** `Float -> Float -> Float -> String` **está escrita a mano** aunque Elm la infiera. Es la convención del ecosistema, y funciona como documentación comprobada (clase 154).
- **No hay** `return`**, ni sentencias, ni mutación**: `let ... in` nombra un valor intermedio, y la función **es** una expresión.
- **Y** `Result String Float` es un tipo algebraico: **o hay un valor, o hay un error con mensaje** — nunca las dos cosas ni ninguna. Es la misma idea que `Result` de Rust y `Either` de Haskell.

📚 Fuentes y bibliografía

- Guía oficial de Elm (https://guide.elm-lang.org/) — corta, completa y muy bien escrita; se lee en una tarde.
- Elm Packages (https://package.elm-lang.org/) — con **versionado semántico impuesto por herramienta**: `elm publish` **calcula** si el cambio es mayor o menor analizando la API. Es la clase 143 automatizada de verdad, y merece conocerse aunque no se use Elm.
- **Richard Feldman**, *Elm in Action*, Manning — el libro de referencia.
- elm-radio (https://elm-radio.com/) — pódcast con discusiones de diseño aplicables a cualquier lenguaje.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Haskell · OCaml · F# · TypeScript · JavaScript

📝 Emacs Lisp — 1985

Emacs Lisp es, probablemente, **el lenguaje incrustado con más código escrito de la historia** — y el único cuyo anfitrión es tan indistinguible de él que la pregunta "¿Emacs es un editor o un intérprete de Lisp?" tiene una respuesta clara: **es un intérprete de Lisp que resulta que edita texto**.

> 🎯 ***Por qué está en este programa***

> *Emacs Lisp es un **primo de la familia Lisp** (Atlas).*
>
> *Aporta al programa el caso de estudio más grande y más antiguo de **incrustar un lenguaje en otro**: cuarenta años de una aplicación cuyo 90 % está escrito en su propio lenguaje de extensión, **modificable en marcha**. Y es el último lugar de uso masivo donde sobrevive el **alcance dinámico** (clase 088), que aquí se puede estudiar en vivo.*

Año	1985 (GNU Emacs 13); Emacs es de 1976, con otro lenguaje
Autoría	**Richard Stallman**, sobre el Emacs original de Stallman y Guy Steele
Familia	Lisp; más cerca de MacLisp que de Scheme o Common Lisp
Paradigma	Imperativo y funcional; orientado a objetos con EIEIO
Tipado	**Dinámico**; con alcance **dinámico por defecto** hasta hace poco
Memoria	Recolección de basura
Ejecución	Bytecode interpretado; **compilación nativa a C** desde Emacs 28
Estado	🟢 **Muy vivo** dentro de su ecosistema; irrelevante fuera

---

📜 Historia

Emacs nació en 1976 en el MIT como un conjunto de macros para el editor TECO —de ahí el nombre, *Editor MACroS*—. Cuando **Richard Stallman** arrancó el proyecto GNU en 1984, reescribió Emacs desde cero y tomó una decisión que lo definió todo: **el editor tendría un intérprete de Lisp dentro, y todo lo que no fuera imprescindible se escribiría en Lisp**.

El resultado es una arquitectura de dos capas exacta a la que Ousterhout describiría trece años después (clase 155):

```
Un núcleo en C:  el intérprete de Lisp, la gestión de búferes, el dibujado
                  y las primitivas de bajo nivel.

Todo lo demás en Emacs Lisp:  los modos de edición, el cliente de correo,
                               la interfaz de git, el gestor de ficheros,
                               el depurador, el calendario, el organizador…
```

Y con una propiedad que ningún otro sistema de esta lista tiene igual: **todo eso es modificable en marcha, sin reiniciar y sin recompilar** (clase 124).

Org mode —un sistema de notas, tareas, agenda y documentos reproducibles— es, por sí solo, uno de los programas Lisp más grandes que existen, y está escrito íntegramente en Emacs Lisp por gente que no es programadora de lenguajes.

Y en **2021, Emacs 28** trajo el cambio más importante en décadas: **compilación nativa**. Emacs Lisp se traduce a C y se compila con GCC, con mejoras de rendimiento de varias veces.

🏭 Dónde vive hoy

- **GNU Emacs**: decenas de miles de paquetes en MELPA, todos en Emacs Lisp.
- **Org mode**: notas, agenda, y **documentos con código ejecutable** — `org-babel` ejecuta bloques de Python, R, SQL o shell dentro del documento y captura la salida (clase 154)—.

- **Magit**: una interfaz de git que muchos consideran mejor que cualquier cliente dedicado, escrita entera en Emacs Lisp.
- **Y como entorno de desarrollo** para Common Lisp —con SLIME—, Haskell, Clojure y media docena más.

🧠 Lo que enseña: el alcance dinámico, en vivo

Emacs Lisp es **el último sitio de uso masivo donde el alcance dinámico es el comportamiento por defecto**, y eso lo hace un laboratorio perfecto para la clase 088:

```
(defvar *nivel* 0)                    ; variable especial: alcance DINÁMICO

(defun escribir (msg)
  (message "%s%s" (make-string *nivel* ?\s) msg))

(defun con-sangria (f)
  (let ((*nivel* (+ *nivel* 2)))      ; ← afecta a TODO lo que se llame desde aquí
    (funcall f)))
```

`let` **sobre una variable especial no crea una variable local: cambia temporalmente la global**, y todo lo que se ejecute dentro —incluidas funciones que no sabían nada de esto— ve el valor nuevo.

Y en un editor eso resulta ser exactamente lo que hace falta: `case-fold-search`, `default-directory`, `inhibit-read-only` son variables que se enlazan alrededor de una operación para cambiar cómo se comporta todo el código que participa.

Es la razón por la que Emacs Lisp lo conservó cuando el resto del mundo Lisp se pasó al alcance léxico — y desde Emacs 24 **se puede activar el léxico por fichero**, que es hoy la recomendación:

```
;;; -*- lexical-binding: t -*-
```

Y hay una segunda cosa que Emacs enseña, y es de la clase 163: **el anfitrión y el lenguaje incrustado casi no tienen frontera**.

```
(defun mi-comando ()
  (interactive)                       ; ← esto lo convierte en un comando invocable
  (insert (format-time-string "%Y-%m-%d")))

(global-set-key (kbd "C-c d") #'mi-comando)
```

No hay API que registrar ni puente que cruzar: las funciones del editor **son** funciones Lisp, y las funciones propias **son** comandos del editor. Es el extremo opuesto de los alias de Safe-Tcl, donde la frontera está cerrada a propósito (clase 153).

> ***Y el precio es el que la clase 163 advierte: no hay aislamiento.*** *Un paquete de MELPA puede hacer cualquier cosa, y la seguridad depende por completo de la confianza en quien lo escribió.*

🔄 Lo que se ha modernizado

- **Compilación nativa** (28): Emacs Lisp a C y a código máquina con GCC.
- **Enlace léxico** por fichero, hoy recomendado siempre; y `cl-lib` con las construcciones de Common Lisp.
- **Hilos** (26) —cooperativos— y `seq` / `map` como bibliotecas genéricas de colecciones.

- **Tree-sitter** (29) para el análisis sintáctico de los modos de lenguaje, sustituyendo décadas de expresiones regulares (clase 123).
- **Y `use-package` y `straight.el`** como gestión declarativa de la configuración y de las dependencias (clase 143).

⚙️ Cómo se ejecuta hoy

```
emacs --batch -l main.el                  # ejecutar un guion sin interfaz
emacs --batch -f batch-byte-compile *.el    # compilar a bytecode
emacs --batch -l ert -f ert-run-tests-batch-and-exit   # pruebas (clase 139)

# Y dentro de Emacs: C-x C-e evalúa la expresión anterior, EN EL SISTEMA VIVO
```

🪠 El programa de la clase 041 en Emacs Lisp

Emacs Lisp puede ejecutarse en lote, aunque no sea su uso natural. **No está verificado en CI** (clase 040).

```
;;; -*- lexical-binding: t -*-
(let* ((linea (read-string ""))
       (campos (split-string linea))
       (nums (mapcar #'string-to-number campos))
       (total (* (nth 0 nums) (nth 1 nums) (- 1 (nth 2 nums)))))
  (princ (format "Total: %.2f\n" total)))
```

Lo que hay que ver.

- **La primera línea, con `lexical-binding: t`, no es un comentario decorativo**: **cambia la semántica del fichero entero** (clase 088). Sin ella, `let*` crearía enlaces dinámicos. Es una de las pocas veces en que un comentario mágico cambia el significado del programa.
- `#'string-to-number` usa la comilla de función: en un *Lisp-2* como este, **funciones y valores viven en espacios de nombres distintos** —a diferencia de Scheme, que es un *Lisp-1*—.
- `princ` **y no** `message`: en modo lote, `message` va al error estándar y `princ` a la salida (clase 167).
- `%.2f` **en** `format` es, otra vez, la herencia de C.
- **Y este programa es lo menos representativo del lenguaje que se puede escribir**: Emacs Lisp está hecho para manipular búferes, no para leer de la entrada estándar. Es un **contrato adaptado** (clase 040), y decirlo es más honesto que fingir.

📚 Fuentes y bibliografía

- An Introduction to Programming in Emacs Lisp (https://www.gnu.org/software/emacs/manual/eintr.html) — **Robert Chassell**, incluido en Emacs; escrito para quien no ha programado nunca.
- Emacs Lisp Reference Manual (https://www.gnu.org/software/emacs/manual/elisp.html) — completo y disponible dentro del propio editor con `C-h i`.
- Mastering Emacs (https://www.masteringemacs.org/) — **Mickey Petersen**; el mejor libro sobre el editor y su modelo de extensión.
- **Org mode** y `org-babel` — para ver la idea de documento reproducible (clase 154) en su forma más madura.
- MELPA (https://melpa.org/) — el archivo de paquetes; útil como ejemplo de ecosistema sin aislamiento (clase 153).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Common Lisp · Scheme · AutoLISP · Tcl · Lua

📞 Erlang — 1986

Erlang se diseñó en Ericsson para centralitas telefónicas, con un requisito que ningún otro lenguaje de esta lista tuvo: **no puede pararse nunca**. De ahí salió un modelo de concurrencia y de tolerancia a fallos tan distinto del resto que **su lema es "déjalo fallar"** — y funciona.

> 🎯 ***Por qué está en este programa***
>
> *Erlang es el representante de la **familia concurrente / actor** (Atlas), que **no tiene representante en el núcleo** — su parienta más cercana es la concurrencia CSP de Go.*
>
> *Aporta al programa **el modelo de actores** (clase 133) y, sobre todo, **una filosofía de tolerancia a fallos distinta de todo lo demás del curso**: no intentar que el código no falle, sino **organizar el sistema para que fallar no importe** (clases 116 y 148).*

Año	1986; libre desde 1998; OTP desde 1996
Autoría	**Joe Armstrong**, Robert Virding y Mike Williams — Ericsson
Familia	Concurrente / actor; con influencia de Prolog —su sintaxis— y de ML
Paradigma	**Funcional, concurrente y distribuido**, sin estado compartido
Tipado	**Dinámico y fuerte**; con Dialyzer para análisis estático opcional
Memoria	**Un montón y un recolector POR PROCESO** — sin pausas globales
Ejecución	Bytecode sobre la **BEAM**, con JIT desde OTP 24
Estado	🟢 **Vivo y estratégico**: telecomunicaciones, mensajería y sistemas distribuidos

---

📜 Historia

En **1986**, el laboratorio de Ericsson buscaba una forma mejor de programar centralitas telefónicas. Los requisitos eran, literalmente, los más exigentes de esta lista:

- **Nueve nueves de disponibilidad**: menos de 32 milisegundos de parada al año.
- **Millones de llamadas simultáneas.**
- **Actualización del software sin cortar el servicio** (clase 148).
- **Y que un fallo en una llamada no afecte a las demás.**

Joe Armstrong y su equipo probaron primero con Prolog —de ahí viene la sintaxis, con los puntos al final de las cláusulas— y acabaron construyendo un lenguaje propio.

Y la decisión de diseño es la que hay que entender: en lugar de intentar escribir código que no falle, **aceptaron que el software falla** y construyeron el sistema para que eso no importe:

> ***"Let it crash"** — deja que falle. **No programes defensivamente**: si algo va mal, que el proceso muera limpiamente y que otro lo reinicie en un estado conocido.*

OTP —*Open Telecom Platform*, 1996— es el conjunto de bibliotecas que hace eso práctico: árboles de supervisión, comportamientos estándar y actualización en caliente.

El resultado se midió: el conmutador **AXD301**, con más de un millón de líneas de Erlang, alcanzó **nueve nueves de disponibilidad** en producción — uno de los datos de fiabilidad más citados de la industria.

Ericsson llegó a **prohibir Erlang internamente** en 1998 para estandarizar en C++; el equipo lo liberó como software libre, y ahí empezó su segunda vida.

🏭 Dónde vive hoy

- **Telecomunicaciones**: sigue en el núcleo de infraestructura de Ericsson y de otros operadores.
- **Mensajería**: **WhatsApp** atendió a cientos de millones de usuarios con un equipo de ingeniería diminuto gracias a Erlang; es el caso más citado.
- **Colas y bases de datos**: **RabbitMQ**, **CouchDB**, **Riak**.
- **Videojuegos en línea y apuestas**: sistemas con muchas conexiones concurrentes y de larga duración.
- **Y como plataforma de Elixir**, que ha traído gente nueva a la misma máquina virtual.

🧠 Lo que enseña: actores, supervisión y aislamiento

Uno, los procesos de Erlang no son hilos (clase 133):

```
Pid = spawn(fun() -> bucle(0) end),     % ← unos cientos de bytes, no un hilo del sistema
Pid ! {sumar, 5},                        % enviar un mensaje: asíncrono, sin bloquear

bucle(Estado) ->
    receive
        {sumar, N} -> bucle(Estado + N);
        {leer, Quien} -> Quien ! Estado, bucle(Estado)
    end.
```

Y las propiedades que los hacen distintos de todo lo demás del curso:

- **No comparten memoria.** Cada proceso tiene **su propio montón y su propio recolector**, así que **no hay pausas globales** y **no puede haber carreras de datos** (clase 136) — por construcción, no por disciplina.
- **Son baratísimos**: cientos de miles o millones en una máquina.
- **Y el planificador es apropiativo**: la máquina virtual **quita el turno** a un proceso que lleva mucho ejecutando, así que **uno lento no bloquea a los demás** — a diferencia de la mayoría de los modelos cooperativos (clase 134).

Dos, la supervisión, que es lo que hace útil el "déjalo fallar":

```
            Supervisor
           /     |     \
     Trabajador  Trab.  Trab.
```

Un supervisor vigila procesos hijos y los reinicia cuando mueren, con una estrategia declarada: reiniciar solo al que murió, reiniciar a todos, o rendirse y morir él también —lo que escala el problema a su propio supervisor—.

Y el efecto práctico es enorme: el código de negocio **no lleva manejo de errores defensivo**. Se escribe el caso correcto, y **la estructura del sistema se ocupa del resto** (clase 116).

Y tres, la distribución transparente:

```
Pid ! Mensaje      % ← funciona igual si Pid está en OTRA MÁQUINA
```

Enviar un mensaje a un proceso remoto es idéntico a enviarlo a uno local. Y aquí la clase 161 obliga a la advertencia: **esa transparencia esconde el fallo de red** —el mismo problema del anexo E de Ada y de CORBA (clase 160)—. Erlang lo compensa porque **el modelo ya asume que las cosas mueren**, así que un fallo de red es un caso más de lo que el supervisor sabe manejar.

🔄 Lo que se ha modernizado

- **JIT en la BEAM** (OTP 24), con mejoras de rendimiento notables.
- **Dialyzer** y las especificaciones de tipo (`-spec`): análisis estático **de tipado exitoso** —solo señala lo que seguro está mal, nunca da falsos positivos (clase 146)—.
- `gen_statem` y los comportamientos modernos de OTP.
- **Actualización de código en caliente** madura, con cambio de versión de módulo y migración de estado (clase 148).
- **Y Elixir**, que ha rejuvenecido el ecosistema entero manteniendo la máquina virtual.

⚙️ Cómo se ejecuta hoy

```
escript main.erl < entrada.txt        # como guion
erl                                    # consola interactiva
rebar3 compile eunit dialyzer          # construcción, pruebas y análisis
```

🧪 El programa de la clase 041 en Erlang

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
#!/usr/bin/env escript

main(_) ->
    Linea = io:get_line(""),
    [P, C, D] = [list_to_float(X) || X <- string:tokens(string:trim(Linea), " ")],
    io:format("Total: ~.2f~n", [P * C * (1 - D)]).
```

Lo que hay que ver.

- `[P, C, D] = ...` **es emparejamiento de patrones**, no asignación: **une la lista con el patrón de tres elementos**, y falla si no encaja. Es la misma idea que en Prolog, de donde viene la sintaxis, y que en Scala.
- `[X || X <- lista]` **es una comprensión de listas**, tomada de Prolog y de Haskell.
- **Las variables empiezan por mayúscula**, como en Prolog — y **se ligan una sola vez**: en Erlang **no hay variables mutables** (clase 102). Reasignar `P` sería un error de patrón.
- `~.2f` **es la directiva de formato** con dos decimales; y las cadenas de Erlang son **listas de números**, lo que explica `list_to_float` y es una peculiaridad histórica que Elixir corrigió.
- **Y este programa no enseña nada del lenguaje real**: Erlang es un sistema de procesos y supervisores, no un guion secuencial. Es un **contrato adaptado** (clase 040).

📚 Fuentes y bibliografía

- Learn You Some Erlang for Great Good! (https://learnyousomeerlang.com/) — **Fred Hébert**, libre en línea; la mejor introducción, con OTP explicado de verdad.
- **Joe Armstrong**, *Programming Erlang*, 2.ª ed., Pragmatic — del creador del lenguaje.
- **Joe Armstrong**, *Making reliable distributed systems in the presence of software errors* (tesis, 2003) — libre; **la mejor exposición que existe del principio "déjalo fallar"**, y lectura recomendada para la clase 116 aunque no se use Erlang.

- Documentación de Erlang/OTP (https://www.erlang.org/docs) — el *Design Principles* sobre árboles de supervisión es material directo de la clase 165.
- **Fred Hébert**, *Erlang in Anger* — libre; qué hacer cuando un sistema en producción se comporta mal (clases 141 y 142).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Elixir · Go · Prolog · Smalltalk · Ada

🔬 Fortran — 1957

El lenguaje con el que se predice el clima. El primer lenguaje de alto nivel de la historia sigue siendo, casi setenta años después, el que ejecutan los supercomputadores más grandes del mundo. No por inercia: por velocidad.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Fortran se ejecuta hoy, y además sigue evolucionando*** *— el estándar vigente es de 2023, hay cuatro compiladores en desarrollo activo y un gestor de paquetes nuevo. No es un lenguaje que sobreviva: es uno que compite.*
>
> *Entra porque* ***deja a la vista un concepto que el núcleo esconde****: el* ***array como ciudadano de primera*** *y el coste real del* ***solapamiento de memoria (aliasing)****. Cuando en Python escribes* `c = a + b` *sobre arrays de NumPy y va rápido, es porque debajo hay código compilado con las garantías que Fortran da por defecto. Ver la versión de Fortran explica* ***por qué*** *NumPy existe y por qué C tuvo que inventar* `restrict` *para acercarse.*

Año	1957 (primer compilador, IBM 704)
Autoría	**John Backus** y su equipo en IBM
Familia	Científica / cálculo numérico y de arrays
Paradigma	Imperativo y procedimental; modular desde F90, OO desde F2003
Tipado	Estático y fuerte, con `implicit none` obligatorio en la práctica moderna
Memoria	Estática y de pila por defecto; `allocatable` gestionado por el compilador
Ejecución	Compilado a nativo, con optimización agresiva
Estado	🟢 **Vivo y en evolución** — estándar 2023; no es solo legacy

---

📜 Historia

En 1954 John Backus propuso a IBM algo que sonaba a herejía: un lenguaje en el que el programador escribiera fórmulas matemáticas y **un traductor generara código máquina tan bueno como el escrito a mano**. Era la parte difícil. Nadie dudaba de que se pudiera traducir; se dudaba de que el resultado fuera aceptablemente rápido, porque en 1954 la programación era ensamblador y el ensamblador era sagrado.

El equipo tardó tres años. **FORTRAN** (*FORmula TRANslating System*) se entregó en 1957 para el IBM 704 y, con él, el primer **compilador optimizador** de la historia: análisis de expresiones, asignación de registros, optimización de bucles. Ese trabajo, más que el lenguaje, es la contribución que fundó la disciplina de construcción de compiladores.

La evolución posterior marca dos épocas claramente distintas:

- **FORTRAN 66 y 77** — la era de las **columnas**. Herencia de la tarjeta perforada: columnas 1–5 para etiquetas, columna 6 para continuación, 7–72 para la sentencia. Variables de seis caracteres, `GO TO` por todas partes, tipado implícito por la inicial del nombre (`I` a `N` eran enteros: de ahí el chiste de que "GOD is REAL, unless declared INTEGER"). Este es el Fortran de la mala fama.
- **Fortran 90 en adelante** — el rediseño. Formato libre, **módulos** con interfaces explícitas, **operaciones sobre arrays completos** (`c = a + b` suma dos matrices sin escribir un bucle), asignación dinámica, punteros, `implicit none` para desactivar el tipado implícito. **2003** trajo orientación a objetos e interoperabilidad estandarizada con C; **2008**, los *coarrays* para paralelismo en el propio lenguaje; **2018** y **2023** siguen añadiendo.

Nota tipográfica útil para leer literatura antigua: hasta el 77 se escribía **FORTRAN** en mayúsculas; desde el 90, **Fortran**. La grafía te dice de qué época habla un texto.

🏭 Dónde sobrevive hoy

- **Modelos climáticos y meteorológicos**: WRF, CESM y los modelos operativos de los servicios meteorológicos nacionales.
- **Dinámica de fluidos computacional (CFD)**: aerodinámica, combustión, turbomaquinaria.
- **Física computacional**: estructura electrónica, física de plasmas, astrofísica, redes cristalinas.
- **Ingeniería estructural**: análisis por elementos finitos.
- **Álgebra lineal de alto rendimiento**: **BLAS** y **LAPACK**, las bibliotecas sobre las que se apoyan NumPy, SciPy, MATLAB, R y buena parte del aprendizaje automático, tienen su implementación de referencia en Fortran. Cuando haces una multiplicación de matrices en Python, hay Fortran compilado debajo.

🧠 Por qué no ha muerto

1. El compilador puede asumir que no hay solapamiento (*aliasing*). Es la razón técnica de fondo. En C, si una función recibe dos punteros, el compilador debe asumir que podrían apuntar a la misma memoria, y eso le impide reordenar y vectorizar libremente. El estándar de Fortran **prohíbe** que dos argumentos de un procedimiento se solapen, así que el compilador optimiza sin esa duda. C tuvo que inventar la palabra clave `restrict` en C99 para recuperar, a mano y bajo responsabilidad del programador, lo que Fortran tiene por defecto.

2. Los arrays son ciudadanos de primera. Un array de Fortran conoce su forma, sus límites (que no tienen por qué empezar en 0) y puede rebanarse (`a(2:5, :)`) o operarse entero. Existen `matmul`, `dot_product`, `sum`, `maxval` y las operaciones elementales en el propio lenguaje, no en una biblioteca. Además, el almacenamiento es **por columnas** (*column-major*), al revés que C — un detalle que hay que tener presente al interoperar.

3. Medio siglo de bibliotecas numéricas validadas. La estabilidad numérica de un solver no se comprueba leyéndolo: se comprueba con décadas de uso en problemas reales. Reescribir LAPACK no es un ejercicio de traducción.

4. El paralelismo está en el ecosistema. OpenMP, MPI y los *coarrays* del estándar cubren desde el multinúcleo hasta el clúster, y los compiladores de NVIDIA y AMD ofrecen descarga a GPU desde directivas. La comunidad HPC no se ha movido porque no ha tenido un motivo de rendimiento para hacerlo.

🔄 Lo que se ha modernizado

Fortran es, de toda esta lista, el caso más claro de lenguaje que **no sobrevive sino que compite**:

- `do concurrent` (2008, ampliado en 2018): un bucle en el que el programador **garantiza** que las iteraciones son independientes. Los compiladores de NVIDIA e Intel lo usan para **descargarlo automáticamente a la GPU** sin una sola directiva ni línea de CUDA. Es paralelismo expresado en el propio lenguaje.
- **Coarrays** (2008): paralelismo distribuido en la sintaxis. `a[3]` significa "la copia de `a` en la imagen 3", con memoria distribuida gestionada por el compilador en lugar de por llamadas a MPI.
- **Interoperabilidad estandarizada con C** (`iso_c_binding`, 2003), que es lo que permite que NumPy, SciPy y R llamen a rutinas Fortran de forma portable y bien definida.
- **Submódulos** (2008): separar la interfaz de la implementación para no recompilar el mundo entero al tocar una función. Un problema de ingeniería moderno, resuelto en el estándar.
- `fpm`, el gestor de paquetes y sistema de construcción, que por fin da a Fortran un flujo equivalente a `cargo` o `npm`.
- **Compiladores nuevos**: **LLVM Flang** entró en el proyecto LLVM, y `nvfortran` y el `flang` de AMD dan acceso directo a GPU. Nadie invierte en escribir compiladores nuevos para un lenguaje muerto.
- **Estándar 2023** vigente, con más de sesenta años de compatibilidad hacia atrás: código de 1977 sigue compilando junto a código con GPU.

⚙️ Cómo se ejecuta hoy

```
# GNU Fortran, parte de GCC
sudo apt-get install -y gfortran

gfortran -O2 total.f90 -o total
echo "15000 2 0.10" | ./total
# Total: 27000.00
```

Compiladores en uso. `gfortran` (GCC, libre y el más extendido), **Intel** `ifx` (oneAPI, el referente en x86 para HPC), **LLVM Flang**, **NVIDIA** `nvfortran` (con descarga a GPU) y **AMD** `flang`. Las extensiones importan: `.f` o `.for` implica **formato fijo** de columnas; `.f90` y posteriores implican **formato libre**. Confundirlas produce errores de sintaxis desconcertantes.

Gestión de paquetes. Históricamente no había ninguna: se copiaban los fuentes. Hoy existe **fpm** (https://fpm.fortran-lang.org/) (*Fortran Package Manager*), que da a Fortran algo parecido a `cargo` o `npm`.

🧪 El programa de la clase 041 en Fortran

```
program total_venta
   implicit none
   real(kind=8) :: precio, cantidad, descuento, total
   character(len=32) :: buffer

   read(*, *) precio, cantidad, descuento
   total = precio * cantidad * (1.0d0 - descuento)

   write(buffer, '(F20.2)') total
   write(*, '(A)') 'Total: ' // trim(adjustl(buffer))
end program total_venta
```

Recorrido, línea a línea.

- `implicit none` es la primera línea de cualquier Fortran moderno serio. Sin ella, una variable no declarada **no es un error**: el compilador le asigna un tipo según su inicial, herencia directa de 1957. Un `total` mal tecleado como `totl` se convertiría en un real nuevo con valor basura, en silencio. `implicit none` apaga esa herencia y convierte el descuido en error de compilación.

- `real(kind=8)` pide un real de 8 bytes (doble precisión). El `kind` es un número entero que identifica la representación; escribir `kind=8` funciona en los compiladores habituales, pero la forma **portable** es `use iso_fortran_env, only: real64` y declarar `real(real64)`. Es la clase de detalle que separa el código que compila aquí del que compila en cualquier sitio.
- `read(*, *)` es lectura **dirigida por lista**: el primer `*` es la unidad (entrada estándar), el segundo el formato (libre). Lee tantos valores como variables haya, separados por espacios o comas, atravesando saltos de línea si hace falta. Es notablemente cómodo comparado con el `scanf` de C.
- `1.0d0` es un literal de **doble precisión**. Escribir `1.0` a secas sería precisión simple, y en una expresión con dobles introduciría una conversión innecesaria. La `d` es la marca; en código moderno se prefiere `1.0_real64`.
- `//` es el operador de **concatenación de cadenas**, no un comentario ni una división. Los comentarios en Fortran libre empiezan por `!`.
- El truco del `buffer`: se escribe el número **a una cadena** con formato `F20.2` (ancho 20, dos decimales), y luego `adjustl` lo empuja a la izquierda y `trim` corta los espacios sobrantes. Se podría escribir `F0.2` para pedir "el ancho mínimo", pero su comportamiento con el cero varía entre compiladores; el rodeo por el buffer es el que da la misma salida en todos.
- Las cadenas de Fortran tienen **longitud fija**: `character(len=32)` reserva 32 caracteres y los rellena con espacios. No hay cadena dinámica sin `allocatable`. De ahí que `trim` aparezca tanto.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En Fortran es…
`import` / `#include`	`use modulo` — con interfaz comprobada por el compilador
`def f(x)` con retorno	`function f(x) result(y)`
`void f(x)`	`subroutine f(x)`
`a[0]`	`a(1)` — paréntesis, y el índice **empieza en 1** por defecto
`for i in range(n)`	`do i = 1, n ... end do`
`c = a + b` sobre listas	`c = a + b` sobre arrays completos — nativo, sin bucle
`numpy.dot(a, b)`	`matmul(a, b)`
`restrict` de C	El comportamiento por defecto
`f"{x:.2f}"`	`write(buf, '(F20.2)') x`

⚠️ Errores comunes al leerlo

- **Índices desde 1.** Y no siempre: `real :: a(0:9)` o `a(-5:5)` son legales. El array lleva sus límites consigo, así que no puedes asumirlos.
- **Column-major.** `a(i, j)` guarda contiguos los que varían en `i`. Recorrer una matriz con el bucle exterior en `i` destruye la localidad de caché. En C es exactamente al revés, y ese es el error clásico al portar código entre ambos.
- **Confundir formato fijo y libre.** Si un `.f` moderno "no compila por sintaxis", casi siempre es que el compilador lo está leyendo como formato fijo y descartando todo más allá de la columna 72.
- `implicit none` **ausente en código antiguo.** Al leerlo, no des por hecho que una variable está declarada: puede existir solo por su inicial.
- **Paso por referencia por defecto.** Un argumento modificado dentro de una subrutina cambia fuera. El Fortran moderno lo documenta con `intent(in)`, `intent(out)` o `intent(inout)`, y usarlos es obligatorio en cualquier código mantenible.

📚 Fuentes y bibliografía

- fortran-lang.org (https://fortran-lang.org/) — el portal actual de la comunidad, con tutoriales, el gestor de paquetes `fpm` y el índice de bibliotecas.
- Documentación de GNU Fortran (https://gcc.gnu.org/onlinedocs/gfortran/) — referencia del compilador libre.
- Intel Fortran Compiler (https://www.intel.com/content/www/us/en/developer/tools/oneapi/fortran-compiler.html) — el compilador de referencia en HPC sobre x86.
- **Michael Metcalf, John Reid, Malcolm Cohen**, *Modern Fortran Explained*, Oxford University Press — la obra de referencia; Reid y Cohen han estado en el comité del estándar, así que explica el porqué de cada decisión.
- **Milan Curcic**, *Modern Fortran*, Manning, 2020 — enfoque práctico y actual, orientado a quien llega desde Python o C.
- **Ronald Hanson, Tim Hopkins**, *Numerical Computing with Modern Fortran*, SIAM — el puente entre el lenguaje y el cálculo numérico serio.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Ada · C · Assembler

🔵 F# — 2005

F# es **OCaml sobre .NET**, y es la demostración de que un lenguaje funcional puede vivir en un ecosistema empresarial sin renunciar a nada: usa las mismas bibliotecas que C#, interopera sin fricción, y **muchas de las mejores ideas de C# de la última década llegaron desde aquí**.

> 🎯 ***Por qué está en este programa***
>
> *F# es un **primo de la familia .NET** (Atlas), cuyo representante en el núcleo es C#, y también de la **familia funcional tipada (ML)**.*
>
> *Aporta al programa la **inferencia de tipos completa al estilo Hindley-Milner** —tipado estático sin escribir casi ningún tipo— y **los tipos de datos algebraicos con emparejamiento exhaustivo** (clase 100), que son las dos ideas de ML que hoy están llegando a todos los lenguajes.*

Año	2005; **1.0** en 2010; hoy en **F# 9**, con .NET
Autoría	**Don Syme**, Microsoft Research Cambridge — también autor de los genéricos de .NET
Familia	.NET y funcional tipada (ML); descendiente directo de OCaml
Paradigma	**Funcional primero**, con OO completo cuando hace falta
Tipado	**Estático con inferencia total**; tipos algebraicos y unidades de medida
Memoria	La del CLR: recolección de basura
Ejecución	Bytecode IL sobre el CLR, con JIT o AOT nativo
Estado	🟢 **Vivo y minoritario**; muy usado en finanzas y análisis de datos

---

📜 Historia

Don Syme trabajaba en Microsoft Research y tiene un mérito que conviene conocer: **diseñó e implementó los genéricos de .NET —reificados**, a diferencia de los de Java (clase 108)— y lo hizo, en parte, **porque los necesitaba para poder llevar ML a la plataforma**.

Es una historia poco común: **una característica del ecosistema entero existe porque alguien quería hacer un lenguaje funcional en él**.

F# apareció en **2005** como puerto de OCaml sobre .NET, y se convirtió en producto oficial de Microsoft en **2010**. Desde entonces es de código abierto y multiplataforma, con la **F# Software Foundation** implicada en su gobierno.

Y su influencia sobre su hermano mayor es grande y está documentada: **los registros, el emparejamiento de patrones, los tipos con nulabilidad, `async` / `await` y las expresiones `switch` de C# vienen, en buena parte, de F#**. Es el laboratorio de la plataforma.

🏭 Dónde vive hoy

- **Finanzas cuantitativas**: es su nicho más fuerte; bancos y fondos lo usan para modelos y valoración, donde la corrección importa más que la popularidad.
- **Análisis de datos y ciencia**: con los **proveedores de tipos**, que son su característica más original.
- **Servicios de fondo y APIs**: con Giraffe o Falco sobre ASP.NET Core.
- **Cálculo y simulación**, y como lenguaje de guion analítico dentro de organizaciones .NET.

🧠 Lo que enseña: inferencia total y tipos algebraicos

Uno, la inferencia Hindley-Milner:

```
let sumar a b = a + b            // ← ni un tipo escrito
// el compilador deduce: int -> int -> int

let procesar lista =
    lista |> List.filter (fun x -> x > 0) |> List.sum
// deduce: int list -> int
```

Tipado estático completo sin escribir tipos. Es más fuerte que la inferencia local de C# con `var` o de Java: **F# infiere las firmas de las funciones**, no solo las variables locales.

Dos, los tipos algebraicos con exhaustividad:

```
type Forma =
    | Circulo of radio: float
    | Rectangulo of ancho: float * alto: float

let area f =
    match f with
    | Circulo r -> System.Math.PI * r * r
    | Rectangulo (a, b) -> a * b
    // ← si falta un caso, el compilador AVISA
```

Y la combinación de las dos cosas es lo que hace productivo el estilo: se modela el dominio con tipos que **hacen imposible representar un estado inválido**, y el compilador comprueba que se han tratado todos los casos (clase 166).

Y tres, dos cosas que F# tiene y casi nadie más:

```
[<Measure>] type m
[<Measure>] type s
let velocidad = 100.0<m> / 9.58<s>    // float<m/s>
// let error = 100.0<m> + 9.58<s>       ← ✗ NO COMPILA
```

Las unidades de medida comprobadas en compilación, sin coste en ejecución. Es exactamente lo que la clase 166 señalaba con el Mars Climate Orbiter: **el contrato de las unidades, en el tipo**.

```
type Datos = CsvProvider<"ventas.csv">     // ← el TIPO se genera del fichero
let filas = Datos.Load("ventas.csv")
filas.Rows |> Seq.sumBy (fun r -> r.Importe)   // con autocompletado y comprobación
```

Los proveedores de tipos leen un esquema —CSV, JSON, SQL, un servicio web— **en tiempo de compilación** y generan los tipos. Es metaprogramación (clase 123) aplicada al problema de la clase 170: **el desajuste de impedancia resuelto generando el tipo desde el esquema**.

🔄 Lo que se ha modernizado

- **F# 6-9**: `task { }` para interoperar con el `async` de C#, expresiones de índice, y mejoras grandes de rendimiento del compilador.
- **AOT nativo** sobre .NET, con binarios de arranque instantáneo (clase 174).
- **Fable**: compila F# **a JavaScript**, y también a Python, Rust y Dart — un objetivo múltiple poco común (clase 162).
- `dotnet fsi`: consola interactiva de primer nivel para explorar datos, al estilo de Lisp (clase 124).
- **Y una comunidad activa** que empuja las novedades hacia C#, que es donde acaban llegando.

⚙️ Cómo se ejecuta hoy

```
dotnet fsi main.fsx < entrada.txt         # ejecutar como guion
dotnet run                                  # como proyecto

dotnet fsi                                  # consola interactiva
dotnet test                                  # pruebas, con Expecto o xUnit (clase 139)
```

🧪 El programa de la clase 041 en F#

```
let [| precio; cantidad; descuento |] =
    stdin.ReadLine().Split(' ') |> Array.map float
let total = precio * cantidad * (1.0 - descuento)
printfn "Total: %.2f" total
```

Lo que hay que ver.

- `|>` **es el operador de canalización**, y es la marca de la casa: `x |> f` es `f x`. Permite leer la transformación **de izquierda a derecha, en el orden en que ocurre** —en lugar de anidada de dentro afuera— y es lo que después adoptaron Elixir, R y JavaScript en propuesta.
- `let [| a; b; c |] = ...` **es emparejamiento de patrones sobre un arreglo**, igual que en Scala: si la línea no trajera tres campos, fallaría.
- `1.0` **con decimal no es un capricho**: F# **no convierte números implícitamente**, así que `1 - descuento` con un entero **no compila**. Es la misma disciplina de Go y Rust (clase 100).
- `printfn "Total: %.2f"` **está comprobado en compilación**: el `%.2f` **exige un** `float`, y el compilador lo verifica. Es más fuerte que el `printf` de C, que no comprueba nada (clase 142), y que el de C#.
- **Y no hay clase, ni** `main`**, ni llaves**: la sangría marca el bloque, como en Python.

📚 Fuentes y bibliografía

- F# for Fun and Profit (https://fsharpforfunandprofit.com/) — **Scott Wlaschin**; probablemente el mejor sitio para aprender programación funcional aplicada, con o sin F#. Su serie sobre **modelar el dominio con tipos** es directamente material de la clase 166.

- Documentación oficial de F# (https://learn.microsoft.com/dotnet/fsharp/) y fsharp.org (https://fsharp.org/).
- **Scott Wlaschin**, *Domain Modeling Made Functional*, Pragmatic — cómo llevar el diseño dirigido por el dominio a tipos que no permiten estados inválidos.
- **Don Syme, Adam Granicz, Antonio Cisternino**, *Expert F#*, Apress — la referencia del autor del lenguaje.
- Fable (https://fable.io/) — para el objetivo JavaScript y los demás.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: OCaml · C# · Haskell · Elm · Scala

🐭 Go — 2009

Go se diseñó a partir de una queja concreta: **los programas grandes tardaban demasiado en compilar y demasiado en entenderse**. Su respuesta fue radical y poco habitual — **quitar características** — y el resultado es un lenguaje que se aprende en un fin de semana y que mueve buena parte de la infraestructura de Internet.

> 🎯 ***Por qué está en este programa***
>
> ***Go es uno de los diez lenguajes del núcleo** y, junto a Rust, el **representante de la familia de sistemas** (Atlas).*
>
> *Aporta al programa el modelo de concurrencia que ningún otro del núcleo enseña: **CSP con gorrutinas y canales** (clases 133 y 134), la idea de Hoare de 1978 llevada a un lenguaje mayoritario. Y aporta una lección de diseño que el curso repite desde varios ángulos (clases 155 y 164): **un lenguaje pequeño es una decisión, y lo que prohíbe es lo que permite prometer**.*

Año	2009; **1.0** en 2012, con **promesa de compatibilidad**; **1.18** con genéricos (2022)
Autoría	**Robert Griesemer, Rob Pike, Ken Thompson**, Google
Familia	Sistemas; herencia de C, Pascal/Oberon (Wirth) y de **CSP** (Hoare)
Paradigma	Imperativo y concurrente; composición en vez de herencia
Tipado	**Estático y fuerte**, con **interfaces estructurales** e inferencia local
Memoria	Recolección de basura de **latencia muy baja**, concurrente
Ejecución	Compilado a nativo, **estático por defecto**, con compilación rapidísima
Estado	🟢 **Estándar de facto** en infraestructura, nube y herramientas de línea de comandos

---

📜 Historia

La historia de Go empieza, según cuenta Rob Pike, **esperando a que compilara un binario de C++ en Google**. Tardaba tanto que en esa espera empezó la conversación que llevó al lenguaje.

Ken Thompson —coautor de Unix y de C—, **Rob Pike** —de Plan 9 y de UTF-8— y **Robert Griesemer** —de la máquina virtual de JavaScript V8 y discípulo de Wirth— diseñaron en **2007** un lenguaje con tres objetivos explícitos:

1. **Compilar rapidísimo**, incluso proyectos enormes.
2. **Ser fácil de leer** por alguien que no lo escribió, en una empresa con miles de programadores.

3. **Hacer la concurrencia natural**, porque los servidores ya eran multinúcleo.

Y para conseguirlo **quitaron cosas**: sin herencia de clases, sin excepciones, sin genéricos —durante trece años—, sin sobrecarga de operadores ni de funciones, sin macros, sin constructores, sin aritmética de punteros.

Go 1.0 (2012) vino con una promesa que ha cumplido: **compatibilidad hacia atrás**. Código de 2012 compila hoy. Es la misma disciplina de Tcl y del mainframe (clase 154), y es una de las razones de su adopción industrial.

Go 1.18 (2022) añadió **genéricos** tras una discusión pública de una década — y el resultado es deliberadamente limitado, coherente con la filosofía.

🏭 Dónde vive hoy

- **Infraestructura de la nube**: **Docker, Kubernetes, Terraform, Prometheus, etcd, containerd**. Prácticamente toda la capa de orquestación moderna está escrita en Go (clase 174).
- **Servicios de red y API**: por el modelo de concurrencia y por el despliegue simple.
- **Herramientas de línea de comandos**: el binario estático sin dependencias es ideal (clase 167).
- **Bases de datos y almacenamiento**: CockroachDB, InfluxDB, Minio.
- **Y como sustituto de Python en tareas de administración** donde el rendimiento o el despliegue importan.

🧠 Lo que enseña: CSP, y por qué es distinto de los hilos

Go implementa **CSP —*Communicating Sequential Processes*, de Tony Hoare, 1978**— con dos primitivas:

```
go trabajar(datos)              // una GORRUTINA: unos pocos KB, no un hilo del sistema
canal <- valor                   // enviar
valor := <-canal                  // recibir, bloqueando hasta que haya algo
```

Y el lema del lenguaje resume la idea:

> ***"No te comuniques compartiendo memoria; comparte memoria comunicándote."***

Es lo contrario del modelo de hilos con cerrojos de Java y C++ (clase 135): en lugar de proteger un dato compartido, **el dato se pasa por un canal** y solo una gorrutina lo tiene a la vez.

Y las gorrutinas son ligeras de verdad: arrancan con unos pocos kilobytes de pila que crece sola, y el planificador de Go multiplexa **cientos de miles** sobre unos pocos hilos del sistema.

> ***Y la honestidad exige decir el límite** (clase 136): **Go no impide las carreras de datos**. Los canales son la forma recomendada, no la obligatoria, y compartir un mapa entre gorrutinas sin sincronizar es un error que compila. Por eso existe `go test -race`, el detector de carreras, y por eso conviene usarlo siempre. Rust resolvió el mismo problema por el otro camino: **hacerlo imposible en el sistema de tipos**.*

Y el segundo concepto que Go enseña es **la interfaz estructural**:

```
type Escritor interface { Write(p []byte) (int, error) }
```

Cualquier tipo con ese método la cumple, sin declararlo — como TypeScript y a diferencia de Java (clase 112). Eso permite definir la interfaz **donde se consume**, no donde se implementa, que es la inversión de dependencias hecha idioma.

🔄 Lo que se ha modernizado

- **Genéricos (1.18)** con restricciones por conjuntos de tipos — limitados a propósito.
- **Errores envueltos** con `%w`, `errors.Is` y `errors.As`: Go **no tiene excepciones** y devuelve errores como valores (clase 116), y esta capa da la información que faltaba.
- **Espacios de trabajo y módulos** con `go.sum`: fichero de bloqueo con sumas de comprobación (clase 143) y **suma de comprobación verificable en un registro público**, que es una defensa real de cadena de suministro (clase 153).
- **Optimización guiada por perfiles (PGO)** desde 1.21: compilar usando datos de ejecución real.
- `log/slog`: registro estructurado en la biblioteca estándar (clase 142).
- **Y una discusión abierta y sana** sobre `for` con variables por iteración, corregida en 1.22 tras años de ser la fuente número uno de errores del lenguaje.

⚙️ Cómo se ejecuta hoy

```
go run main.go < entrada.txt        # el comando de la clase 041
go build -o venta .                  # binario estático, sin dependencias

go vet ./... && gofmt -l .           # calidad: gofmt NO tiene opciones (clase 146)
go test -race ./...                   # pruebas con detector de carreras (clase 136)
```

> ***`gofmt` merece la mención: no tiene opciones de configuración.*** *La comunidad renunció a la discusión de estilo eliminando la posibilidad de tenerla, y es probablemente la decisión de la clase 146 mejor ejecutada de cualquier ecosistema.*

🍼 El programa de la clase 041 en Go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strconv"
    "strings"
)

func main() {
    reader := bufio.NewReader(os.Stdin)
    linea, _ := reader.ReadString('\n')
    campos := strings.Fields(linea)

    // Go: tipado estático explícito; conversión float64(cantidad) obligatoria.
    precioUnitario, _ := strconv.ParseFloat(campos[0], 64)
    cantidad, _ := strconv.Atoi(campos[1])
    descuento, _ := strconv.ParseFloat(campos[2], 64)

    subtotal := precioUnitario * float64(cantidad)
    total := subtotal * (1 - descuento)

    fmt.Printf("Total: %.2f\n", total)
}
```

Lo que hay que ver.

- `float64(cantidad)` **es obligatorio.** Go **no convierte números implícitamente**, ni siquiera de `int` a `float64`. Es más verboso y elimina una familia entera de sorpresas (clase 100) —lo contrario de JavaScript y PHP—.
- **El `_` que descarta el error aparece cuatro veces, y en código real no debería.** Go devuelve errores como valores y **obliga a mirarlos**; ignorarlos con `_` es una decisión explícita y visible, que es justamente la virtud del modelo (clase 116).
- `strings.Fields` parte por cualquier espacio en blanco, como `split ' '` de Perl.
- **No hay `try` ni `catch`**: si algo falla, se ve en la línea donde falla. Es la decisión más discutida del lenguaje y la más coherente con su filosofía de que **el flujo de control se lee**.

📚 Fuentes y bibliografía

- go.dev (https://go.dev/doc/) — *A Tour of Go*, *Effective Go* y la referencia; de lo mejor documentado de esta lista.
- El blog de Go (https://go.dev/blog/) — los artículos sobre concurrencia, errores y genéricos explican el porqué de cada decisión.
- **Alan Donovan, Brian Kernighan**, *The Go Programming Language*, Addison-Wesley — el "K&R de Go", escrito por uno de los autores del original.
- **Jon Bodner**, *Learning Go*, 2.ª ed., O'Reilly — actualizado a genéricos; enseña los idiomas del lenguaje, no solo la sintaxis.
- **Katherine Cox-Buday**, *Concurrency in Go*, O'Reilly — los patrones de canales y gorrutinas.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Rust · C · Zig · Erlang · Pascal

🎸 Groovy — 2003

Groovy es **Java sin la ceremonia**: casi todo programa Java es un programa Groovy válido, y a partir de ahí se puede ir quitando —los tipos, los puntos y coma, los `getters`— hasta que quede algo que parece Python o Ruby. Su mayor éxito no fue como lenguaje de aplicación, sino **como lenguaje de configuración programable**.

> 🎯 ***Por qué está en este programa***
>
> *Groovy es un **primo de la familia JVM** (Atlas), cuyo representante en el núcleo es Java.*
>
> *Aporta al programa el caso más visible de **lenguaje incrustado como DSL** (clase 163): **los `build.gradle` y los `Jenkinsfile` que millones de personas editan sin saber que están escribiendo Groovy**. Y aporta el concepto de **tipado dinámico y estático en el mismo lenguaje, elegible por anotación** (clase 146).*

Año	2003; **1.0** en 2007; **2.0** con compilación estática (2012); **4.x** actual
Autoría	**James Strachan** y **Bob McWhirter**; hoy proyecto de Apache
Familia	JVM; sintaxis de Java con influencia de Python, Ruby y Smalltalk
Paradigma	OO y funcional; muy orientado a construir DSL
Tipado	**Dinámico por defecto**, con `@CompileStatic` para comprobación estática
Memoria	La de la JVM
Ejecución	Bytecode JVM, con despacho dinámico; estático con la anotación
Estado	🟢 **Muy usado como DSL** (Gradle, Jenkins); poco como lenguaje de aplicación

---

📜 Historia

James Strachan empezó Groovy en **2003** con una idea directa: **la JVM es excelente y Java es verboso**; hagamos un lenguaje dinámico que se apoye en toda la biblioteca de Java y que se lea como Python.

El proyecto pasó por dificultades —Strachan lo abandonó, y en 2009 comentó que si hubiera conocido Scala antes probablemente no lo habría empezado— y lo rescató la comunidad, con SpringSource y después **Apache**.

Su éxito real llegó por un camino lateral, que es lo interesante de esta ficha:

- **Gradle (2007)** eligió Groovy como lenguaje de sus ficheros de construcción, y Gradle acabó siendo el sistema de construcción de **Android** y de buena parte del mundo JVM.
- **Jenkins (2016)** adoptó Groovy para sus canalizaciones declarativas — los `Jenkinsfile` que definen la integración continua de miles de empresas (clase 147).
- **Grails** llevó la filosofía de Rails a la JVM.

Y de ahí una situación curiosa: **muchísima gente edita Groovy a diario sin saber que lo hace**.

Groovy 2.0 (2012) añadió `@CompileStatic` y `@TypeChecked`, que permiten pedir comprobación estática y rendimiento de Java **por clase o por método**.

🏭 Dónde vive hoy

- **Gradle**: los `build.gradle` clásicos son Groovy (hoy compite con el DSL de Kotlin).
- **Jenkins**: las canalizaciones, declarativas o de guion, son Groovy (clase 171).
- **Pruebas**: **Spock**, un marco de pruebas con una sintaxis de especificación excelente, muy usado incluso en proyectos Java (clase 139).
- **Guiones de administración en la JVM**, con acceso directo a todas las bibliotecas Java.
- **Grails**, en aplicaciones heredadas y en algunos proyectos nuevos.

🧠 Lo que enseña: un lenguaje que se convierte en configuración

Este bloque es Groovy, y casi nadie lo piensa así:

```
plugins { id 'java' }

repositories { mavenCentral() }

dependencies {
    implementation 'org.apache.commons:commons-lang3:3.14.0'
    testImplementation 'junit:junit:4.13.2'
}
```

Cada línea es una llamada a un método con un bloque — el mecanismo es el de la clase 163: **una sintaxis que permite omitir paréntesis y pasar bloques hace que el código parezca configuración**.

Y la consecuencia buena: cuando la configuración se queda corta, **no hay que salir a otro lenguaje**, porque ya se está en uno completo. Se puede poner un `if`, un bucle o una función.

Y la consecuencia mala, que la clase 163 advierte: un `build.gradle` puede ejecutar cualquier cosa. **La configuración deja de ser declarativa**, se vuelve imposible de analizar sin ejecutarla, y es una superficie de ataque en la cadena de suministro (clase 153).

Es exactamente la tensión que la clase 163 plantea, y Groovy es su mejor ejemplo cotidiano.

Y el segundo concepto es el tipado elegible:

```
def suma(a, b) { a + b }                    // dinámico: se resuelve en ejecución

@CompileStatic
int sumaRapida(int a, int b) { a + b }      // ← estático: bytecode como el de Java
```

El mismo lenguaje, dos modos, decididos con una anotación. Es tipado gradual (clase 146) llevado al nivel del método.

🔄 Lo que se ha modernizado

- `@CompileStatic` y `@TypeChecked` con extensiones de comprobación propias.
- **Groovy 4** sobre Java 17+, con registros, `switch` de expresión y soporte de las novedades de la JVM.
- `invokedynamic` como mecanismo de despacho, que acercó el rendimiento del modo dinámico al de Java.
- **Spock** como marco de pruebas de referencia, con bloques `given/when/then` y **aserciones que muestran todos los valores intermedios al fallar** — lo mismo que Catch2 hace en C++ (clase 139).
- **Y la competencia sana con el DSL de Kotlin en Gradle**, que ha empujado a los dos.

⚙️ Cómo se ejecuta hoy

```
groovy main.groovy < entrada.txt         # el comando de la clase 041
groovyc Venta.groovy && java -cp .:$GROOVY_HOME/lib/groovy.jar Venta

gradle build                             # ← esto ejecuta Groovy (o Kotlin)
```

🍼 El programa de la clase 041 en Groovy

```
def (precio, cantidad, descuento) = System.in.newReader().readLine().split(' ')*.toDouble()
def total = precio * cantidad * (1 - descuento)
printf("Total: %.2f%n", total)
```

Lo que hay que ver.

- **Una sola línea hace lo que en Java ocupa cinco**, y usa **exactamente las mismas clases**: `System.in`, `BufferedReader`, `String.split`. La interoperabilidad es total.
- `*.toDouble()` **es el operador de expansión (*spread*)**: aplica el método **a cada elemento** de la colección. Es el `map` de otros lenguajes convertido en sintaxis, y no existe en Java.
- `def` **no significa "sin tipo"**: significa **tipo dinámico**. Con `@CompileStatic` estas mismas líneas se comprobarían en compilación.
- `printf` **con** `%n` en lugar de `\n` — el separador de línea de la plataforma, herencia de Java.
- **Y no hay clase ni** `main`: un guion Groovy se ejecuta tal cual, lo que es la razón de que sirva como lenguaje de configuración.

📚 Fuentes y bibliografía

- groovy-lang.org (https://groovy-lang.org/documentation.html) — documentación oficial de Apache Groovy; el apartado de DSL es directamente material de la clase 163.
- Guía de Gradle (https://docs.gradle.org/current/userguide/userguide.html) — para ver el lenguaje en su uso real.
- Spock Framework (https://spockframework.org/) — el marco de pruebas; merece conocerse aunque el proyecto sea Java (clase 139).
- **Ken Kousen**, *Making Java Groovy*, Manning — cómo se combinan los dos en un proyecto real.
- **Dierk König et al.**, *Groovy in Action*, 2.ª ed., Manning — la referencia completa.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Java · Kotlin · Ruby · Python · Tcl

λ Haskell — 1990

Haskell es el lenguaje del que la industria lleva treinta años copiando ideas sin adoptarlo. Las funciones de primera clase, la inferencia de tipos, las lambdas, `Option` / `Maybe`, las colecciones inmutables, los tipos algebraicos y `async` **estaban aquí antes de estar en ningún lenguaje mayoritario** — y su lema no oficial lo resume: *"evitar el éxito a toda costa"*.

> 🎯 ***Por qué está en este programa***
>
> *Haskell es el representante de la **familia funcional tipada (ML)** (Atlas), que **no tiene representante en el núcleo** — su influencia llega al curso a través de Rust, Scala, F# y Elm.*
>
> *Aporta al programa dos conceptos que ningún otro lenguaje del Atlas enseña igual: **la evaluación perezosa por defecto** (clase 115) y **la pureza con efectos en el sistema de tipos** (clase 118).*

Año	1990; **Haskell 98** el estándar de referencia; **GHC 2021/2024** los conjuntos actuales
Autoría	**Comité académico** (Hudak, Peyton Jones, Wadler, Hughes y otros)
Familia	Funcional tipada (ML); nombrado por **Haskell Curry**, lógico
Paradigma	**Funcional puro**
Tipado	**Estático, inferido (Hindley-Milner extendido)**, con clases de tipos
Memoria	Recolección de basura generacional, muy afinada para asignación rápida
Ejecución	Compilado a nativo con **GHC**, vía su propia representación intermedia
Estado	🟢 **Vivo, minoritario y muy influyente**

📜 Historia

A finales de los ochenta había **más de una docena de lenguajes funcionales perezosos**, cada uno de un grupo de investigación, y ninguno con masa crítica. En **1987**, en una conferencia en Portland, se formó un comité para hacer **uno solo**, abierto y estándar.

El resultado, en **1990**, fue **Haskell** — con el nombre del lógico **Haskell Curry**, cuyo trabajo sobre lógica combinatoria fundamenta el cálculo lambda tipado.

El problema que tuvieron que resolver era serio: **en un lenguaje puro y perezoso, ¿cómo se hace entrada y salida?** Si una función no puede tener efectos y el orden de evaluación no está determinado, imprimir en pantalla es un problema conceptual, no técnico.

La solución, hacia **1992**, fue **la mónada** `IO` —trabajo de Eugenio Moggi y Philip Wadler—, y es la idea más influyente y peor explicada de la informática moderna: **los efectos se representan como valores de un tipo, y se componen**.

GHC —el compilador de Glasgow— se convirtió en la implementación de referencia y en un laboratorio de investigación de tipos, del que han salido las extensiones que después llegaron a otros lenguajes: **GADT, familias de tipos, tipos de rango superior,** `deriving` **automático,** `Applicative`.

Y el lema *"avoid success at all costs"* —una broma de Simon Peyton Jones con doble lectura— refleja una postura real: **priorizar la corrección del diseño sobre la adopción**, para poder seguir cambiando cosas.

🏭 Dónde vive hoy

- **Finanzas**: Standard Chartered tiene un equipo grande escribiendo Haskell para modelos de derivados; Barclays y otros lo han usado.
- **Verificación y compiladores**: **Agda**, **Idris**, **Elm**, **PureScript** y buena parte de las herramientas de análisis formal están escritas en Haskell.
- **Criptomonedas**: Cardano y varios proyectos con requisitos de corrección alta.
- **Herramientas**: **Pandoc** —el conversor universal de documentos, que probablemente usas sin saberlo—, ShellCheck, Hasura, Semantic (GitHub).
- **Y la academia**, donde sigue siendo la lengua franca de la investigación en lenguajes.

🧠 Lo que enseña: pereza y efectos en el tipo

Uno, la evaluación perezosa por defecto (clase 115):

```
naturales = [1..]                    -- una lista INFINITA
primeros10 = take 10 naturales        -- [1,2,3,4,5,6,7,8,9,10]

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)   -- ← se define en términos de sí misma
```

Nada se calcula hasta que se necesita, así que una estructura infinita es un valor normal. Eso permite **separar la generación del consumo** —el productor no sabe cuánto se va a consumir— y es la misma idea que los generadores de Python, los iteradores perezosos de Rust y los rangos de D, **con la diferencia de que aquí es la norma y no la excepción**.

Y el coste hay que decirlo: la pereza hace **muy difícil razonar sobre el consumo de memoria**. Un acumulador que no se fuerza construye una cadena de cálculos pendientes que puede agotar la memoria — el problema clásico de Haskell, que se resuelve con anotaciones de rigor (`seq`, `!`).

Dos, los efectos en el tipo:

```
longitud :: String -> Int             -- pura: mismo argumento, mismo resultado, SIEMPRE
leerFichero :: FilePath -> IO String    -- ← el IO en el tipo DECLARA que hay efectos
```

El tipo dice si una función toca el mundo. No es una convención ni una anotación opcional: **una función pura no puede llamar a una con efectos** sin que aparezca en su firma.

Y eso da lo que la clase 118 busca: **al leer una firma se sabe qué puede hacer la función**, y el compilador lo garantiza. Es la versión más fuerte de la idea que Fortran tiene con `pure` y D con `@nogc`.

Y tres, las clases de tipos, que son el sistema de polimorfismo más limpio de esta lista:

```
class Mostrable a where
    mostrar :: a -> String

instance Mostrable Bool where
    mostrar True = "sí"
    mostrar False = "no"
```

Se añade un comportamiento a un tipo existente sin tocarlo y sin herencia — lo que Rust llama rasgos, Scala implementa con implícitos y Go aproxima con interfaces estructurales (clase 112).

🔄 Lo que se ha modernizado

- **GHC2021/GHC2024**: conjuntos de extensiones activados por defecto, que acaban con la lista de `{-# LANGUAGE ... #-}` al principio de cada fichero.
- **Tipos dependientes por partes**: `DataKinds`, `TypeFamilies`, `LinearTypes` — el camino hacia lo que Idris y Agda tienen completo.
- **Herramientas por fin buenas**: **GHCup**, **Stack**, **Cabal** con fichero de bloqueo (clase 143), y **HLS**, el servidor de lenguaje, que resolvió la peor queja histórica.
- **`Text` y `ByteString`** en lugar de `String` como lista de caracteres, que era una fuente clásica de mal rendimiento (clase 093).
- **Y el ecosistema de efectos**: `mtl`, `effectful`, `polysemy` — formas de componer efectos que van más allá de las mónadas anidadas.

⚙️ Cómo se ejecuta hoy

```
runghc main.hs < entrada.txt          # ejecutar directamente
ghc -O2 main.hs -o venta               # compilar optimizado
ghci                                    # consola interactiva

cabal test && hlint src                 # pruebas y análisis (clases 139 y 146)
```

🧪 El programa de la clase 041 en Haskell

Haskell **no está en el `primos.md` de la clase 041**, así que esta versión se escribe aquí y **no está verificada en CI** — como el resto de los primos de lectura (clase 040).

```
import Text.Printf (printf)

main :: IO ()
main = do
    linea <- getLine
    let [precio, cantidad, descuento] = map read (words linea) :: [Double]
    printf "Total: %.2f\n" (precio * cantidad * (1 - descuento))
```

Lo que hay que ver.

- `main :: IO ()` declara en el tipo que este programa **hace entrada y salida**. Es la única parte impura, y está marcada.
- `<-` **no es una asignación**: extrae el valor de dentro de `IO`. Y `let` sí liga un nombre a un valor puro. **Esa distinción es todo el modelo de efectos** en una línea.
- `map read (words linea)` es puro: parte y convierte sin tocar nada. **Y** `:: [Double]` **es necesario** porque `read` es polimórfico —podría leer enteros— y aquí hay que decidir; es un caso donde la inferencia no basta.
- **La desestructuración** `[a, b, c]` es emparejamiento de patrones sobre una lista, como en Scala: si hubiera cuatro campos, fallaría en ejecución.
- **Y** `printf` **está tipado**: el compilador comprueba que el `%.2f` recibe un `Double`, mediante una clase de tipos variádica — un truco de tipos que casi ningún lenguaje puede hacer (clase 142).

📚 Fuentes y bibliografía

- Learn You a Haskell for Great Good! (http://learnyouahaskell.com/) — libre en línea; la introducción más amable que existe, aunque algo anticuada.
- Haskell from First Principles (https://haskellbook.com/) — **Allen y Moronuki**; largo, exigente y probablemente el mejor camino para aprenderlo de verdad.
- Documentación de GHC (https://downloads.haskell.org/ghc/latest/docs/users_guide/) — imprescindible para las extensiones de tipos.
- **Simon Peyton Jones**, charlas y artículos — sobre todo *A History of Haskell: Being Lazy with Class* (HOPL III), que cuenta el diseño desde dentro.
- **Graham Hutton**, *Programming in Haskell*, 2.ª ed., Cambridge — el libro de texto académico de referencia, corto y preciso.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: OCaml · F# · Elm · Scala · Rust

➗ J — 1990

J es **APL escrito con el teclado que ya tienes**: el mismo Kenneth Iverson, veinticuatro años después, sustituyó los símbolos matemáticos por combinaciones de caracteres ASCII. Resolvió el problema del teclado y creó otro — porque `+/ % #` es tan denso como `+/ ÷ ≢`/y bastante menos evocador.

> 🎯 ***Por qué está en este programa***
>
> *J es un* ***primo de la familia array / científica*** *(Atlas), junto a APL, R, Julia, MATLAB y Fortran.*
>
> *Aporta al programa* ***la programación tácita*** *—componer funciones* ***sin nombrar los argumentos*** *—, que es una idea que solo esta familia y Haskell llevan al extremo (clase 115). Y aporta un caso de estudio de la clase 175:* ***qué se gana y qué se pierde al cambiar la notación de un lenguaje****.*

Año	1990; libre (GPL) desde 2011
Autoría	**Kenneth E. Iverson** y **Roger Hui** — el mismo autor de APL
Familia	Array / científica; sucesor directo de APL
Paradigma	**Funcional y por arreglos**, con programación **tácita**
Tipado	Dinámico; **todo es un arreglo**, con rango y forma
Memoria	Recolección de basura
Ejecución	Interpretado, con un núcleo en C muy compacto
Estado	🟡 **Nicho pequeño y estable**; finanzas, análisis y comunidad de entusiastas

---

📜 Historia

Kenneth Iverson tenía casi setenta años cuando, en 1989, se propuso rehacer APL con lo aprendido en treinta años. Le acompañó **Roger Hui**, y el diseño se publicó en **1990**.

Los objetivos eran tres, y merecen conocerse porque son decisiones de diseño puras:

1. **Solo caracteres ASCII**: nada de teclados especiales ni de fuentes raras.
2. **Un modelo de arreglos más regular** que el de APL, con reglas de rango uniformes.
3. **Y hacer de primera clase la programación tácita**, que en APL existía y estaba poco desarrollada.

La sustitución de símbolos siguió un patrón sistemático: **una letra o un símbolo, opcionalmente seguido de un punto o de dos puntos**, forma un verbo. Así, `#` es la cuenta, `#.` es la conversión de base, `#:` es la representación en base.

Y aquí está la lección de la clase 175: **el cambio resolvió un problema real y no trajo la adopción**. Quien ya usaba APL prefería sus símbolos —cada uno tenía forma propia y era reconocible—, y quien no lo usaba **seguía viendo una línea de puntuación indescifrable**.

J es libre desde 2011, y su comunidad es pequeña y muy dedicada.

🏭 Dónde vive hoy

- **Análisis financiero y actuarial**, en organizaciones de la órbita APL.
- **Análisis exploratorio de datos** por quien viene de la tradición de arreglos.
- **Enseñanza de la programación por arreglos**, por ser libre y no necesitar teclado especial.
- **Y como referencia de diseño**: **BQN** y **Uiua**, los lenguajes de arreglos recientes, parten de las lecciones de J tanto como de las de APL.

🧠 Lo que enseña: la programación tácita

Esta es la aportación, y es genuinamente distinta de todo lo demás del Atlas.

La media aritmética, en J:

```
media =: +/ % #
```

Y hay que leerlo con cuidado, porque no hay ningún argumento escrito: `+/` es "sumar todo", `#` es "contar", y `%` es dividir. **La expresión `+/ % #` es un *fork*: aplica `+/` y `#` al mismo argumento y combina los resultados con `%`.**

```
   media 1 2 3 4 5
3
```

Nunca se nombró el vector. Eso es la **programación tácita** —o *point-free*—: **se compone la función, no se describe el cálculo sobre valores**.

Y la comparación con Haskell es directa:

```
media = (/) <$> sum <*> (fromIntegral . length)   -- lo mismo, y más largo
```

J tiene una gramática para esto: los **trenes** de dos y tres verbos —*hook* y *fork*— tienen reglas de composición definidas en el lenguaje, no son una biblioteca.

Y el segundo concepto es el rango, que es más regular que en APL:

```
   suma =: +/
   suma"1 tabla       NB. aplicar a cada FILA (rango 1)
   suma"2 tabla        NB. aplicar a cada MATRIZ (rango 2)
```

`"n` **declara sobre qué nivel de la estructura actúa el verbo.** Es una forma explícita y uniforme de lo que NumPy llama `axis=` y R resuelve con la familia `apply` — y en J es parte de la gramática.

> ***Y el compromiso es el de la clase 154: el código tácito es extraordinariamente conciso y difícil de leer para quien no está entrenado.*** *J es probablemente el lenguaje de este Atlas con la curva más pronunciada, y quien la sube dice —de forma bastante unánime— que le cambió la manera de pensar en datos.*

🔄 Estado actual

- **Libre desde 2011** (GPL), con versiones regulares; **J9** es la actual.
- **JQt y Jupyter**: entornos gráficos y cuadernos, que hacen la exploración mucho más accesible.
- `jd`: una base de datos columnar sobre J, en la línea de kdb+.
- **Y el linaje continúa**: **BQN** (Marshall Lochbaum, 2020) toma de J la regularidad del rango y vuelve a los símbolos; **Uiua** (2023) combina arreglos con una pila.

⚙️ Cómo se ejecuta hoy

```
jconsole main.ijs < entrada.txt       # consola de J
ijconsole -js "echo +/ 1 2 3"          # una expresión suelta
# Y jqt para el entorno gráfico, o el núcleo de Jupyter
```

🍼 El programa de la clase 041 en J

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
NB. Leer la línea, convertirla a números y calcular el total
v =: ".  1!:1  3           NB. 1!:1[3 lee la entrada estándar; ". la convierte
total =: (*/ 2 {. v) * 1 - {: v

NB. Y la versión tácita, sin nombrar el vector:
total2 =: 3 : '(*/ 2 {. y) * 1 - {: y'

echo 'Total: ' , (8j2 ": total)
```

Lo que hay que ver.

- `*/ 2 {. v` es exactamente lo mismo que `×/ 2↑v` en [APL: **toma los dos primeros y redúcelos con el producto**. La correspondencia símbolo a símbolo entre los dos lenguajes es directa, y comparar las dos fichas enseña la traducción.
- `{: v` **es el último elemento**, donde APL usaba `⊃⌽v`.
- `8j2 ": total` formatea con 8 posiciones y 2 decimales: `":` es el verbo de formato, y `8j2` es su argumento izquierdo. **Es `%.2f` con otra gramática.**
- `1!:1` **es una llamada al sistema** por número — la interfaz de entrada y salida de J, que es deliberadamente austera.
- **Y `3 : '...'` define un verbo explícito**, con `y` como argumento derecho. **La existencia de dos estilos —tácito y explícito— es la seña de identidad del lenguaje**, y la elección entre ellos es la decisión de legibilidad de la clase 146.

📚 Fuentes y bibliografía

- jsoftware.com (https://www.jsoftware.com/) — descarga, documentación y el *J Primer*.
- Learning J (https://www.jsoftware.com/help/learning/contents.htm) — **Roger Stokes**; libre y el camino recomendado.
- J Wiki (https://code.jsoftware.com/wiki/Main_Page) — ensayos, recetas y la explicación de los trenes.
- **Kenneth Iverson**, *J Introduction and Dictionary* — el diccionario del lenguaje, que es literalmente un diccionario: cada símbolo con su definición.
- BQN (https://mlochbaum.github.io/BQN/) — para ver hacia dónde ha ido el diseño de lenguajes de arreglos después de J.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: APL · R · Haskell · Julia · MATLAB

🧉 Java — 1995

Java es el lenguaje que llevó la máquina virtual y la recolección de basura al software empresarial del mundo entero. Treinta años después sigue moviendo los sistemas de banca, seguros y comercio de media Europa — y, después de una década de fama de lento y verboso, **ha cambiado más en los últimos seis años que en los veinte anteriores**.

> 🎯 ***Por qué está en este programa***
>
> ***Java es uno de los diez lenguajes del núcleo** y el **representante de la familia JVM** (Atlas): quien entiende la versión Java de una clase reconoce después Kotlin, Scala, Groovy y —con más esfuerzo — Clojure, porque **los cuatro compilan al mismo bytecode y comparten biblioteca**.*
>
> *Y aporta al programa los conceptos de **máquina virtual con JIT** (clases 125 y 126), **recolección de basura generacional** (clase 131) y **orientación a objetos nominal con interfaces** (clase 112), que son los que definieron la práctica de la industria durante veinte años.*

Año	1995; **5** con genéricos (2004); **8** con lambdas (2014); **cada 6 meses** desde 2017
Autoría	**James Gosling** y equipo, Sun Microsystems (hoy Oracle)
Familia	JVM; sintaxis de C, semántica influida por Smalltalk y Objective-C
Paradigma	Orientado a objetos con clases; funcional desde Java 8
Tipado	**Estático, nominal y fuerte**; con genéricos por borrado
Memoria	**Recolección de basura**; varios recolectores intercambiables
Ejecución	Bytecode sobre la JVM, con **JIT** de dos niveles; también AOT con GraalVM
Estado	🟢 **Dominante** en sistemas empresariales y en el ecosistema de datos

---

📜 Historia

En **1991**, Sun montó el proyecto *Green* para hacer software de electrodomésticos: un mando a distancia inteligente. **James Gosling** diseñó un lenguaje llamado **Oak** —por un roble que veía desde su ventana— con dos requisitos que resultaron proféticos: **portabilidad entre chips distintos** y **seguridad**, porque el código iba a viajar.

El mercado de la televisión interactiva no llegó, pero en **1995** llegó la web, y Sun reposicionó el lenguaje como **Java**, con el eslogan **"write once, run anywhere"** y los *applets* en el navegador. Los applets murieron; **la máquina virtual sobrevivió y ganó**.

Los hitos que lo formaron:

- **Java 2 / J2EE (1999)**: el servidor de aplicaciones y toda la arquitectura empresarial.
- **Java 5 (2004)**: **genéricos** —implementados por *borrado* para no romper la compatibilidad binaria, una decisión que se discute desde entonces (clase 143)—, anotaciones, `enum`, `for-each`.
- **Java 7 (2011)**: `try-with-resources` (clase 132), NIO.2.
- **Java 8 (2014)**: **lambdas y** `Stream` — el cambio más grande de su historia, que llevó lo funcional al lenguaje empresarial (clase 115).
- **2017**: cambio a **una versión cada seis meses**, con versiones de soporte largo cada dos años (11, 17, 21, 25).
- **Java 21 (2023)**: **hilos virtuales** (Project Loom), emparejamiento de patrones, clases selladas.

Y por el camino, el **juicio Oracle contra Google** sobre las APIs de Java —resuelto en 2021 a favor del uso legítimo— definió jurisprudencia sobre si una interfaz de programación se puede copiar.

🏭 Dónde vive hoy

- **Banca, seguros y administración**: el lenguaje al que se migra el COBOL cuando se migra (clase 175).
- **Android**: durante quince años el lenguaje oficial; hoy comparte sitio con Kotlin.
- **Datos a gran escala**: Hadoop, Spark, Kafka, Flink, Elasticsearch, Cassandra — el ecosistema de datos está escrito en Java y en Scala.
- **Servidores de aplicaciones**: Spring y Jakarta EE mueven una parte enorme del software corporativo.
- **Herramientas**: Maven, Gradle, IntelliJ, Jenkins.

🧠 Lo que enseña: la máquina virtual como decisión de arquitectura

Java es el sitio donde mejor se ve el compromiso central de las clases 125 y 126:

```
Compilar a bytecode y ejecutarlo en una máquina virtual con JIT da:
  + portabilidad real entre sistemas y arquitecturas
  + optimización con información de EJECUCIÓN, que un compilador estático no tiene
  + recolección de basura, seguridad de memoria y verificación del bytecode
  - arranque más lento y consumo de memoria mayor
  - y pausas del recolector, que en latencia baja importan (clase 152)
```

El JIT con información de ejecución merece el detalle, porque es contraintuitivo: **la JVM puede generar código más rápido que un compilador estático** en ciertos casos, porque sabe **qué rama se toma siempre** y **qué tipo llega de verdad a un método polimórfico**, y compila especializando —con una comprobación de guarda por si cambia—.

Es exactamente la **caché de envío en línea** que se inventó en Smalltalk y en Self (clase 152), y es la razón por la que el despacho dinámico dejó de ser caro.

Y el otro concepto que Java llevó a la industria es el de la clase 112: **la interfaz como contrato sin implementación**, con herencia simple de clases y múltiple de interfaces — la respuesta al problema del diamante que C++ tiene abierto.

🔄 Lo que se ha modernizado

- **Hilos virtuales (Java 21)**: millones de hilos ligeros gestionados por la JVM, con **código secuencial** en lugar de asíncrono (clases 133 y 135). Es la respuesta de Java al modelo de Go y de Erlang, y sin cambiar la forma de escribir el código.
- **Registros, clases selladas y emparejamiento de patrones**: tipos de datos algebraicos, la idea de Haskell y OCaml llegando al lenguaje empresarial (clase 100).
- `var` para inferencia local, y ficheros de una sola clase ejecutables sin `main` ceremonioso.
- **GraalVM con imagen nativa**: compilación anticipada a binario, con **arranque de milisegundos** — lo que hace competitivo a Java en funciones sin servidor y en contenedores (clase 174).
- **Recolectores modernos**: **ZGC** y **Shenandoah**, con pausas por debajo del milisegundo incluso con montones de terabytes.
- **Panama**: interoperabilidad con C **sin JNI** (clase 156), con memoria fuera del montón segura.

⚙️ Cómo se ejecuta hoy

```
javac Main.java && java Main < entrada.txt     # el camino de la clase 041
java Main.java                                  # desde Java 11: sin compilar antes

mvn verify        # o: gradle build             (clase 143)
java -jar app.jar
native-image -jar app.jar                       # GraalVM: binario nativo (clase 174)
```

🍼 El programa de la clase 041 en Java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Locale;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] p = br.readLine().trim().split("\\s+");

        // Tipado estático nominal: cada valor declara su tipo.
        final double precioUnitario = Double.parseDouble(p[0]);
        final int cantidad = Integer.parseInt(p[1]);
        final double descuento = Double.parseDouble(p[2]);

        double subtotal = precioUnitario * cantidad;
        double total = subtotal * (1 - descuento);

        System.out.printf(Locale.US, "Total: %.2f%n", total);
    }
}
```

Lo que hay que ver, comparando con las otras fichas.

- **La ceremonia es la seña de identidad**: clase, método estático, tipos declarados y excepción propagada. Kotlin y Groovy hacen lo mismo en tres líneas sobre la misma máquina virtual, y esa comparación es exactamente el contenido del Atlas.
- `final` **sí es una constante**, a diferencia de la convención de mayúsculas de Python (clase 041) — aunque `final` sobre una referencia solo fija **el nombre**, no el objeto (clase 102).
- `Locale.US` **no es decorativo**: sin él, en una máquina con configuración española, `%.2f` imprimiría `27000,00` con coma. Es la misma trampa que `CultureInfo.InvariantCulture` en C# y VB.NET, y una de las causas más frecuentes de que una prueba pase en local y falle en el servidor (clase 147).
- `int` **y** `double` **son primitivos**, no objetos: Java tiene dos mundos de tipos, y el autoboxing entre ellos cuesta memoria y tiempo (clase 128). Es una de las cosas que Project Valhalla está en camino de resolver.

📚 Fuentes y bibliografía

- Documentación de Java (Oracle) (https://docs.oracle.com/en/java/javase/) y las JEP (https://openjdk.org/jeps/0) — cada característica nueva, con su motivación.
- Especificación del lenguaje y de la JVM (https://docs.oracle.com/javase/specs/) — para la clase 125.
- **Joshua Bloch**, *Effective Java*, 3.ª ed., Addison-Wesley — el libro de referencia; 90 elementos que explican no solo qué hacer sino por qué.
- **Brian Goetz et al.**, *Java Concurrency in Practice* — anterior a Loom y todavía imprescindible para entender el modelo de memoria (clase 136).
- **Ben Evans, Jason Clark, Martijn Verburg**, *The Well-Grounded Java Developer*, 2.ª ed., Manning — Java moderno, con la JVM por dentro.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Kotlin · Scala · Groovy · Clojure · C# · Smalltalk

🟧 JavaScript — 1995

JavaScript se diseñó **en diez días**, se le puso el nombre de otro lenguaje por marketing, y hoy es el único que ejecutan todos los navegadores del mundo. Es el caso más claro de que **la suerte de una tecnología no depende solo de su calidad** — y también de que un lenguaje puede mejorar muchísimo sin poder romper nada.

> 🎯 ***Por qué está en este programa***
>
> ***JavaScript es uno de los diez lenguajes del núcleo**, y es el **representante de la familia web** (Atlas) junto con TypeScript.*
>
> *Aporta al programa tres conceptos que ningún otro del núcleo enseña igual: **la herencia por prototipos** en vez de por clases (clase 112), **el bucle de eventos con un solo hilo** (clase 134) y **la coerción débil de tipos**, que es el mejor ejemplo posible de por qué la igualdad es un tema (clase 100).*

Año	1995; **ES5** en 2009; **ES6/ES2015**, el punto de inflexión; anual desde entonces
Autoría	**Brendan Eich**, Netscape — el prototipo, en diez días
Familia	JavaScript / web; sintaxis de C, semántica de Scheme y Self
Paradigma	Multiparadigma: funcional, orientado a objetos por prototipos, imperativo
Tipado	**Dinámico y débil**: las conversiones implícitas son parte del diseño
Memoria	Automática, con recolector generacional
Ejecución	JIT (V8, SpiderMonkey, JavaScriptCore); también AOT en algunos entornos
Estado	🟢 **Ubicuo** — navegador, servidor, escritorio, móvil y sistemas embebidos

---

📜 Historia

En **1995**, Netscape quería un lenguaje de guion para su navegador. Contrató a **Brendan Eich** con el encargo de que **se pareciera a Java** —que era lo que estaba de moda— y le dio **diez días** para el prototipo.

Eich quería hacer un Scheme para el navegador. El resultado fue un compromiso que explica casi todas las rarezas del lenguaje: **funciones de primera clase y cierres de Scheme, herencia por prototipos de Self, y sintaxis de llaves de Java**. Se llamó primero Mocha, luego LiveScript, y finalmente **JavaScript** por un acuerdo comercial con Sun — un nombre que ha causado confusión durante treinta años, porque **Java y JavaScript no tienen relación**.

Microsoft respondió con **JScript**, y de ahí nació la estandarización en **ECMA** — de donde viene el nombre feo pero correcto: **ECMAScript**.

Los años de **ES3 a ES5 (1999-2009)** fueron de estancamiento: ES4 se abandonó tras una guerra política, y el lenguaje se quedó congelado mientras la web crecía. Lo que sí ocurrió en esa década fue **jQuery** —que tapó las diferencias entre navegadores— y **V8** (2008), el motor de Chrome con JIT que **multiplicó por cien el rendimiento** y cambió lo que se podía hacer con el lenguaje.

Node.js (2009) sacó JavaScript del navegador usando V8, y **ES2015 (ES6)** lo modernizó de golpe: `let` / `const`, clases, módulos, promesas, funciones flecha, desestructuración, plantillas de cadena. Desde entonces hay **una versión anual**, y el lenguaje ha crecido con `async` / `await`, opcional encadenado, `BigInt` y

el emparejamiento de patrones en camino.

🏭 Dónde vive hoy

- **Todos los navegadores**: es el único lenguaje que ejecutan de forma nativa, y por eso todo lo demás compila a él o a WebAssembly.
- **Servidores**: Node.js, Deno y Bun mueven una parte enorme de la web.
- **Escritorio**: Electron — VS Code, Slack, Discord, Figma.
- **Móvil**: React Native.
- **Herramientas de construcción** de casi todos los ecosistemas front-end.
- **Sistemas embebidos y bases de datos**: motores ligeros como QuickJS y Duktape se incrustan en routers, televisores y bases de datos como lenguaje de extensión (clase 163).

🧠 Lo que enseña y nadie más enseña igual

Uno, los prototipos (clase 112). En JavaScript **no hay clases de verdad**: `class` es azúcar sobre un mecanismo donde **cada objeto tiene un enlace a otro objeto**, y la búsqueda de una propiedad sube por esa cadena. Es más simple y más flexible que las clases, y es el modelo que Self inventó.

Dos, el bucle de eventos (clases 134 y 161). JavaScript tiene **un solo hilo**, y toda la concurrencia es asíncrona: la operación se registra y se sigue; cuando termina, el bucle ejecuta la retrollamada. Es el mismo modelo que Tcl tenía en 1993 con `fileevent`, y es lo que permite atender decenas de miles de conexiones sin un hilo por cada una.

Y tres, la coerción débil, que es el ejemplo didáctico perfecto de la clase 100:

```
0 == "0"          // true
0 == []           // true
"0" == []         // false   ← la igualdad NO es transitiva
[] + {}           // "[object Object]"
0.1 + 0.2         // 0.30000000000000004   (clase 073: esto pasa en TODOS)
```

> ***Y aquí conviene ser justo.*** *La última línea* ***no es un defecto de JavaScript****: es IEEE 754, y ocurre igual en Python, Java y C (clase 073). Las anteriores sí lo son, y la respuesta del ecosistema fue clara:* ***usar siempre*** `===`*, que no convierte. Es exactamente el patrón de la clase 146 —* ***una regla de estilo que corrige una decisión del lenguaje*** *y que los analizadores hacen cumplir.*

🔄 Lo que se ha modernizado

- **De** `var` **a** `let` / `const`, que arregla el alcance por función y el *hoisting* (clase 087).
- **Módulos ESM** en el estándar, tras años de CommonJS y de empaquetadores.
- `async` / `await` sobre promesas: código asíncrono con aspecto secuencial (clase 134).
- `BigInt` para enteros de precisión arbitraria, y `Temporal` para fechas —la corrección de la peor API de la biblioteca estándar—.
- **Ejecutores nuevos**: **Deno** (seguro por capacidades desde el arranque, clase 153) y **Bun** (rapidez y herramientas integradas).
- **Aislamiento real**: los *realms* y los objetos endurecidos, herederos directos de la investigación en capacidades de Smalltalk y del lenguaje E (clase 153).

⚙️ Cómo se ejecuta hoy

```
node main.mjs < entrada.txt        # el comando de la clase 041
deno run --allow-read main.mjs      # con permisos EXPLÍCITOS
bun run main.mjs

npx eslint . && npx prettier --check .    # calidad (clase 146)
node --test                                # pruebas, en el propio Node
```

🍼 El programa de la clase 041 en JavaScript

```
import { readFileSync } from "node:fs";

// Números en JS: un solo tipo `number` (doble de 64 bits) para todo.
const [precio, cantidad, descuento] = readFileSync(0, "utf8")
  .trim()
  .split(/\s+/)
  .map(Number);

const subtotal = precio * cantidad;
const total = subtotal * (1 - descuento);

console.log(`Total: ${total.toFixed(2)}`);
```

Lo que hay que ver.

- **Hay un solo tipo numérico.** No existe `int`: `cantidad` es un **doble de 64 bits**, igual que el precio. Eso simplifica el lenguaje y trae la consecuencia de la clase 045: **los enteros por encima de 2^{53} pierden precisión**, que es la razón de que las APIs envíen los identificadores grandes como cadenas (clase 159).
- `const` **no significa constante**: significa que **el nombre no se reasigna**. Un objeto declarado con `const` sigue siendo mutable — una distinción que la clase 102 desarrolla.
- **La cadena de** `.trim().split().map()` es el estilo funcional del lenguaje, heredado de Scheme y popularizado por las bibliotecas.
- `readFileSync(0, ...)` lee el descriptor 0, que es la entrada estándar: Node no tiene una lectura de línea síncrona sencilla, y este es el idioma habitual.

📚 Fuentes y bibliografía

- MDN Web Docs (https://developer.mozilla.org/es/docs/Web/JavaScript) — la referencia, y está en español.
- Especificación ECMA-262 (https://tc39.es/ecma262/) y las propuestas de TC39 (https://github.com/tc39/proposals) — para saber qué viene y en qué etapa está.
- **Kyle Simpson**, *You Don't Know JS Yet*, 2.ª ed. — libre en línea; la mejor explicación de cierres, `this` y prototipos.
- **Axel Rauschmayer**, *JavaScript for impatient programmers* — libre en línea, actualizado cada año.
- **Douglas Crockford**, *JavaScript: The Good Parts* — histórico y discutible hoy, pero explica por qué el ecosistema adoptó `===` y el modo estricto.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: TypeScript · Dart · Elm · ActionScript · Lua

📋 JCL — década de 1960

No es un lenguaje de programación, y por eso hay que estudiarlo. JCL no calcula nada: describe **qué programa ejecutar, con qué datos y en qué orden**. Todo el proceso nocturno por lotes de la banca mundial —el momento en que se liquidan las operaciones del día— está descrito en JCL.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: JCL se ejecuta cada noche en cada mainframe z/OS del planeta.*** *Es la pieza sin la cual el COBOL y el PL/I de producción no arrancan. Quien trabaje profesionalmente con mainframe escribe JCL a diario, aunque no escriba COBOL.*
>
> *Y entra, sobre todo, porque* ***enseña una separación que los lenguajes del núcleo tienen borrosa****: la que hay entre* ***el programa*** *y* ***su entorno de ejecución****. Un programa COBOL no sabe dónde están sus ficheros: se refiere a nombres lógicos (ddnames), y es el JCL quien los conecta a ficheros reales* ***en el momento de ejecutar****. Eso es inyección de dependencias a nivel de sistema operativo, en 1964. La misma idea que hoy llamamos "configuración por entorno", "volúmenes de Docker" o "los doce factores" estaba resuelta —de otra manera y con otro vocabulario— antes de que existiera Unix. Y su* `COND` *sobre códigos de retorno es el antepasado directo de cualquier pipeline de CI que decide si el siguiente paso se ejecuta.*

Año	1964, con el **IBM System/360** y el sistema operativo OS/360
Autoría	**IBM**
Familia	Lenguajes de control de trabajos (*job control*)
Paradigma	**Declarativo**: describe recursos y secuencia, no algoritmos
Tipado	No aplica — describe recursos, no valores
Memoria	No aplica; sí gestiona **asignación de espacio en disco**
Ejecución	Interpretado por el **subsistema de entrada de trabajos** (JES2/JES3) de z/OS
Estado	🟢 **Imprescindible en el mundo mainframe** — banca, seguros, gobierno, retail

---

📜 Historia

Cuando IBM lanzó el **System/360** en 1964, resolvió un problema que hoy ni nos planteamos: **cómo decirle a un ordenador compartido qué hacer a continuación**. No había línea de comandos, ni sesión interactiva, ni terminal. Se entregaba una bandeja de tarjetas perforadas al operador, y esa bandeja tenía que explicarse a sí misma: qué programa cargar, cuánta memoria necesitaba, en qué unidad de cinta estaba la entrada, dónde dejar la salida, cuánto podía tardar y qué hacer si fallaba.

JCL es ese lenguaje de descripción. Sus rarezas se explican todas por la tarjeta perforada:

- Cada sentencia empieza por `//` en las **columnas 1 y 2**, porque así se distinguía de una tarjeta de datos.
- El campo de nombre va en la columna 3, la operación después, y **las columnas 73 a 80 se ignoran** porque ahí iba el número de secuencia de la tarjeta (para reordenar la bandeja si se caía al suelo).
- La continuación de línea se marca con una coma final y el `//` de la siguiente.

Sesenta años después, esas columnas siguen siendo obligatorias.

El modelo conceptual es de tres niveles, y no ha cambiado:

- **JOB** — la unidad de trabajo: quién lo envía, con qué prioridad, con qué límites de recursos.
- **EXEC** — un **paso**: qué programa (o qué procedimiento) ejecutar. Un trabajo tiene de uno a cientos de pasos.

- **DD** (*Data Definition*) — la conexión entre **un nombre lógico que el programa conoce** y **un fichero real del sistema**.

🏭 Dónde sobrevive hoy

- **El proceso nocturno por lotes** de bancos, aseguradoras, procesadores de tarjetas y organismos públicos: la "ventana batch" en la que se consolidan las operaciones del día, se calculan intereses, se generan extractos y se cuadran los saldos.
- **Compilación y despliegue** en z/OS: compilar un COBOL, enlazarlo y copiarlo a la librería de carga se hace enviando un trabajo JCL.
- **Operación y explotación**: cargas masivas, copias de seguridad, reorganización de bases de datos, intercambio de ficheros entre entidades.
- **Planificación**: herramientas como IBM Workload Scheduler o Control-M orquestan miles de trabajos JCL con dependencias entre ellos. La "malla batch" de una entidad grande son decenas de miles de ejecuciones diarias.

🧠 Por qué no ha muerto

1. Porque el proceso por lotes no ha muerto. Hay trabajo que solo tiene sentido hacer en bloque a una hora fija: cerrar un día contable, calcular intereses sobre todas las cuentas, compensar entre bancos. Eso no es un microservicio, es un lote, y necesita alguien que lo describa.

2. El desacoplamiento programa/datos es real y sigue siendo útil. El programa dice `SELECT CLIENTES ASSIGN TO ENTRADA`; el JCL dice `//ENTRADA DD DSN=PROD.CLIENTES.2026,DISP=SHR`. Cambiar de fichero de prueba a fichero de producción no toca ni una línea del programa ni requiere recompilar. Es exactamente el problema que resuelven hoy las variables de entorno y los volúmenes montados.

3. La gestión de recursos es explícita y auditable. El JCL declara cuánto espacio pedir, en qué volumen, con qué disposición si el trabajo falla, y qué hacer con el fichero después. En un sistema compartido por miles de trabajos con SLA, esa explicitud es una característica, no una molestia.

4. Los códigos de retorno gobiernan el flujo. `COND` —y su forma moderna `IF/THEN/ELSE`— decide si un paso se ejecuta según cómo terminaron los anteriores. Es el mismo modelo mental que `&&` en un shell o `needs:` en GitHub Actions.

5. Décadas de mallas de trabajos con dependencias. Rehacerlo implica reconstruir el mapa completo de qué depende de qué, y ese mapa muchas veces solo existe en la propia malla.

🔄 Lo que se ha modernizado

- `IF/THEN/ELSE/ENDIF` sustituyó al críptico `COND=(4,LT,PASO)`, cuya lógica invertida —se ejecuta el paso si la condición es **falsa**— ha confundido a generaciones enteras.
- **JCL simbólico y** `SET`: variables y sustitución para escribir procedimientos parametrizados y no copiar el mismo trabajo veinte veces.
- **Zowe**: el marco de código abierto (bajo la Open Mainframe Project) que expone z/OS mediante **API REST**, una **CLI** y extensiones de **VS Code**. Hoy se puede enviar un trabajo JCL y leer su salida desde un terminal de Linux o desde un pipeline de GitHub Actions. Eso ha cambiado de verdad la forma de trabajar.
- **DevOps sobre z/OS**: IBM Dependency Based Build y Git para los fuentes, con pipelines que construyen y despliegan en el mainframe igual que en cualquier otra plataforma.
- **JSON, REST y contenedores en z/OS**: z/OS Connect expone programas por lotes y transaccionales como APIs, y z/OS Container Extensions ejecuta contenedores Linux en la misma máquina.

⚙️ Cómo se ejecuta hoy

```
# Desde ISPF/TSO en el propio mainframe: se edita el JCL y se teclea
SUBMIT

# Desde fuera, con Zowe CLI — así es como se hace hoy desde un portátil:
zowe jobs submit local-file "./totvta.jcl" --view-all-spool-content
zowe jobs list jobs --owner VLAD
```

Para aprender sin acceso a un mainframe: IBM Z Xplore ofrece un entorno gratuito con ejercicios guiados; el emulador **Hercules** con **MVS 3.8j** (de dominio público) permite montar un mainframe de los 70 en un portátil, con su JCL auténtico.

🍼 El programa de la clase 041… en JCL

> ⚠️ ***Aquí el contrato cambia de naturaleza, y es justo el punto.*** *JCL* ***no puede*** *calcular el total de una venta: no tiene variables, ni aritmética, ni expresiones. Lo que hace es* ***compilar el programa COBOL de la clase 041, ejecutarlo, darle la entrada y recoger la salida****. Fingir un JCL que multiplica números sería inventar un lenguaje que no existe.* ***No se verifica en CI****: requiere z/OS.*

```
//TOTVTA   JOB (CONTAB),'TOTAL VENTA',CLASS=A,MSGCLASS=X,
//             NOTIFY=&SYSUID,REGION=0M
//*
//* ---------------------------------------------------------------
//* PASO 1 - COMPILAR Y ENLAZAR EL COBOL DE LA CLASE 041
//* ---------------------------------------------------------------
//COMPILA  EXEC IGYWCL
//COBOL.SYSIN    DD DSN=VLAD.FUENTE.COBOL(TOTVTA),DISP=SHR
//LKED.SYSLMOD   DD DSN=VLAD.LOADLIB(TOTVTA),DISP=SHR
//*
//* ---------------------------------------------------------------
//* PASO 2 - EJECUTARLO, SOLO SI EL PASO ANTERIOR FUE LIMPIO
//* ---------------------------------------------------------------
//EJECUTA  EXEC PGM=TOTVTA
//STEPLIB  DD DSN=VLAD.LOADLIB,DISP=SHR
//SYSOUT   DD SYSOUT=*
//SYSIN    DD *
15000 2 0.10
/*
//
```

Y con la sintaxis condicional moderna, que es como se escribe hoy:

```
//  IF (COMPILA.LKED.RC <= 4) THEN
//EJECUTA  EXEC PGM=TOTVTA
//STEPLIB  DD DSN=VLAD.LOADLIB,DISP=SHR
//SYSOUT   DD SYSOUT=*
//SYSIN    DD *
15000 2 0.10
/*
//  ENDIF
```

Recorrido, línea a línea.

- `//TOTVTA JOB ...` — la sentencia **JOB**. `TOTVTA` es el nombre del trabajo (máximo 8 caracteres), `(CONTAB)` es la información de contabilidad —a quién se le imputa el consumo de CPU, porque en un mainframe **los recursos se facturan por departamento**—, `CLASS` define la cola de ejecución, `MSGCLASS` dónde va el

registro, `NOTIFY=&SYSUID` avisa al usuario al terminar y `REGION=0M` pide memoria sin límite artificial.

- La **coma final** de la primera línea indica continuación; la siguiente empieza también por `//` y deja espacios antes del parámetro.
- `//*` es un comentario.
- `//COMPILA EXEC IGYWCL` — un **paso** llamado `COMPILA`. `IGYWCL` no es un programa: es un **procedimiento catalogado** de IBM que ya contiene los pasos de compilar (`COBOL`) y enlazar (`LKED`). Es reutilización de JCL: el equivalente a una *action* reutilizable de un pipeline.
- `//COBOL.SYSIN DD ...` — el punto en `COBOL.SYSIN` significa "la sentencia DD llamada `SYSIN` **dentro del paso** `COBOL` del procedimiento". Así se parametriza un procedimiento desde fuera: se sobrescriben sus DD.
- `DSN=VLAD.FUENTE.COBOL(TOTVTA)` — el nombre del *dataset*. Los conjuntos de datos de z/OS no forman un árbol de directorios: son **nombres jerárquicos de hasta 44 caracteres** en calificadores separados por puntos. Los paréntesis indican un **miembro** dentro de un dataset particionado (PDS), que es lo más parecido a una carpeta.
- `DISP=SHR` es la **disposición**: puede compartirse con otros trabajos. Las otras opciones — `OLD` (uso exclusivo), `NEW` (créalo), `MOD` (añade al final)— y la disposición **al terminar** (`CATLG`, `DELETE`, `KEEP`) son la mitad del oficio: describen qué pasa con el fichero si el trabajo va bien y si va mal.
- `//STEPLIB DD ...` indica dónde buscar el programa ejecutable. Es, literalmente, el `PATH` de ese paso.
- `//SYSOUT DD SYSOUT=*` envía la salida al *spool* del sistema, de donde se recoge después.
- **`//SYSIN DD *` es la clave de la conexión con la clase.** El asterisco significa "los datos vienen aquí mismo, en las líneas siguientes, hasta el `/*`". Ese es el `stdin` del programa: exactamente el `15000 2 0.10` del `casos.json` de la clase 041. El programa COBOL lo lee con `ACCEPT` sin saber que venía incrustado en el JCL, y mañana el mismo programa puede leerlo de un fichero de un millón de registros cambiando solo esta línea.
- `//` a solas cierra el trabajo.

La lección, en una frase: el programa declara *qué nombres lógicos usa*; el JCL decide *a qué apuntan de verdad*, en el momento de ejecutar. Cuando en la Parte 9 se hable de configuración externa y de entornos, este es el ejemplo más antiguo y más literal que existe.

🔍 Qué reconocer si vienes de otro mundo

Si conoces…	En JCL es…
Un script de shell	El **JOB** completo
`command args`	`//PASO EXEC PGM=PROGRAMA,PARM='...'`
`< entrada.txt`	`//SYSIN DD DSN=...` o `//SYSIN DD *` con datos en línea
`> salida.txt`	`//SALIDA DD DSN=...,DISP=(NEW,CATLG)`
`$PATH`	`//STEPLIB DD DSN=...`
Variable de entorno	Parámetro simbólico (`&VAR`) y `// SET VAR=valor`
`cmd1 && cmd2`	`COND=` o `IF (PASO.RC = 0) THEN`
`needs:` de GitHub Actions	La secuencia de pasos con sus condiciones de código de retorno
Función reutilizable	**Procedimiento catalogado** (`PROC` … `PEND`)
Volumen de Docker	La sentencia **DD**: monta un recurso externo con un nombre lógico
`docker run --memory`	`REGION=`

⚠️ Errores comunes al leerlo

- **La lógica invertida de `COND`.** `COND=(4,LT,PASO1)` significa "**omite** este paso si 4 es menor que el código de retorno de PASO1". Se lee al revés de lo que la intuición sugiere y ha causado más incidentes que ninguna otra construcción del lenguaje. Por eso existe `IF/THEN`.
- **Ignorar las columnas.** `//` en 1–2 sin excepción, y todo lo escrito a partir de la columna 73 se descarta en silencio. Un parámetro que "no se aplica" suele ser un parámetro que cruzó la columna 72.
- **Confundir un espacio con un separador cualquiera.** El espacio tras el campo de operandos **termina** los operandos; lo que venga después es comentario. Un espacio de más dentro de los parámetros corta la sentencia.
- **Creer que un dataset es un fichero de Unix.** Tiene formato de registro (`RECFM`), longitud de registro (`LRECL`) y tamaño de bloque declarados. Un `LRECL` equivocado no da un error bonito: da datos ilegibles.
- **Suponer que existe aritmética.** No la hay. Ni variables de usuario, ni bucles, ni expresiones. Si algo hay que calcular, lo calcula un programa y JCL decide qué hacer con su código de retorno.
- **Olvidar la disposición de fallo.** `DISP=(NEW,CATLG,DELETE)` significa: nuevo, catalógalo si va bien, **bórralo si falla**. Escribir `CATLG` en el tercer campo deja basura tras cada error y, en la siguiente ejecución, el dataset ya existe y el trabajo falla por otra razón.

📚 Fuentes y bibliografía

- z/OS MVS JCL Reference (IBM) (https://www.ibm.com/docs/en/zos/3.1.0?topic=mvs-zos-jcl-reference) — la referencia completa de cada sentencia y parámetro.
- z/OS MVS JCL User's Guide (IBM) (https://www.ibm.com/docs/en/zos/3.1.0?topic=mvs-zos-jcl-users-guide) — la guía con el porqué y los ejemplos.
- IBM Z Xplore (https://www.ibmzxplore.com/) — entorno gratuito con ejercicios guiados sobre un z/OS real; la forma práctica de tocar JCL sin tener un mainframe.
- Zowe (https://www.zowe.org/) — la CLI, las APIs REST y la extensión de VS Code que conectan z/OS con herramientas modernas.
- **Gary DeWard Brown**, *z/OS JCL*, 6.ª ed., Wiley — el libro de referencia sobre JCL desde hace décadas.
- **Mike Murach, Raul Menendez, Doug Lowe**, *Murach's OS/390 and z/OS JCL* — el manual didáctico, con el enfoque práctico del programador de aplicaciones.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: COBOL · PL/I · RPG

🔬 Julia — 2012

Julia nació para resolver **el problema de los dos lenguajes**: escribir el prototipo en Python o MATLAB y luego reescribir lo lento en C o Fortran. Su propuesta es que **un lenguaje puede ser cómodo y rápido a la vez**, y lo consigue con una idea poco frecuente: **despacho múltiple compilado especializando por tipos en ejecución**.

> 🎯 ***Por qué está en este programa***
>
> *Julia es un **primo de la familia array / científica** (Atlas), junto a APL, J, R, MATLAB y Fortran.*
>
> *Aporta al programa **el despacho múltiple como paradigma central** (clase 111) — la idea que Common Lisp tenía con CLOS, aquí convertida en la forma normal de organizar el código. Y aporta el ejemplo más claro de **JIT especializante** (clase 126).*

Año	2012; **1.0** en 2018, con promesa de estabilidad; **1.11** actual
Autoría	**Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman** — MIT
Familia	Array / científica; con Lisp, Python, MATLAB y R dentro
Paradigma	Multiparadigma con **despacho múltiple**; funcional y por arreglos
Tipado	**Dinámico con tipos ricos**, usados para especializar en compilación
Memoria	Recolección de basura
Ejecución	**JIT sobre LLVM**, compilando una versión por combinación de tipos
Estado	🟢 **En crecimiento** en cálculo científico, optimización y simulación

---

📜 Historia

En **2012**, cuatro investigadores del MIT publicaron un manifiesto —*Why We Created Julia*— que enumeraba lo que querían, y que se lee como una lista de deseos imposible:

> *Queremos la velocidad de C, el dinamismo de Ruby, la homoiconicidad de Lisp con macros de verdad, la notación matemática de MATLAB, la potencia estadística de R, la facilidad de Python para guiones, la capacidad de Perl para procesar texto y la potencia lineal de Matlab.* ***Y que sea fácil de aprender.***

El problema concreto que atacaban tiene nombre en la comunidad científica: **el problema de los dos lenguajes**. Se prototipa en un lenguaje cómodo y, cuando hace falta rendimiento, **se reescriben las partes críticas en C o Fortran** (clase 155) — con el coste de mantener dos versiones y la frontera entre ellas.

La solución de Julia es el despacho múltiple con especialización en compilación, y es lo que hace que funcione: **el JIT compila una versión de cada función para cada combinación concreta de tipos con la que se llama**, así que el código genérico acaba siendo tan rápido como el escrito a mano para esos tipos.

Julia 1.0 (2018) estabilizó el lenguaje, y desde entonces la evolución se ha centrado en el arranque —su punto débil histórico— y en las herramientas.

🏭 Dónde vive hoy

- **Cálculo científico y simulación**: ecuaciones diferenciales (**DifferentialEquations.jl** es de las mejores bibliotecas que existen en cualquier lenguaje), mecánica, clima.
- **Optimización matemática**: **JuMP** es referencia en programación lineal y entera.
- **Aprendizaje automático científico**: **Flux**, **SciML**, y la diferenciación automática, que en Julia es especialmente natural.
- **Farmacometría y biología de sistemas**: Pumas es un caso comercial notable.
- **Y en investigación en general**, como alternativa a MATLAB sin licencia.

🧠 Lo que enseña: despacho múltiple

Es el concepto central y merece verlo con calma (clase 111):

```
area(c::Circulo) = π * c.r^2
area(r::Rectangulo) = r.a * r.b

# Y con DOS tipos a la vez:
interactuar(a::Asteroide, n::Nave) = "la nave esquiva"
interactuar(a::Asteroide, p::Planeta) = "impacto"
interactuar(n::Nave, p::Planeta) = "aterrizaje"
```

El método se elige según los tipos de TODOS los argumentos, no solo del primero. En un lenguaje con despacho simple —Java, C++, Python— eso obliga al patrón Visitante o al doble despacho (clase 151); aquí **es la forma normal de escribir**.

Y la consecuencia arquitectónica es grande, y explica el ecosistema de Julia:

```
Un paquete define el tipo `Cuaternion`.
OTRO paquete, sin conocerlo, define la función `graficar`.
Y un TERCERO puede escribir `graficar(::Cuaternion)` sin tocar ninguno de los dos.
```

Eso hace que las bibliotecas se combinen sin haberlo previsto —lo que la comunidad llama composabilidad— y es la razón de que el ecosistema científico de Julia sea tan interoperable.

Y el JIT especializante es la otra mitad:

```
f(x, y) = x * y + 1

f(2, 3)          # compila una versión para (Int64, Int64)
f(2.0, 3.0)       # compila OTRA para (Float64, Float64)
@code_native f(2, 3)     # ← se puede VER el código máquina generado
```

El mismo código genérico produce código máquina especializado, sin comprobaciones de tipo en el bucle interno. Eso es lo que iguala el rendimiento con C.

> ***Y el coste, que hay que decir*** *(clase 164):* ***el tiempo hasta el primer gráfico****. Como se compila al llamar,* ***la primera ejecución de cada función paga la compilación*** *— lo que hacía frustrante el arranque interactivo. Julia 1.9+ lo ha mejorado mucho con la caché de código nativo en los paquetes, y sigue siendo su punto más criticado.*

🔄 Lo que se ha modernizado

- **Precompilación de código nativo en paquetes** (1.9): reduce drásticamente el tiempo hasta el primer resultado.
- `PackageCompiler.jl` y las imágenes de sistema: binarios autocontenidos (clase 174).
- **Multihilo maduro** (`Threads.@threads` , tareas) y computación distribuida en la biblioteca estándar (clase 135).
- **Interoperabilidad excelente**: `ccall` con C **sin escribir envoltorios** (clase 156), y `PythonCall` , `RCall` , `MATLAB.jl` para llamar a los vecinos.
- **Y** `Pkg` , el gestor de paquetes, con entornos por proyecto y `Manifest.toml` como fichero de bloqueo (clase 143) — de lo mejor diseñado de esta lista.

⚙️ Cómo se ejecuta hoy

```
julia main.jl < entrada.txt          # el comando
julia --project=. -e 'using Pkg; Pkg.test()'    # pruebas con el entorno del proyecto

julia                                 # el REPL, que es donde se trabaja de verdad
# ] activate .    ] instantiate    ] test        ← el modo de paquetes
```

🍼 El programa de la clase 041 en Julia

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
v = parse.(Float64, split(readline()))
total = v[1] * v[2] * (1 - v[3])
println("Total: ", round(total; digits=2))
```

Lo que hay que ver.

- `parse.(Float64, ...)` **con el punto es la difusión (*broadcasting*)**: el punto **aplica la función a cada elemento**. Es la marca de la casa, y funciona con **cualquier** función y **cualquier** operador: `a .+ b`, `sin.(v)`, `f.(x, y)`. **Es** `map` **convertido en sintaxis uniforme** (clase 115), y elimina la mayoría de los bucles.
- **Los índices empiezan en 1**, como Fortran, R, MATLAB y APL — la convención de la familia científica.
- `v` **es un** `Vector{Float64}` **con tipo concreto**, y el compilador lo sabe: por eso la aritmética de la segunda línea genera el mismo código máquina que en C.
- `round(total; digits=2)` usa un argumento con nombre tras el punto y coma — otra decisión de legibilidad de la familia científica.
- **Y compárese con R**: los dos tratan la línea como un vector, pero Julia **conoce el tipo del vector y compila para él**, mientras que R interpreta.

📚 Fuentes y bibliografía

- Documentación de Julia (https://docs.julialang.org/) — el manual es muy bueno; el apartado *Performance Tips* es material directo de la clase 152.
- **Bezanson, Karpinski, Shah, Edelman**, *Why We Created Julia* (2012) y *Julia: A Fresh Approach to Numerical Computing* (SIAM Review, 2017) — el manifiesto y el artículo técnico.
- **Ben Lauwens, Allen Downey**, *Think Julia* — libre en línea; introducción desde cero.
- Julia Academy (https://juliaacademy.com/) y JuliaCon (https://juliacon.org/) — cursos y charlas.
- **Stefan Karpinski**, *The Unreasonable Effectiveness of Multiple Dispatch* (JuliaCon 2019) — la mejor explicación de por qué el despacho múltiple cambia la composición de bibliotecas (clase 151).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Python · R · MATLAB · Fortran · Common Lisp

🟠 Kotlin — 2011

Kotlin es lo que pasa cuando una empresa que hace herramientas de desarrollo —JetBrains— diseña un lenguaje: **cada decisión responde a un problema concreto que habían visto miles de veces en código Java real**. No inventa paradigmas; **quita fricción**, y esa modestia es la razón de su éxito.

> 🎯 ***Por qué está en este programa***
>
> *Kotlin es un* ***primo de la familia JVM*** *(Atlas), cuyo representante en el núcleo es Java.*

> *Aporta al programa la comparación más didáctica de todo el Atlas:* ***el mismo programa, en la misma máquina virtual, con la misma biblioteca — y la mitad de líneas****. Y aporta dos conceptos concretos:* ***la nulabilidad en el sistema de tipos*** *(clase 100) y* ***las corrutinas con concurrencia estructurada*** *(clase 134).*

Año	2011; **1.0** en 2016; oficial en Android desde 2017; **2.0** en 2024
Autoría	**JetBrains**, con Andrey Breslav al frente del diseño inicial
Familia	JVM; con influencia de Scala, C#, Groovy y ML
Paradigma	Multiparadigma: OO y funcional, con orientación a DSL
Tipado	**Estático, con inferencia** y **nulabilidad en el tipo**
Memoria	La de la JVM; en Native, con conteo de referencias
Ejecución	Bytecode JVM; también **Kotlin/Native**, **Kotlin/JS** y **Kotlin/Wasm**
Estado	🟢 **Oficial en Android**, en crecimiento en servidor y multiplataforma

---

📜 Historia

JetBrains hacía **IntelliJ IDEA**, y su producto estaba escrito en Java. En **2010** tenían dos problemas: Java evolucionaba despacio —los genéricos habían tardado ocho años, las lambdas tardarían diez— y Scala, que les gustaba, **compilaba demasiado lento** para una base de código de su tamaño.

Así que hicieron el suyo, con dos requisitos poco habituales y muy reveladores:

1. **Interoperabilidad total con Java, en las dos direcciones.** Nada de migrar: **poder mezclar ficheros `.java` y `.kt` en el mismo proyecto**.
2. **Compilar tan rápido como Java.**

Esa primera decisión es la que explica la adopción: **no hubo que elegir**. Es la estrategia opuesta a la de D frente a C++ (clase 175), y funcionó.

El punto de inflexión llegó en **2017**, cuando **Google anunció Kotlin como lenguaje oficial de Android**, y en **2019** lo declaró preferente. El contexto ayudaba: Google estaba en pleito con Oracle por las APIs de Java, y Kotlin ofrecía una salida.

Kotlin 2.0 (2024) trajo el compilador **K2**, con mejoras grandes de velocidad, y consolidó **Kotlin Multiplatform**: el mismo código de negocio compilado a JVM, a nativo para iOS, a JavaScript y a WebAssembly.

🏭 Dónde vive hoy

- **Android**: es el lenguaje por defecto; la mayoría de las aplicaciones nuevas se escriben en Kotlin.
- **Servidores**: Spring Boot tiene soporte de primera clase, y **Ktor** es el marco nativo del ecosistema.
- **Multiplataforma móvil**: **Compose Multiplatform** comparte interfaz y lógica entre Android, iOS, escritorio y web.
- **Guiones de construcción**: **Gradle** usa Kotlin como DSL de configuración (clase 163).
- **Herramientas y datos**: como sustituto de Java en proyectos nuevos de la JVM.

🧠 Lo que enseña: la nulabilidad en el tipo

Es la aportación más transferible de esta ficha (clase 100):

```
var nombre: String = "Ana"
nombre = null           // ✗ NO COMPILA

var apodo: String? = null    // ← el ? declara que puede faltar
println(apodo?.length)        // llamada segura: si es nulo, da nulo
println(apodo ?: "sin apodo")  // operador Elvis: valor por defecto
println(apodo!!.length)         // afirmación: falla si es nulo
```

Tony Hoare llamó a la referencia nula "su error del billón de dólares", y Kotlin lo resuelve como Rust, Swift y Elm: **separando en el sistema de tipos lo que puede faltar de lo que no**.

Y el segundo concepto, las **corrutinas con concurrencia estructurada**:

```
coroutineScope {                       // ← el ÁMBITO es la clave
    val a = async { pedirUsuario() }
    val b = async { pedirPedidos() }
    procesar(a.await(), b.await())
}   // si algo falla aquí dentro, se CANCELA todo lo lanzado en este ámbito
```

La concurrencia estructurada significa que **una tarea no puede sobrevivir al ámbito que la lanzó** (clase 135). Eso elimina de raíz las tareas huérfanas y las cancelaciones a medias, que son el problema clásico del código asíncrono — y es una idea que después han adoptado Java (Loom), Swift y Python.

Y hay una tercera cosa que Kotlin hace especialmente bien y que la clase 163 aprovecha: **los DSL**.

```
dependencies {                 // ← esto es Kotlin, no un formato de configuración
    implementation("org.jetbrains:annotations:24.0.0")
}
```

Las lambdas con receptor y las funciones de extensión permiten construir lenguajes de dominio que parecen configuración y **están comprobados por el compilador** — que es lo que Gradle usa.

🔄 Lo que se ha modernizado

- **Compilador K2 (2.0)**: mucho más rápido, con mejor inferencia y análisis compartido entre plataformas.
- **Kotlin Multiplatform** estable: lógica compartida y interfaz nativa, con `expect` / `actual` para lo específico de cada plataforma.
- **Kotlin/Wasm** con WasmGC (clase 162).
- **Clases de valor** (`value class`) sin coste en ejecución, y **contratos** que ayudan al análisis de flujo.
- **Y `kotlinx` completo**: serialización con generación de código en compilación —sin reflexión (clase 159)—, fechas, corrutinas y entrada/salida.

⚙️ Cómo se ejecuta hoy

```
kotlinc main.kt -include-runtime -d venta.jar && java -jar venta.jar
kotlin main.kts                                  # como guion, sin compilar

./gradlew build test                              # lo habitual (clases 143 y 147)
./gradlew ktlintCheck detekt                       # estilo y análisis (clase 146)
```

🍼 El programa de la clase 041 en Kotlin

```
fun main() {
    val (precio, cantidad, descuento) = readLine()!!.split(" ").map { it.toDouble() }
    val total = precio * cantidad * (1 - descuento)
    println("Total: %.2f".format(total))
}
```

Lo que hay que ver, comparando con Java línea a línea.

- **Cuatro líneas frente a quince**, sobre la misma máquina virtual y con la misma biblioteca. Esa comparación es, por sí sola, el argumento del lenguaje.
- **La desestructuración** `val (a, b, c)` no existe en Java, y es de las cosas que más se echan de menos al volver.
- **El** `!!` **después de** `readLine()` es la nulabilidad en acción: `readLine()` devuelve `String?` porque **puede no haber línea**, y `!!` afirma que la hay. **En Java eso sería un** `null` **silencioso esperando a explotar**, y aquí está escrito en el código.
- `val` **es inmutable** — `var` es la mutable—, y la convención del ecosistema es usar `val` por defecto (clase 102), igual que Rust y Scala.
- `"%.2f".format(total)` llama por debajo a `String.format` de Java: **la interoperabilidad no es una capa, es el mismo objeto**. Y por eso `Locale` sigue siendo una trampa a vigilar, igual que en Java.

📚 Fuentes y bibliografía

- kotlinlang.org/docs (https://kotlinlang.org/docs/home.html) — documentación oficial, con el tour interactivo (https://kotlinlang.org/docs/kotlin-tour-welcome.html).
- Guía de convenciones de código (https://kotlinlang.org/docs/coding-conventions.html) — la de la clase 146.
- **Dmitry Jemerov, Svetlana Isakova, Roman Elizarov**, *Kotlin in Action*, 2.ª ed., Manning — el libro de referencia; Elizarov diseñó las corrutinas.
- **Marcin Moskała**, *Effective Kotlin* — 50 elementos al estilo de *Effective Java*.
- Blog de Kotlin y KEEP (https://github.com/Kotlin/KEEP) — las propuestas de evolución, con su discusión.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Java · Scala · Swift · Groovy · C#

🌙 Lua — 1993

Lua nació en Brasil para que unos ingenieros de Petrobras pudieran configurar simulaciones sin recompilarlas, y hoy es **el lenguaje incrustado más usado del mundo**: está en World of Warcraft, en Roblox, en Redis, en Nginx y en las plantillas de Wikipedia. Su intérprete completo **cabe en unos 200 kilobytes**.

> ### 🎯 *Por qué está en este programa*
>
> *Lua es un* ***primo de la familia de scripting dinámico*** *(Atlas) y* ***uno de los tres primos verificados en CI****, junto a Ruby y Perl.*
>
> *Aporta al programa el caso central de una clase entera:* ***incrustar un lenguaje en otro*** *(clase 163), con las cuatro decisiones que eso obliga a tomar. Y aporta* ***la tabla como estructura única*** *(clase 099) y* ***las corrutinas*** *(clase 134) en su forma más simple.*

Año	1993; **5.1** (2006) es la base de LuaJIT; **5.4** la actual
Autoría	**Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes** — PUC-Rio, Brasil
Familia	Scripting dinámico; con influencia de Scheme y de lenguajes de configuración
Paradigma	Multiparadigma; OO **por prototipos**, con metatablas
Tipado	**Dinámico**, con ocho tipos en total
Memoria	Recolector incremental o generacional
Ejecución	Bytecode sobre una VM de registros; **LuaJIT** compila a nativo
Estado	🟢 **Dominante** como lenguaje incrustado; licencia MIT

📜 Historia

En **1993**, el grupo Tecgraf de la **Pontificia Universidad Católica de Río de Janeiro** trabajaba para **Petrobras**. Los ingenieros necesitaban configurar programas de simulación y visualización de datos geológicos, y los ficheros de configuración se les habían quedado cortos: querían condicionales, cálculos y variables (clase 163).

Y había una restricción de contexto que resultó decisiva: **Brasil tenía una política de reserva de mercado para la informática**, así que comprar software extranjero era difícil. **Tuvieron que hacerlo ellos.**

Diseñaron Lua —"luna" en portugués— con unos objetivos muy concretos que explican todo lo demás:

1. **Que se pueda incrustar en un programa de C** con una API pequeña.
2. **Que sea diminuto y portable**: C89 puro, sin dependencias.
3. **Que sea rápido**.
4. **Y que un ingeniero que no es programador pueda escribirlo.**

LuaJIT (Mike Pall, 2005) llevó el rendimiento a otro nivel: **es uno de los compiladores JIT más rápidos que existen para un lenguaje dinámico**, y sigue basado en Lua 5.1 — lo que ha partido el ecosistema en dos ramas que conviven.

Y el momento en que el mundo lo conoció fue **World of Warcraft (2004)**, que expuso toda su interfaz como guiones Lua y creó un ecosistema de miles de complementos escritos por jugadores.

🏭 Dónde vive hoy

- **Videojuegos**: World of Warcraft, **Roblox** (con Luau, su variante tipada), Garry's Mod, Angry Birds, Love2D, y como lenguaje de guion en decenas de motores.
- **Infraestructura de red**: **OpenResty/Nginx** —lógica de peticiones en Lua—, HAProxy, Kong.
- **Bases de datos**: **Redis** ejecuta guiones Lua de forma atómica, lo que resuelve operaciones compuestas sin condiciones de carrera (clase 161).
- **Wikipedia**: el módulo Scribunto ejecuta Lua para las plantillas, en los servidores de Wikimedia.
- **Editores y herramientas**: **Neovim** lo adoptó como lenguaje de configuración y extensión.
- **Sistemas embebidos**: routers (OpenWrt), televisores, dispositivos industriales.

🧠 Lo que enseña: hacer mucho con muy poco

Uno, la tabla es la única estructura de datos (clase 099):

```
local t = {}
t[1] = "primero"          -- array
t.nombre = "Ana"           -- registro
t["clave"] = valor          -- diccionario
```

Arreglo, registro, diccionario, objeto, espacio de nombres y módulo son la misma cosa. Es minimalismo llevado al extremo, y funciona: la biblioteca estándar entera se construye con tablas.

Dos, las metatablas dan orientación a objetos sin tenerla:

```
local Cuenta = {}
Cuenta.__index = Cuenta        -- ← si no encuentras el campo, búscalo aquí

function Cuenta.new(saldo)
  return setmetatable({saldo = saldo}, Cuenta)
end
function Cuenta:depositar(x) self.saldo = self.saldo + x end
```

`__index` **es la herencia por prototipos** (clase 112), la misma idea que JavaScript y Self — construida con el mecanismo general de metatablas, que también permite sobrecargar operadores y controlar el acceso.

Y tres, las corrutinas (clase 134), que Lua tiene desde 1993:

```
local co = coroutine.create(function()
  for i = 1, 3 do coroutine.yield(i) end
end)
print(coroutine.resume(co))   -- true 1
```

Multitarea cooperativa con una pila propia, que es la base de los generadores y de `async` / `await` de casi todos los lenguajes modernos.

Y para la clase 163, lo importante es **cómo se incrusta**:

```
lua_State *L = luaL_newstate();
luaL_openlibs(L);                     /* ← o NO abrirlas: se elige QUÉ existe */
lua_register(L, "dibujar", c_dibujar);
luaL_dofile(L, "config.lua");
lua_close(L);
```

> ***La línea comentada es la clave y es la razón de su éxito**: **un estado de Lua recién creado no tiene `io`, ni `os`, ni nada**. El anfitrión abre las bibliotecas que quiera y registra las funciones que quiera. **Es el modelo de capacidades de la clase 153**, integrado en el diseño — y por eso Wikipedia puede ejecutar plantillas escritas por cualquiera sin que tumben el servidor.*

🔄 Lo que se ha modernizado

- **Lua 5.4**: recolector generacional, enteros de 64 bits separados de los reales (5.3), y variables `<close>` para liberación determinista (clase 132).
- **LuaJIT**: JIT de trazas y una interfaz con C (`ffi`) que llama a bibliotecas nativas **sin escribir envoltorios** (clase 156).
- **Luau** (Roblox): Lua con **tipado gradual**, comprobación estática y aislamiento reforzado — con millones de personas escribiéndolo.
- **LuaRocks** como gestor de paquetes (clase 143).

- **Y ganchos de límite de instrucciones** (`lua_sethook`), que permiten cortar un guion que no termina — la segunda decisión del cierre de la clase 163.

⚙️ Cómo se ejecuta hoy

```
lua main.lua < entrada.txt        # el comando de la clase 041
luajit main.lua                    # el JIT, muy superior en cálculo

luarocks install <paquete>
```

🧹 El programa de la clase 041 en Lua

Esta versión **se ejecuta y se verifica en CI**.

```
local precio, cantidad, descuento = io.read("n", "n", "n")
local total = precio * cantidad * (1 - descuento)
print(string.format("Total: %.2f", total))
```

Lo que hay que ver.

- `io.read("n", "n", "n")` **lee tres números directamente**, sin partir la línea ni convertir. Es la lectura más limpia de las veinte versiones de la clase 041 — el intérprete hace el análisis.
- `local` **no es opcional en la práctica**: sin él, la variable sería **global**, que es el mismo problema que M tiene por defecto (clase 146). La regla del ecosistema es tajante: `local` **siempre**.
- `string.format` **con** `%.2f` es, otra vez, la herencia de C — Lua está escrito en C y su biblioteca lo refleja.
- **Y el detalle que no se ve**: hasta Lua 5.3, **todos los números eran reales de 64 bits**, como en JavaScript. Desde 5.3 hay enteros separados, y esa distinción importa en índices y en identificadores grandes (clase 045).

📚 Fuentes y bibliografía

- lua.org/manual (https://www.lua.org/manual/5.4/) — el manual completo cabe en unas cien páginas; es de lo más corto y preciso de esta lista.
- **Roberto Ierusalimschy**, *Programming in Lua*, 4.ª ed. — escrito por el autor del lenguaje; la primera edición está libre en línea.
- Lua PIL y la wiki de usuarios (http://lua-users.org/wiki/) — patrones e idiomas de la comunidad.
- **Roberto Ierusalimschy et al.**, *The Evolution of Lua* (HOPL III, 2007) — el artículo que cuenta la historia y las decisiones de diseño; excelente lectura para la clase 163.
- LuaJIT.org (https://luajit.org/) — y la documentación de su `ffi`, que es un caso de estudio de la clase 156.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Tcl · JavaScript · Python · C

📐 MATLAB — 1984

MATLAB empezó como una interfaz cómoda para las bibliotecas de álgebra lineal en Fortran —de ahí el nombre, *MATrix LABoratory*— y se convirtió en el entorno estándar de la ingeniería mundial. Es de los pocos lenguajes de esta lista que **es a la vez un lenguaje y un producto comercial**, y esa dualidad explica sus fortalezas y su principal debilidad.

> 🎯 ***Por qué está en este programa***
>
> *MATLAB es un **primo de la familia array / científica** (Atlas), junto a APL, J, R, Julia y Fortran.*

> *Aporta al programa dos cosas:* ***la programación vectorizada aplicada a la ingeniería*** *(clase 089) y el caso de estudio más claro de* ***generación de código desde un modelo*** *—Simulink— que la clase 155 nombra como cuarta capa de un sistema poliglota, y que en aviación y automoción es donde de verdad se escribe el software.*

Año	1984 (comercial); el prototipo, de finales de los setenta
Autoría	**Cleve Moler**, Universidad de Nuevo México; **MathWorks** desde 1984
Familia	Array / científica; construido sobre **LINPACK** y **EISPACK** (Fortran)
Paradigma	Imperativo y vectorizado; con objetos añadidos después
Tipado	**Dinámico**; **todo es una matriz** de dobles por defecto
Memoria	Automática, con **copia al modificar**
Ejecución	Interpretado con JIT; con generación de C y de HDL como productos
Estado	🟢 **Dominante en ingeniería**; **propietario y caro**, con GNU Octave como alternativa

---

📜 Historia

Cleve Moler enseñaba análisis numérico en la Universidad de Nuevo México en los años setenta, y sus estudiantes tenían que escribir Fortran para usar **LINPACK** y **EISPACK** —las bibliotecas de álgebra lineal de la época, precursoras de LAPACK (clase 149)—.

Escribir Fortran para hacer un ejercicio de matrices era desproporcionado, así que Moler hizo **un intérprete que llamaba a esas bibliotecas** con una notación matricial directa. **No pretendía ser un producto**: era material docente, y lo repartía libremente.

En **1983**, el ingeniero **Jack Little** vio el potencial, lo reescribió en C, le añadió funciones y gráficos, y fundó **MathWorks** con Moler y Steve Bangert. **MATLAB salió al mercado en 1984**.

Y el momento que definió su posición industrial fue **Simulink (1990)**: un entorno **gráfico** de modelado por bloques para sistemas dinámicos y de control. Con **Simulink Coder** y **Embedded Coder**, **del modelo se genera código C o Ada certificable** que se ejecuta en el vehículo real.

Eso cambió qué significa "programar" en esos sectores: en buena parte de la automoción y la aviación, **el ingeniero de control dibuja el modelo y el código lo genera la herramienta** (clase 155) — y lo que se certifica es la cadena de generación, no el código a mano.

🏭 Dónde vive hoy

- **Ingeniería de control y sistemas embebidos**: automoción, aeroespacial, energía. **Simulink es el estándar del sector.**
- **Procesamiento de señales e imágenes**, comunicaciones y radar.
- **Investigación en ingeniería y enseñanza universitaria**: casi cualquier titulación de ingeniería del mundo pasa por MATLAB.
- **Biomedicina y neurociencia**, con herramientas específicas muy asentadas.
- **Y en modelos financieros**, con las cajas de herramientas correspondientes.

🧠 Lo que enseña: matrices por defecto, y el coste de un ecosistema propietario

La vectorización, que es la misma idea de R y APL (clase 089):

```
A = [1 2; 3 4];
b = [5; 6];
x = A \ b;              % ← resolver Ax = b. Una barra invertida.
y = sin(0:0.1:2*pi);     % la función se aplica a TODO el vector
```

`A \ b` **es probablemente la operación más elegante de esta lista**: resuelve el sistema lineal eligiendo el algoritmo según la forma de la matriz —LU, QR, Cholesky o mínimos cuadrados—. **Es LAPACK por debajo** (clase 149), y esa es exactamente la razón de ser del lenguaje.

Y hay dos peculiaridades que conviene conocer:

Una, todo es una matriz de dobles. Un escalar es una matriz 1×1, y el tipo por defecto es `double`. Eso simplifica el modelo y **hace que trabajar con enteros o con memoria sea antinatural**.

Y dos, los índices empiezan en 1 y se usan paréntesis: `A(1,2)` es a la vez la indexación de una matriz y la llamada a una función, según qué sea `A` — una ambigüedad sintáctica que Julia heredó y que Fortran también tiene.

> ***Y la ficha debe ser honesta con lo que más pesa en la clase 164: MATLAB es propietario y caro.***
>
> *Una licencia completa con cajas de herramientas cuesta miles de euros al año, y **cada caja de herramientas se compra aparte**. Eso tiene tres consecuencias reales: **el código no se puede ejecutar sin licencia**, lo que complica la reproducibilidad (clase 154); **la comunidad no puede corregir el lenguaje**; y **quien aprende MATLAB en la universidad se encuentra con que en su empresa no hay licencias**.*
>
> ***GNU Octave** es una reimplementación libre muy compatible para el lenguaje base, y **no cubre Simulink** — que es justamente donde está el valor industrial. Y esa es la razón por la que Julia y Python+SciPy le han ido ganando terreno en investigación, aunque no en ingeniería de control.*

🔄 Lo que se ha modernizado

- **JIT** desde 2002 y mejoras continuas: los bucles ya no son catastróficos, aunque la vectorización sigue siendo lo idiomático.
- `arguments` **blocks** (R2019b): validación de argumentos declarativa, con tipos y restricciones — contratos en el lenguaje (clase 118).
- `string` **como tipo** (2016), separado de los arreglos de caracteres — la corrección de un problema histórico (clase 093).
- **Interoperabilidad**: llamar a Python, C++, Java y .NET desde MATLAB, y al revés (clase 156).
- **MATLAB Online y Live Scripts**: cuadernos ejecutables con texto, código y resultados (clase 154).
- **Y la generación de código** cada vez más central: C, C++, HDL para FPGA y CUDA para GPU.

⚙️ Cómo se ejecuta hoy

```
matlab -batch "main"                 # ejecutar sin interfaz
octave --no-gui main.m < entrada.txt   # ← la alternativa libre

# Y dentro: runtests para las pruebas, y el Profiler para la clase 152
```

🧪 El programa de la clase 041 en MATLAB

Esta versión se escribe aquí y **no está verificada en CI** (clase 040). Funciona igual en **GNU Octave**.

```
v = sscanf(input('', 's'), '%f');
total = v(1) * v(2) * (1 - v(3));
fprintf('Total: %.2f\n', total);
```

Lo que hay que ver.

- `sscanf(..., '%f')` **devuelve un VECTOR con todos los números que encuentra**, no uno solo. Es el reflejo de la familia: **la unidad es el arreglo**, igual que en R y APL.
- `v(1)` **con paréntesis**, no con corchetes: en MATLAB la indexación usa la misma sintaxis que la llamada a función.
- **Los índices empiezan en 1** (clase 089).
- `fprintf` **con** `%.2f` es, otra vez, la herencia de C — MATLAB está escrito en C y su entrada y salida lo refleja.
- **Y el detalle que delata al lenguaje**: `v` es un `double` de 64 bits **aunque los tres valores fueran enteros**. En MATLAB **todo es coma flotante salvo que se pida lo contrario**, con las consecuencias de la clase 045.

📚 Fuentes y bibliografía

- Documentación de MathWorks (https://www.mathworks.com/help/matlab/) — extensa y de calidad alta; la sección de rendimiento es material de la clase 152.
- **Cleve Moler**, *Numerical Computing with MATLAB* — libre en la web de MathWorks; escrito por el autor original y excelente como libro de análisis numérico.
- Blog de Cleve Moler (https://blogs.mathworks.com/cleve/) — historia del lenguaje y del cálculo numérico, contada por quien estuvo.
- GNU Octave (https://octave.org/doc/latest/) — la alternativa libre y su documentación.
- **Documentación de Simulink y Embedded Coder** — para entender la generación de código certificable (clases 155 y 174).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Fortran · Julia · R · APL · Ada

🗄️ M / MUMPS — 1966

El lenguaje de tu historia clínica. Si te han atendido en un hospital grande, es muy probable que tu expediente viva en una base de datos MUMPS. Es un lenguaje de 1966 que sostiene una parte enorme de la sanidad mundial y que casi nadie fuera de ese sector ha visto nunca.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: M se ejecuta hoy en los sistemas de información clínica más grandes del mundo.*** *VistA, el sistema del Departamento de Veteranos de EE. UU., tiene componentes escritos en M y su documentación oficial lo declara. InterSystems (IRIS, antes Caché) y YottaDB son productos vivos y en desarrollo activo, no arqueología.*
>
> *Entra porque* ***encarna una idea que ningún lenguaje del núcleo tiene: el lenguaje y la base de datos son la misma cosa****. En M no existe un ORM, ni un driver, ni una conexión, ni un* `COMMIT` *obligatorio para guardar. Escribes* `set ^VENTAS(id)=total` *y eso* ***ya está en disco****, con la misma sintaxis con la que escribirías una variable local. Todo lo que en el núcleo se estudia como "persistencia" —serialización, mapeo objeto-relacional, capas de acceso a datos— aquí simplemente* ***no existe como problema****. Ver eso una vez cambia la forma de mirar cualquier ORM.*

Año	1966; primer estándar ANSI en 1977; ISO/IEC 11756
Autoría	**Neil Pappalardo**, Robert Greenes y Curt Marble, Massachusetts General Hospital
Familia	Lenguajes de base de datos / sanidad
Paradigma	Imperativo y procedimental, con persistencia integrada
Tipado	**Dinámico y sin tipos**: todo es una cadena que se interpreta según el uso
Memoria	Gestionada; los datos persistentes viven en árboles en disco
Ejecución	Interpretado / compilado a bytecode según la implementación
Estado	🟡 **Muy especializado y difícil de sustituir** — sanidad, algo de finanzas

---

📜 Historia

En 1966, en el laboratorio de computación del **Massachusetts General Hospital**, un equipo dirigido por Neil Pappalardo necesitaba algo que no existía: un sistema en el que varios terminales pudieran consultar y actualizar historiales de pacientes **a la vez**, sobre un PDP-7 con memoria ridícula para los estándares actuales. Las bases de datos comerciales de la época eran caras, rígidas y pensadas para lotes, no para consultas interactivas de datos muy irregulares.

La solución fue radical: hacer que **el lenguaje incorporara el almacenamiento**. Nació **MUMPS** (*Massachusetts General Hospital Utility Multi-Programming System*), luego estandarizado simplemente como **M**.

Su estructura de datos única es el **array disperso multidimensional**. Un array de M no tiene tamaño ni índices numéricos obligatorios: las claves pueden ser números o texto, y solo existen los nodos que has creado. Y aquí está la idea decisiva: si el nombre del array empieza por `^`, **es persistente**. `PACIENTE("nombre")` vive en memoria y desaparece; `^PACIENTE(1234,"nombre")` está en disco, es transaccional y lo ven todos los procesos. Misma sintaxis, dos mundos.

El otro rasgo llamativo es la **brevedad extrema**. En 1966 la memoria se contaba en kilobytes y cada carácter del programa fuente ocupaba espacio, así que todos los comandos admiten abreviatura a una letra: `S` por `SET`, `W` por `WRITE`, `I` por `IF`, `F` por `FOR`, `Q` por `QUIT`, `D` por `DO`. El código real de producción está escrito así, y para un ojo no entrenado resulta casi ilegible.

🏭 Dónde sobrevive hoy

- **Historia clínica electrónica**: **Epic Systems**, el sistema de historia clínica de mayor cuota en los hospitales de Estados Unidos, se construye sobre tecnología InterSystems (Caché/IRIS), cuyo motor de datos y lenguaje descienden directamente de M. **MEDITECH** también viene de esta familia.
- **VistA**, el sistema del Departamento de Asuntos de Veteranos de EE. UU. — uno de los despliegues de software sanitario más grandes del mundo, con código abierto y componentes en M.
- **Laboratorios y sistemas departamentales** hospitalarios.
- **Sector financiero**: algunos sistemas de banca y trading europeos sobre Caché/IRIS, por la misma razón que la sanidad — rendimiento en escrituras y esquemas irregulares.

Descendiente moderno: InterSystems ObjectScript, que añade orientación a objetos, SQL y persistencia de clases sobre el mismo motor de *globals*. Un desarrollador de IRIS trabaja hoy en ObjectScript, pero el `^global` sigue estando debajo.

🧠 Por qué no ha muerto

1. El modelo de datos encaja con la medicina. Un historial clínico es irregular por naturaleza: un paciente tiene tres alergias y otro ninguna; una consulta tiene quince campos y otra dos. En un modelo relacional eso produce tablas anchas llenas de nulos o decenas de tablas de detalle. En M, un árbol disperso guarda exactamente lo que hay. Es, en la práctica, una base de datos **jerárquica y sin esquema** treinta años antes de que se acuñara "NoSQL".

2. Latencia mínima. No hay traducción entre el modelo del lenguaje y el del almacenamiento, ni capa de red, ni parseo de SQL. Una escritura es una escritura. En sistemas con miles de terminales concurrentes, eso se nota.

3. Sustituirlo obliga a reescribir el hospital. No es solo el código: es que las reglas clínicas, los formularios, los protocolos y las integraciones están tejidos en esa base de datos. Y un fallo en un sistema clínico tiene consecuencias directas sobre pacientes.

4. Los productos están vivos. InterSystems IRIS y YottaDB reciben versiones nuevas, soportan contenedores, SQL sobre los mismos datos, APIs REST y FHIR. El lenguaje es viejo; la plataforma no.

🔄 Lo que se ha modernizado

El lenguaje apenas ha cambiado; **la plataforma se ha reinventado entera**, y esa es justo la estrategia que lo mantiene vivo:

- **Enlaces para lenguajes actuales.** YottaDB expone sus *globals* a **C, Go, Python, Node.js, Rust, Perl y Java**. Puedes escribir la lógica nueva en Go y seguir leyendo y escribiendo el mismo árbol de datos que el código M de 1990. Es una estrategia de convivencia, no de migración.
- **Contenedores y nube.** Hay imágenes Docker oficiales de YottaDB e InterSystems IRIS; IRIS se despliega en Kubernetes con alta disponibilidad.
- **SQL sobre los mismos datos.** IRIS permite consultar con SQL estándar las estructuras persistentes y exponerlas por ODBC/JDBC, sin duplicar la información.
- **FHIR y HL7 nativos.** IRIS for Health implementa los estándares actuales de interoperabilidad sanitaria; es la razón principal de que la plataforma siga ganando contratos.
- **APIs REST y JSON** directamente desde el lenguaje, y un motor de aprendizaje automático (IntegratedML) que se invoca desde SQL.
- **ObjectScript** añadió clases, herencia, excepciones estructuradas y persistencia de objetos sobre el mismo motor. Un desarrollador de IRIS de 2026 rara vez escribe M puro, pero el `^global` sigue debajo de todo.

⚙️ Cómo se ejecuta hoy

```
# YottaDB: implementación libre de M, con paquetes .deb/.rpm y contenedor oficial
docker run --rm -it yottadb/yottadb-debian:latest

# Dentro del contenedor, el shell directo del lenguaje:
$ydb_dist/yottadb -direct
YDB> write "hola",!
```

Implementaciones actuales: YottaDB (libre, derivada de GT.M, con enlaces para C, Go, Python, Node.js, Rust y Perl), **GT.M** (FIS), e **InterSystems IRIS** (comercial, con ObjectScript, SQL, analítica e interoperabilidad FHIR). Para trastear con VistA existen distribuciones empaquetadas de la comunidad.

🍼 El programa de la clase 041 en M

> ⚠️ ***Material de lectura, no verificado.*** *No hay intérprete M en los runners de CI.*

```
TOTVTA ; Total de una venta -- clase 041
 read linea
 set precio    = $piece(linea, " ", 1)
 set cantidad  = $piece(linea, " ", 2)
 set descuento = $piece(linea, " ", 3)
 set total     = precio * cantidad * (1 - descuento)
 write "Total: ", $justify(total, 0, 2), !
 quit
```

Recorrido, línea a línea.

- La **primera columna es sagrada.** Una etiqueta (`TOTVTA`) empieza en la columna 1; **toda línea de código debe empezar con al menos un espacio**. Es la regla que más rompe quien empieza, y no da un error claro: el intérprete cree que estás definiendo una etiqueta.
- `;` inicia un comentario.
- `read linea` lee de la entrada. En M la E/S va contra un **dispositivo actual**, y `use` cambia cuál es; no hay una noción separada de "entrada estándar".
- `$piece(cadena, delimitador, n)` —abreviado `$P`— es la función más usada del lenguaje: devuelve el trozo *n* de una cadena partida por un delimitador. **No hay listas ni arrays de resultado**: en M la cadena delimitada *es* la estructura de datos ligera, y `$piece` es cómo se accede a ella. Verás registros enteros guardados como `"Pérez^María^1978-04-12^O+"` y accedidos con `$P(dato,"^",3)`.
- `$justify(valor, ancho, decimales)` —abreviado `$J`— formatea con decimales y justifica a la derecha. Con ancho `0` no añade relleno, así que da exactamente `27000.00`. Es el `printf("%.2f")` de M.
- `!` dentro de un `write` es el salto de línea. No es una cadena: es un **código de formato** del comando `write`.
- No hay declaración de tipos en ninguna parte. `precio` contiene la cadena `"15000"`, y al multiplicarla se interpreta como número. M es el caso extremo del tipado débil: **todo dato es una cadena** y el operador decide cómo leerla.

Y ahora la parte que hace único a M. Añade una línea:

```
 set ^VENTAS($horolog, "total") = total
```

Eso **ya está en disco**. Sin `INSERT`, sin conexión, sin `commit`, sin ORM, sin fichero abierto. `^VENTAS` es un *global*: un árbol persistente, transaccional y compartido por todos los procesos. Otro proceso puede leerlo inmediatamente con `set x = ^VENTAS(clave,"total")`. Y se puede recorrer sin conocer las claves:

```
 set clave = ""
 for  set clave = $order(^VENTAS(clave))  quit:clave = ""  do
 . write clave, " -> ", ^VENTAS(clave, "total"), !
```

`$order` da la siguiente clave existente en orden. Es el `SELECT` de M: no describes qué quieres, recorres el árbol. El `quit:condición` es un **postcondicional**, otra marca de la casa: casi cualquier comando admite `:condición` para ejecutarse solo si se cumple. Y el punto que abre la última línea marca el **nivel de anidamiento del bloque**.

Si esta sintaxis te parece de otro planeta, es porque lo es: M no desciende de ALGOL como casi todo lo demás en este Atlas. Es una rama evolutiva independiente.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces...	En M es...
`x = 5`	`set x=5` (o `S x=5`)
`print(x)`	`write x,!`
`if cond: ...`	`if cond do ...` o el postcondicional `write:cond x`
`for i in range(1,10)`	`for i=1:1:10 do ...`
`dict["a"]["b"]`	`array("a","b")` — arrays multidimensionales dispersos
`INSERT INTO ventas ...`	`set ^VENTAS(clave)=valor` — sin más
`SELECT ... ORDER BY`	Recorrido con `$order` sobre el global
`"a,b,c".split(",")[2]`	`$piece("a,b,c", ",", 2)`
`f"{x:.2f}"`	`$justify(x, 0, 2)`
Transacción	`tstart` / `tcommit` — existen, pero no hacen falta para persistir

⚠️ Errores comunes al leerlo

- **La columna 1.** Código en la columna 1 = etiqueta. Es el error de novato universal en M.
- **Confundir `^` con "puntero" o con XOR.** El circunflejo al inicio de un nombre significa **persistente**. En medio de una cadena de datos, suele ser simplemente el delimitador de campos por convención — dos usos distintos del mismo carácter.
- **Buscar tipos.** No hay. `1` y `"1"` son lo mismo. `"12ABC" + 1` da `13`, porque M convierte leyendo el prefijo numérico y descartando el resto, sin avisar.
- **Leer el código abreviado como ofuscación intencionada.** `S X=$P(L,U,2)` es idioma normal, no código sucio. Aprende las abreviaturas antes de juzgar.
- **Espacios que importan.** Un `for` sin argumentos (bucle infinito) lleva **dos espacios** antes del siguiente comando. Un espacio de más o de menos cambia el significado.
- **El punto de anidamiento.** Los `.` al comienzo de línea marcan bloques dentro de un `do`. No son decoración.

📚 Fuentes y bibliografía

- YottaDB — documentación (https://docs.yottadb.com/) — la implementación libre; el *Programmer's Guide* es la mejor referencia gratuita del lenguaje.
- InterSystems IRIS (https://docs.intersystems.com/) — documentación del producto comercial y de **ObjectScript**, el descendiente moderno.
- VistA — Departamento de Asuntos de Veteranos de EE. UU. (https://www.va.gov/health/vista.asp) — el sistema abierto donde ver M aplicado a escala real.
- The M Technology Association (MTA) (https://mtaonline.org/) — comunidad y materiales del estándar.
- **Richard Walters**, *M Programming: A Comprehensive Guide*, Digital Press — el libro clásico y todavía el más completo sobre el lenguaje.
- **Ed de Moel**, *M[UMPS] by Example* — referencia práctica y compacta, disponible en línea.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Common Lisp · Smalltalk · COBOL

👑 Nim — 2008

Nim demuestra que **"parecerse a Python" y "compilar como C" no son incompatibles**. Tiene sangría significativa, inferencia de tipos y una sintaxis ligera — y produce un binario nativo sin tiempo de ejecución pesado, con macros que operan sobre el árbol sintáctico.

> 🎯 ***Por qué está en este programa***
>
> *Nim es un **primo de la familia C / llaves** (Atlas) en el eje de sistemas, junto a Zig y D.*
>
> *Aporta al programa una comparación muy útil: **es la prueba de que la sintaxis y el modelo de ejecución son decisiones independientes**. Quien crea que un lenguaje "de guion" es lento por su aspecto, aquí ve lo contrario. Y aporta **macros higiénicas sobre el árbol sintáctico** en un lenguaje de tipado estático (clase 122), que es una combinación poco frecuente.*

Año	2008 (como Nimrod); **1.0** en 2019; **2.0** en 2023
Autoría	**Andreas Rumpf**, con una comunidad pequeña y activa
Familia	Sistemas; sintaxis de Python, semántica de Pascal/Modula y C
Paradigma	Multiparadigma: imperativo, orientado a objetos, funcional y metaprogramación
Tipado	**Estático y fuerte**, con inferencia; genéricos y conceptos
Memoria	**ARC/ORC**: conteo de referencias con detección de ciclos, en compilación
Ejecución	**Compila a C, C++ o JavaScript**, y de ahí a nativo
Estado	🟡 **Estable y minoritario**: excelente y con poca adopción industrial

---

📜 Historia

Andreas Rumpf empezó Nimrod en **2008** con un objetivo poco común: **un lenguaje tan expresivo como Python, tan rápido como C y tan seguro como Ada**. La estrategia para conseguirlo fue pragmática y muy eficaz: **no generar código máquina, sino generar C**.

Esa decisión —la misma que tomó GnuCOBOL y que en su día tomó C++ con `cfront`— le dio gratis lo que a un lenguaje nuevo le cuesta años: **portabilidad a cualquier plataforma con un compilador de C**, incluidos microcontroladores, y **acceso directo a todas las bibliotecas de C** (clase 156).

El cambio de nombre a **Nim** llegó en 2014 —Nimrod tiene connotaciones desafortunadas en inglés—, la **1.0** en 2019, y la **2.0** en 2023 con **ORC por defecto**: gestión de memoria determinista, sin recolector con pausas.

Y ahí está su posición actual: **un lenguaje técnicamente muy logrado con una comunidad pequeña** — que es, como la clase 164 explica, un factor que pesa más que casi cualquier característica.

🏭 Dónde vive hoy

- **Herramientas de línea de comandos y utilidades** donde se quiere un binario pequeño y rápido con poco código (clase 167).
- **Sistemas embebidos**: al generar C, llega a plataformas donde no hay LLVM.
- **Extensiones nativas para Python**, con `nimpy` (clase 158).
- **Desarrollo web**: Karax compila Nim a JavaScript para el navegador (clase 169).
- **Bioinformática y ciencia**: `hts-nim` y varias herramientas de genómica.

🧠 Lo que enseña: la sintaxis no determina el rendimiento

```
import std/[strutils, sequtils]

let numeros = "1 2 3 4".splitWhitespace().map(parseInt)
echo numeros.sum()
```

Eso podría ser Python, y **compila a un binario nativo sin intérprete**. Es la demostración más directa de una idea que este curso repite (clases 123 y 164): **el aspecto de un lenguaje y su modelo de ejecución son decisiones separadas**.

Y Nim tiene tres características que merecen conocerse:

Uno, la identificación de nombres es *insensible al estilo*:

```
proc miFuncion() = discard
mifuncion()      # ✓ lo mismo
mi_funcion()      # ✓ también: se ignoran los guiones bajos y las mayúsculas (menos la primera)
```

Es una decisión discutida —y muy práctica— para acabar con las guerras de estilo de nombres (clase 146): **cada quien escribe como quiera y el compilador los une**.

Dos, la notación uniforme de llamada:

```
len(cadena)      # y
cadena.len        # son EXACTAMENTE lo mismo
```

No hace falta que la función sea un método para escribirla con punto. Eso permite encadenar sin que la biblioteca lo haya previsto, y es lo que D también tiene.

Y tres, las macros, que son lo más potente del lenguaje:

```
macro miDsl(cuerpo: untyped): untyped =
  # recibe el ÁRBOL SINTÁCTICO y devuelve otro árbol
```

Son macros sobre el AST, higiénicas y con tipos — la potencia de Lisp (clase 122) en un lenguaje estático. Con ellas se construyen los generadores de HTML, los ORM y los DSL del ecosistema, sin coste en ejecución.

🔄 Lo que se ha modernizado

- **ORC por defecto (2.0)**: conteo de referencias insertado por el compilador **con detección de ciclos**, sin recolector con pausas — determinista, apto para tiempo real (clase 131).
- `nimble` como gestor de paquetes con fichero de bloqueo (clase 143).
- **Comprobación de efectos**: se puede declarar que una función **no lanza excepciones** o **no reserva memoria**, y el compilador lo verifica — la idea de la clase 146 llevada al sistema de tipos.
- **Objetivo JavaScript** como destino de primera clase (clase 162).

⚙️ Cómo se ejecuta hoy

```
nim c -r main.nim < entrada.txt          # compilar y ejecutar
nim c -d:release --opt:speed main.nim     # binario optimizado
nim js main.nim                            # ← el mismo código, a JavaScript

nimble install && nimble test              # dependencias y pruebas
```

🧪 El programa de la clase 041 en Nim

```
import std/[strutils, sequtils, strformat]

let v = stdin.readLine().splitWhitespace().map(parseFloat)
let total = v[0] * v[1] * (1 - v[2])
echo &"Total: {total:.2f}"
```

Lo que hay que ver.

- **Tres líneas, y compila a un ejecutable nativo.** Compárese con la versión de C —que hace lo mismo en quince— y con la de Python, que se le parece y necesita un intérprete.
- `let` **es inmutable**, como en Rust; para poder reasignar hay que escribir `var` (clase 102).
- `&"..."` **es una macro de interpolación** con formato comprobado **en compilación**: si el formato no encaja con el tipo, **no compila** (clase 142). Es la macro del punto anterior aplicada a algo cotidiano.
- **La sangría marca los bloques**, como en Python — pero el tipo de `v` está inferido y comprobado, y `v[0] * v[1]` es aritmética de coma flotante nativa, sin objetos ni despacho.

📚 Fuentes y bibliografía

- nim-lang.org/documentation (https://nim-lang.org/documentation.html) — manual, referencia de la biblioteca y el tutorial oficial.
- Nim by Example (https://nim-by-example.github.io/) — introducción práctica corta.
- Nim in Action (https://www.manning.com/books/nim-in-action) — **Dominik Picheta**, Manning; el libro de referencia, escrito por uno de los principales colaboradores.
- Foro y repositorio de Nim (https://forum.nim-lang.org/) — comunidad pequeña y muy accesible, con el propio autor respondiendo.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Python · C · D · Zig · Pascal

🧮 Objective-C — 1984

Objective-C es **C más Smalltalk**, literalmente: la sintaxis de mensajes de uno pegada sobre el otro, sin mezclarlos. Durante treinta años fue el lenguaje del Mac y del iPhone, y aunque Swift lo ha sustituido para lo nuevo, **hay demasiado código y demasiadas APIs escritas en él como para que desaparezca**.

> 🎯 ***Por qué está en este programa***
>
> *Objective-C es un **primo de la familia C / llaves** (Atlas), cuyo representante en el núcleo es C.*
>
> *Aporta al programa algo que ningún otro lenguaje de la lista enseña con tanta claridad: **el despacho dinámico de mensajes en un lenguaje compilado** (clases 111 y 112). En Objective-C **cualquier objeto puede recibir cualquier mensaje**, y eso se decide en ejecución — el modelo de Smalltalk, sobre memoria y punteros de C.*

Año	1984; **Objective-C 2.0** en 2006; **ARC** en 2011
Autoría	**Brad Cox** y **Tom Love**, Stepstone; adoptado por **NeXT** y luego Apple
Familia	C / llaves — con el modelo de objetos de Smalltalk
Paradigma	Imperativo + orientado a objetos por **paso de mensajes**
Tipado	**Estático para C, dinámico para los objetos**; `id` es "cualquier objeto"
Memoria	Conteo de referencias; manual hasta 2011, **automático (ARC)** desde entonces
Ejecución	Compilado a nativo (Clang), con un tiempo de ejecución que resuelve los mensajes
Estado	🟡 **Mantenimiento**: no se elige para lo nuevo, y hay muchísimo escrito

---

📜 Historia

A principios de los ochenta, **Brad Cox** quería la reutilización de componentes que Smalltalk prometía, pero con el rendimiento y la portabilidad de C. Su solución fue **no mezclar los dos lenguajes**: añadir a C **una sintaxis nueva y separada** —los corchetes— para el envío de mensajes, dejando C intacto por debajo.

Steve Jobs lo eligió para NeXTSTEP en 1988, y ahí se construyeron las bibliotecas que hoy siguen vivas: las clases con prefijo `NS` —de **NeXTSTEP**— que cualquiera que haya tocado macOS o iOS reconoce.

Cuando Apple compró NeXT en 1996, NeXTSTEP se convirtió en **Mac OS X**, y Objective-C pasó a ser el lenguaje de Apple. Con el **iPhone SDK (2008)** se volvió, de golpe, uno de los lenguajes más demandados del mundo.

Objective-C 2.0 (2006) trajo propiedades, enumeración rápida y recolección de basura —después descartada—; y **ARC (2011)** automatizó el conteo de referencias, que hasta entonces se escribía a mano con `retain` y `release`.

En **2014** Apple presentó Swift y Objective-C entró en mantenimiento. No ha desaparecido: **buena parte de los marcos de Apple sigue siendo Objective-C por dentro**, y la interoperabilidad entre los dos es diaria.

🏭 Dónde vive hoy

- **Aplicaciones de iOS y macOS anteriores a 2015**, que son muchísimas y siguen mantenidas.
- **Los marcos de Apple**: Foundation, AppKit y buena parte de UIKit tienen implementación Objective-C, aunque se usen desde Swift.
- **Bibliotecas y SDK de terceros** que aún no se han migrado.
- **GNUstep**: la reimplementación libre de los marcos de NeXT, que sigue viva.

🧠 Lo que enseña: el mensaje no es una llamada

Esta es la idea que hay que llevarse:

```
[objeto hacerAlgoCon:valor y:otro];      // NO es objeto.hacerAlgo(valor, otro)
```

Enviar un mensaje no es llamar a un método: es pedirle al tiempo de ejecución que busque qué método corresponde, **en ejecución**, por el nombre del selector.

Y de ahí salen tres consecuencias que la clase 111 desarrolla:

- **Se puede enviar un mensaje a `nil` y no pasa nada** — devuelve cero y sigue. Es una decisión deliberada que elimina montañas de comprobaciones y que a la vez esconde errores.

- **Se puede preguntar en ejecución** si un objeto responde a un mensaje (`respondsToSelector:`), y **añadir métodos a una clase existente** con las *categorías* — lo que en Ruby se llama clase abierta y en C# métodos de extensión.
- **Y existe** `forwardInvocation:`, el equivalente de `doesNotUnderstand:` de Smalltalk (clase 158): un objeto puede **interceptar los mensajes que no entiende** y reenviarlos. Es la base de los objetos proxy y de la simulación en pruebas, sin biblioteca.

> ***Y el precio es el de siempre en este curso*** *(clase 164): esa flexibilidad* ***impide comprobar en compilación*** *que el mensaje existe. Enviar un selector mal escrito compila con un aviso y falla al ejecutarse. Swift eligió lo contrario, y esa es la diferencia principal entre los dos.*

🔄 Lo que se ha modernizado

- **ARC** (2011): el compilador inserta `retain` / `release`, con lo que la gestión manual desapareció — sin recolector y sin pausas (clase 131).
- **Literales y suscripción** (`@[]`, `@{}`, `array[0]`), que hicieron el lenguaje mucho menos verboso.
- **Anotaciones de nulabilidad** (`nullable`, `nonnull`) y **genéricos ligeros**, añadidos sobre todo **para que Swift pueda importar las APIs con tipos precisos** — la interoperabilidad guiando el diseño.
- **Módulos** (`@import`) en lugar de `#import` textual, con compilación mucho más rápida (clase 149).

⚙️ Cómo se ejecuta hoy

```
clang -framework Foundation main.m -o venta        # macOS
gcc `gnustep-config --objc-flags` main.m -o venta   # GNUstep, en Linux
```

🧪 El programa de la clase 041 en Objective-C

Es la versión que aparece en el `primos.md` de la clase 041.

```
#import <Foundation/Foundation.h>

int main(void) {
    @autoreleasepool {
        double precio, cantidad, descuento;
        scanf("%lf %lf %lf", &precio, &cantidad, &descuento);
        double total = precio * cantidad * (1 - descuento);
        printf("Total: %.2f\n", total);
    }
    return 0;
}
```

Lo que hay que ver.

- **Es C tal cual.** `scanf`, `printf`, `double`, `&precio`: **el programa de la ficha de C compila aquí sin tocar nada**. Esa es la tesis del lenguaje — la capa de objetos **se añade**, no sustituye.
- `@autoreleasepool` **es lo único que no es C**, y marca el ámbito donde se liberan los objetos aplazados. Con ARC ya casi no hace falta, y aparece por costumbre.
- **Lo que este ejemplo no enseña son los corchetes**, porque no crea ningún objeto. La versión idiomática usaría `NSString` y `NSScanner`, y ahí aparecería `[cadena doubleValue]` — el paso de mensajes.
- **Y esa mezcla es exactamente la seña de identidad**: en un mismo fichero conviven la aritmética de C y el modelo de objetos de Smalltalk, **sin fundirse**.

📚 Fuentes y bibliografía

- Documentación de Apple para Objective-C (https://developer.apple.com/documentation/objectivec) y la guía *Programming with Objective-C*.
- Objective-C Runtime Reference (https://developer.apple.com/documentation/objectivec/objective-c_runtime) — la API del tiempo de ejecución; leerla explica cómo funciona el despacho (clase 111).
- GNUstep (https://www.gnustep.org/) — para ejecutarlo fuera de Apple.
- **Aaron Hillegass, Mikey Ward**, *Objective-C Programming: The Big Nerd Ranch Guide* — la introducción de referencia.
- **Matt Galloway**, *Effective Objective-C 2.0*, Addison-Wesley — 52 elementos sobre el lenguaje y su tiempo de ejecución.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: C · Smalltalk · Swift · C++

🐫 OCaml — 1996

OCaml es la rama **pragmática** de la familia ML: funcional como Haskell, pero **evaluación ansiosa**, con mutación cuando hace falta y con un compilador rapidísimo. Es el lenguaje en el que están escritos varios de los verificadores formales más serios del mundo — y el que inspiró directamente a Rust y a F#.

> 🎯 ***Por qué está en este programa***
>
> *OCaml es un **primo de la familia funcional tipada (ML)** (Atlas), que no tiene representante en el núcleo.*
>
> *Aporta al programa **el sistema de módulos más potente de esta lista** —functores: módulos que reciben módulos (clase 149)— y la demostración de que **un lenguaje funcional puede ser rápido y pragmático** sin renunciar a la seguridad de tipos.*

Año	1996 (Objective Caml); Caml es de 1985; **OCaml 5** con efectos (2022)
Autoría	**INRIA** (Francia): Xavier Leroy, Damien Doligez, Didier Rémy y otros
Familia	Funcional tipada (ML); descendiente de ML (Milner, 1973) y de Caml
Paradigma	**Funcional, imperativo y orientado a objetos** — las tres cosas de verdad
Tipado	**Estático con inferencia total**; tipos algebraicos y módulos de primera clase
Memoria	Recolección de basura generacional, muy rápida en asignación
Ejecución	**Compilado a nativo** o a bytecode; compilación muy veloz
Estado	🟢 **Vivo y de nicho**: verificación formal, finanzas y herramientas

---

📜 Historia

ML —*Meta Language*— lo creó **Robin Milner** en Edimburgo en **1973**, no como lenguaje de programación sino como **el lenguaje para escribir tácticas de un demostrador de teoremas**. De ahí salieron dos ideas que hoy están en todas partes:

- **La inferencia de tipos Hindley-Milner**: tipado estático completo **sin escribir tipos**.
- **Los tipos algebraicos con emparejamiento de patrones** y comprobación de exhaustividad.

Milner recibió el Premio Turing en 1991, en buena parte por esto.

INRIA desarrolló **Caml** (1985) y después **Objective Caml (1996)**, que añadió objetos y un sistema de módulos muy elaborado. En **2011** pasó a llamarse simplemente **OCaml**.

Y su influencia es directa y reconocida:

- **Rust**: el primer compilador de Rust estaba escrito en OCaml, y su sistema de tipos —`Option`, `Result`, `match`, los rasgos— viene de aquí.
- **F#** es esencialmente OCaml sobre .NET.
- **Y Flow y Hack**, las herramientas de tipos de Meta para JavaScript y PHP, están escritas en OCaml.

OCaml 5 (2022) fue un cambio profundo: **paralelismo real con multinúcleo** —hasta entonces había un bloqueo global— y **manejadores de efectos**, una idea de investigación que permite implementar corrutinas, generadores y `async` **como bibliotecas** en lugar de como características del lenguaje (clase 134).

🏭 Dónde vive hoy

- **Verificación formal**: **Coq/Rocq** —el asistente de demostración con el que se verificó el compilador **CompCert**— está escrito en OCaml. También **Frama-C** y varias herramientas de análisis usadas en aviación (clase 164).
- **Finanzas**: **Jane Street** es el caso más conocido: cientos de personas escribiendo OCaml para negociación algorítmica, con su propia biblioteca estándar publicada.
- **Herramientas de desarrollo**: Flow, Hack, Infer (Meta), Semgrep, Unison.
- **Blockchain**: Tezos está implementado en OCaml.
- **Y compiladores en general**, que es su terreno natural.

🧠 Lo que enseña: módulos y functores

Es su aportación más distintiva y la que ningún otro lenguaje de esta lista tiene igual (clase 149):

```
module type ORDENABLE = sig
  type t
  val compare : t -> t -> int
end

module Conjunto (E : ORDENABLE) = struct     (* ← un MÓDULO que recibe un MÓDULO *)
  type elemento = E.t
  let vacio = []
  let rec insertar x = function
    | [] -> [x]
    | y :: r when E.compare x y < 0 -> x :: y :: r
    | y :: r -> y :: insertar x r
end

module ConjuntoEnteros = Conjunto (Int)      (* ← se instancia con otro módulo *)
```

Un *functor* es una función de módulos a módulos, comprobada por el sistema de tipos. Es **parametrización a nivel de arquitectura**, no de clase — la idea que la clase 166 pide y que en otros lenguajes se aproxima con inyección de dependencias, con genéricos o con interfaces.

Y merece la comparación con Ada, que tiene lo más parecido: **los genéricos de Ada con parámetros formales de subprograma y de paquete** (clase 151) son la misma familia de idea, con otra sintaxis.

Y la segunda cosa que OCaml hace muy bien es ser pragmático:

```
let contador = ref 0        (* mutable, EXPLÍCITAMENTE *)
contador := !contador + 1
let arreglo = [| 1; 2; 3 |]  (* arreglos mutables de verdad *)
```

La mutación existe y se ve. A diferencia de Haskell, no hay que envolverla en nada: está ahí, marcada con `ref` y `:=`, y se usa cuando conviene. Eso hace que **el rendimiento sea predecible** —evaluación ansiosa, sin cadenas de cálculos pendientes— y es la razón de que Jane Street pueda escribir sistemas de latencia baja en él.

🔄 Lo que se ha modernizado

- **OCaml 5**: **multinúcleo real** con dominios, y **manejadores de efectos** — con los que `async` / `await` deja de ser sintaxis y pasa a ser una biblioteca (clase 134).
- **Dune** como sistema de construcción, rápido y declarativo, y **opam** como gestor de paquetes con fichero de bloqueo (clase 143).
- **js_of_ocaml** y **Melange**: compilar a JavaScript (clase 162).
- **Bibliotecas estándar alternativas**: `Base` y `Core` de Jane Street, hoy muy usadas.
- **Y** `ppx`: metaprogramación mediante transformación del árbol sintáctico, con la que se generan serializadores y comparadores automáticamente (clases 122 y 159).

⚙️ Cómo se ejecuta hoy

```
ocaml main.ml < entrada.txt              # intérprete
ocamlfind ocamlopt -package str main.ml   # compilar a nativo

dune build && dune test                    # lo habitual (clases 143 y 147)
opam install <paquete>
```

🪠 El programa de la clase 041 en OCaml

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
let () =
  let linea = read_line () in
  match String.split_on_char ' ' linea |> List.map float_of_string with
  | [precio; cantidad; descuento] ->
      Printf.printf "Total: %.2f\n" (precio *. cantidad *. (1. -. descuento))
  | _ -> prerr_endline "Se esperaban tres valores"; exit 1
```

Lo que hay que ver.

- `*.` **y** `-.` **con punto** son los operadores de **coma flotante**: en OCaml, `*` es solo para enteros. **No hay sobrecarga de operadores**, así que cada tipo tiene los suyos. Es incómodo y elimina toda ambigüedad — la misma disciplina de Go y Rust, llevada más lejos (clase 100).
- **El** `match` **trata los dos casos**: la lista con tres elementos y **todo lo demás**. El compilador **avisaría si faltara un caso**, y por eso el programa maneja el error de entrada sin escribir un `if`.
- `|>` **es la canalización**, igual que en F# — de hecho, F# la heredó de aquí.
- `let () = ...` es el punto de entrada: se liga el resultado al patrón vacío, que es la forma idiomática de decir "esto se ejecuta por sus efectos".
- **Y no hay ni una anotación de tipo**, y el programa está completamente tipado: eso es Hindley-Milner.

📚 Fuentes y bibliografía

- Real World OCaml (https://dev.realworldocaml.org/) — **Minsky, Madhavapeddy y Hickey**; libre en línea, escrito desde Jane Street; el libro de referencia.
- ocaml.org/docs (https://ocaml.org/docs) — tutoriales oficiales y el manual del lenguaje.

- **Xavier Leroy et al.**, *The OCaml system: documentation and user's manual* — la referencia formal; el capítulo de módulos merece la pena.
- **Jason Hickey**, *Introduction to Objective Caml* — libre; muy bueno para el sistema de módulos.
- CompCert (https://compcert.org/) — el compilador de C **verificado formalmente**, escrito en Coq y OCaml; el mejor argumento existente sobre lo que la verificación puede lograr (clase 164).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Haskell · F# · Rust · Scala · Ada

📘 Pascal — 1970

El lenguaje diseñado para enseñar a programar bien. Niklaus Wirth lo escribió como una tesis sobre cómo debía ser un lenguaje: pequeño, ordenado, con tipos fuertes y sin trampas. Millones de programadores aprendieron con él, y su influencia está en la sintaxis de media docena de lenguajes posteriores que probablemente sí usas.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Pascal se ejecuta hoy**, principalmente a través de sus dos descendientes vivos: **Free Pascal** (compilador libre y multiplataforma, con el IDE **Lazarus**) y **Delphi** (comercial y en desarrollo activo). El Pascal ISO clásico es minoritario; la línea Object Pascal es un negocio en marcha.*
>
> *Entra porque **es el origen documentado de ideas que el núcleo da por sentadas y ya no explica**: el tipado fuerte con conversiones explícitas, los **tipos definidos por el usuario** (enumerados, subrangos, registros), y la separación limpia entre declaración y ejecución. Cuando en la Parte 3 se estudia por qué `:=` no es `=`, o por qué un `enum` de Rust es distinto de un entero, se está estudiando una decisión que Wirth tomó en 1970 y argumentó por escrito. Leer Pascal es leer el razonamiento original.*

Año	1970; estándar **ISO 7185** en 1983; **Extended Pascal** ISO 10206 en 1990
Autoría	**Niklaus Wirth**, ETH de Zúrich
Familia	ALGOL — descendiente directo de **ALGOL W**, también de Wirth
Paradigma	Imperativo, procedimental y **estructurado**
Tipado	**Estático y fuerte**, con subrangos y enumerados comprobados
Memoria	Pila y montículo explícito (`New` / `Dispose`)
Ejecución	Compilado a nativo; históricamente a **P-code** para portabilidad
Estado	🟡 **El Pascal clásico es minoritario; su rama Object Pascal está muy viva**

📜 Historia

Niklaus Wirth participó en el comité de ALGOL 68 y se marchó en desacuerdo: consideraba que el lenguaje se estaba volviendo demasiado grande y complejo para lo que aportaba. Su propuesta rechazada, **ALGOL W**, se convirtió en la semilla de lo que en 1970 publicó como **Pascal**, en honor a Blaise Pascal.

El objetivo era doble y explícito: **enseñar programación estructurada** y demostrar que un lenguaje podía ser potente siendo pequeño. Wirth dedicó su carrera a esa tesis y la resumió en una frase que sigue citándose: *"hacer las cosas lo más simples posible, pero no más simples"*. Escribió también la **Ley de Wirth** —"el

software se vuelve más lento más deprisa de lo que el hardware se vuelve más rápido"— que ha envejecido de forma incómodamente buena.

Pascal introdujo o consolidó cosas que hoy son evidentes y entonces no lo eran:

- **Tipos definidos por el usuario**: enumerados (`type Color = (Rojo, Verde, Azul)`), **subrangos** (`type Nota = 1..7`) y registros. Un enumerado de Pascal **no es un entero**: no se le puede sumar, y esa restricción es el punto.
- **Punteros con tipo**, en lugar de direcciones sin más.
- **Sintaxis de declaración separada** de la ejecución, con `var`, `const` y `type` en secciones propias.
- `:=` **para asignar y** `=` **para comparar**, distinción que Ada, Go (`:=`) y muchos otros heredaron precisamente para evitar el clásico `if (x = 5)` de C.

El truco de la portabilidad. El compilador de Zúrich no generaba código máquina: generaba **P-code**, un bytecode para una máquina virtual imaginaria. Para llevar Pascal a un ordenador nuevo bastaba con escribir un intérprete de P-code. **UCSD Pascal** llevó esa idea al mercado en los 70 y consiguió que el mismo software corriera en máquinas incompatibles. Es exactamente el modelo que Java popularizó veinte años después con la JVM y el bytecode.

En 1983 **Borland** lanzó **Turbo Pascal**: un compilador y un editor en un solo programa, que cabía en un disquete, compilaba en segundos y costaba 49,95 dólares cuando la competencia pedía cientos. Fue un fenómeno, y su descendiente directo es Delphi.

Wirth siguió adelante con **Modula-2** (1978, con módulos de verdad) y **Oberon** (1986, aún más pequeño). Ninguno alcanzó la difusión de Pascal, pero Modula-2 influyó en el sistema de módulos de casi todo lo que vino después. Wirth murió en enero de 2024.

🏭 Dónde sobrevive hoy

- **A través de Delphi y Free Pascal/Lazarus**: aplicaciones de escritorio, ERP, TPV, software industrial. Ahí está el grueso del Pascal en producción.
- **Educación**: sigue usándose en enseñanza secundaria y primeros cursos universitarios en varios países, precisamente porque fue diseñado para eso.
- **Competición de programación**: fue durante décadas uno de los lenguajes admitidos en olimpiadas de informática, y aún hay comunidades que lo usan por la velocidad de compilación.
- **Aplicaciones heredadas** de los 80 y 90 en administración, laboratorios e industria.

🧠 Por qué no ha muerto

1. Su descendiente es un producto comercial vivo. El Pascal ISO puro es una pieza de museo, pero Object Pascal no. Ver Delphi.

2. Compila absurdamente rápido. El diseño del lenguaje —una sola pasada, declaraciones antes del uso, sin preprocesador— permite compiladores que procesan cientos de miles de líneas por segundo. Free Pascal y Delphi siguen siendo de los compiladores más rápidos que existen, y quien viene de esperar minutos con C++ o Rust lo nota de inmediato.

3. Es genuinamente bueno para enseñar. Fuerza a declarar antes de usar, distingue asignación de comparación, no permite conversiones silenciosas y su sintaxis se lee en voz alta. Los errores que un principiante comete en C —confundir `=` con `==`, desbordar un array, olvidar un tipo— en Pascal son errores de compilación.

4. Free Pascal es un compilador serio. Multiplataforma real (Windows, Linux, macOS, Android, Raspberry Pi, incluso microcontroladores), compatible con la sintaxis de Delphi, libre y mantenido.

🔄 Lo que se ha modernizado

El Pascal ISO de 1983 sí está congelado. **Free Pascal, que es donde vive el lenguaje, no**:

- **Genéricos, sobrecarga de operadores, métodos anónimos (*closures*), interfaces, ayudantes de tipo y** `for..in` — todo lo que se espera de un lenguaje actual.
- **Multiplataforma real**: x86, x86-64, ARM, **AArch64**, **RISC-V**, MIPS, PowerPC; Windows, Linux, macOS (incluida Apple Silicon), FreeBSD, **Android**, y hasta microcontroladores AVR y ARM Cortex-M sin sistema operativo.
- **WebAssembly**: FPC tiene un generador de código para **wasm**, así que el mismo Pascal puede ejecutarse en el navegador.
- **Compatibilidad con el dialecto de Delphi** (`-Mdelphi`), suficiente para compilar proyectos comerciales completos con un compilador libre.
- **Lazarus** ofrece el diseñador visual, depurador integrado y un gestor de paquetes en línea.
- **Velocidad de compilación** que sigue siendo de las mejores del mercado — un rasgo del diseño original de Wirth que hoy vuelve a valorarse.

⚙️ Cómo se ejecuta hoy

```
sudo apt-get install -y fpc

fpc -Mobjfpc total_venta.pas
echo "15000 2 0.10" | ./total_venta
# Total: 27000.00
```

Implementaciones: Free Pascal (FPC) es la libre y la más completa; **Lazarus** es su IDE con diseñador visual de formularios, clon libre del de Delphi. **Delphi** de Embarcadero es la comercial. **GNU Pascal** existe pero lleva años sin desarrollo activo.

Modos de compilación de FPC, que hay que conocer porque cambian el lenguaje: `-Mfpc` (dialecto propio), `-Mobjfpc` (Object Pascal con extensiones propias, el recomendado), `-Mdelphi` (compatibilidad con Delphi), `-Miso` (ISO 7185 estricto).

🧪 El programa de la clase 041 en Pascal

```
program TotalVenta;
{$MODE OBJFPC}{$H+}

var
  Precio, Cantidad, Descuento, Total: Double;

begin
  Read(Precio, Cantidad, Descuento);

  Total := Precio * Cantidad * (1 - Descuento);

  WriteLn('Total: ', Total:0:2);
end.
```

Recorrido, línea a línea.

- `program TotalVenta;` nombra el programa. En el Pascal ISO era obligatorio y llevaba además la lista de ficheros externos (`program X(input, output);`); FPC lo acepta sin ella.
- `{$MODE OBJFPC}` es una **directiva de compilador** —van entre llaves con `$`, y sí, ocupan el mismo espacio sintáctico que los comentarios—. Selecciona el dialecto. `{$H+}` hace que `string` sea de longitud dinámica en lugar del `ShortString` de 255 caracteres heredado de Turbo Pascal.
- **La sección `var` va antes del cuerpo, y eso no es negociable.** Pascal exige declarar todo antes de usarlo, en su sitio. Es la restricción que más molesta a quien viene de lenguajes modernos y es, literalmente, la razón por la que el compilador puede trabajar en una sola pasada y ser tan rápido.
- `begin` … `end.` delimita el bloque principal. El **punto final** tras el último `end` marca el fin del programa; los `end` internos llevan punto y coma. Olvidarlo es el error de sintaxis clásico.
- `:=` asigna. `=` compara. La separación es deliberada y es una de las mejores decisiones de diseño del lenguaje: en Pascal es **imposible** escribir por accidente el bug de `if (x = 5)` de C.
- `Read(Precio, Cantidad, Descuento)` lee tres reales separados por espacios o saltos de línea. La E/S está **en el lenguaje**, no en una biblioteca: `Read`, `ReadLn`, `Write` y `WriteLn` aceptan un número variable de argumentos de tipos distintos, algo que ninguna función normal de Pascal puede hacer. Son casos especiales del compilador.
- **`Total:0:2` es el detalle que hay que llevarse.** En un `Write`, la sintaxis `valor:ancho:decimales` formatea directamente: ancho mínimo 0 (sin relleno), 2 decimales. Es formateo integrado en la sintaxis, sin cadena de plantilla, sin `printf` y **sin depender de la configuración regional**. Para el caso `0.0` produce `0.00`, que es justo lo que otros lenguajes complican.

Y ahora el Pascal que merece la pena ver, el de los tipos:

```
type
  TDia       = (Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo);
  TLaborable = Lunes..Viernes;              { subrango: un tipo con límites }
  TVenta = record
    Producto : string[40];
    Unidades : 1..9999;                     { el rango es parte del tipo }
    Importe  : Double;
  end;

var
  Hoy   : TDia;
  Venta : TVenta;
begin
  Hoy := Miercoles;
  { Hoy := Hoy + 1;   <-- no compila: un enumerado no es un entero }
  Hoy := Succ(Hoy);   { esta es la forma correcta }

  Venta.Unidades := 0;  { con {$R+} activo, esto es un error en ejecución }
end.
```

Un enumerado de Pascal **no** es un entero disfrazado: no se le puede sumar, y `Succ` / `Pred` son las operaciones legítimas. Un subrango `1..9999` es un tipo cuyo dominio comprueba el compilador cuando puede y el runtime cuando no (con `{$R+}`). Compara con `enum` de C, que sí es un entero y admite cualquier valor. Esa diferencia —el tipo como afirmación sobre los valores posibles, no solo sobre el tamaño en bytes— es la línea recta que va de Pascal a Ada, y de ahí a los tipos suma de Rust y a los tipos literales de TypeScript.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En Pascal es…
`int x = 5;`	`var x: Integer;` … `x := 5;`
`==`	`=` ; y la asignación es `:=`
`!=`	`<>`
`&&` / `||` / `!`	`and` / `or` / `not`
`{ ... }`	`begin ... end`
`// comentario`	`{ comentario }` o `(* comentario *)` ; FPC acepta `//`
`struct`	`record`
`enum`	`type T = (A, B, C);` — y **no** es un entero
`typedef unsigned char`	`type TNota = 1..7;` — subrango comprobado
`void f()` / `int f()`	`procedure f;` / `function f: Integer;`
`return x`	`Result := x;` (Object Pascal) o `NombreFuncion := x;` (clásico)
`malloc` / `free`	`New(p)` / `Dispose(p)`
`printf("%.2f", x)`	`Write(x:0:2)`

⚠️ Errores comunes al leerlo

- **El punto final.** `end.` cierra el programa; `end;` cierra un bloque. Confundirlos es el primer error de todo el mundo.
- **Punto y coma antes de `else`.** `if c then a; else b;` no compila: el `;` **termina** la sentencia `if` y deja el `else` huérfano. Es la trampa sintáctica más famosa del lenguaje.
- **Declarar en medio del código.** No se puede. Toda variable va en la sección `var` del bloque.
- **Suponer evaluación en cortocircuito.** Depende del modo del compilador (`{$B-}` la activa, que es lo habitual, pero el ISO no la garantiza). Conviene no depender de ella al leer código antiguo.
- **Confundir `ShortString` con `string`.** Sin `{$H+}`, `string` es de 255 caracteres como máximo, herencia de Turbo Pascal. Con `{$H+}`, es dinámico con conteo de referencias.
- **Índices arbitrarios.** `array[1..10]` empieza en 1, pero `array[-5..5]` y `array['a'..'z']` son perfectamente legales. El array lleva sus límites en el tipo.

📚 Fuentes y bibliografía

- Free Pascal (https://www.freepascal.org/) — compilador libre, con la referencia del lenguaje y de la biblioteca en línea.
- Lazarus IDE (https://www.lazarus-ide.org/) — el entorno visual libre.
- Free Pascal — Reference Guide (https://www.freepascal.org/docs.html) — la documentación que usarás a diario.
- **Niklaus Wirth**, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976 — uno de los libros fundacionales de la disciplina; el título es una tesis y el lenguaje de los ejemplos es Pascal. Sigue siendo excelente.
- **Niklaus Wirth**, *The Programming Language Pascal*, Acta Informatica, 1971 — el artículo original; merece la pena leer el razonamiento de diseño de primera mano.
- **Kathleen Jensen, Niklaus Wirth**, *Pascal User Manual and Report* — el documento de referencia clásico.
- **Marco Cantù**, *Object Pascal Handbook* — para el salto al Pascal moderno; ver Delphi.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Delphi / Object Pascal · Ada · C

🐪 Perl — 1987

La cinta adhesiva de Internet. Durante los años 90, buena parte de la web dinámica del planeta funcionaba con guiones CGI en Perl. Esa época pasó, pero Perl no: sigue procesando texto, administrando sistemas y secuenciando genomas, con una nueva versión estable cada año.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Perl 5 se mantiene activamente y publica una versión estable anual**; la documentación vigente corresponde a la serie 5.44. Viene instalado en prácticamente cualquier sistema Unix o Linux, sostiene herramientas que se usan a diario (`git send-email`, Bugzilla, cPanel, buena parte de la infraestructura de Debian) y sigue siendo dominante en bioinformática.*
>
> *Entra porque **muestra dos conceptos que el núcleo no expresa bien**. El primero: las **expresiones regulares como parte de la sintaxis del lenguaje**, no como una biblioteca. Perl las integró tan a fondo que definió el estándar de facto —**PCRE**, Perl Compatible Regular Expressions— que hoy usan Python, JavaScript, PHP, Java, Go y casi todos los demás. Cuando escribes una regex en cualquier lenguaje, estás escribiendo Perl. El segundo: el **contexto**, un concepto que casi no existe fuera de aquí: en Perl **la misma expresión devuelve cosas distintas según dónde se use**, y entenderlo enseña a mirar con otros ojos la evaluación de expresiones.*
>
> *Perl ya aparece resuelto en cada `primos.md` del programa y **se ejecuta en CI**; esta ficha es su historia y su porqué.*

Año	18 de diciembre de 1987 (Perl 1.0); **Perl 5** desde 1994
Autoría	**Larry Wall** — lingüista de formación, y se nota en el diseño
Familia	Scripting dinámico; influencias de C, `sed`, `awk`, shell y Lisp
Paradigma	Multiparadigma: procedimental, funcional y OO
Tipado	**Dinámico**, con *sigilos* que marcan la forma del dato (`$`, `@`, `%`)
Memoria	Conteo de referencias
Ejecución	Compilado a un árbol interno en cada arranque y ejecutado
Estado	🟡 **Mantenido y en uso** — administración, texto, bioinformática, sistemas heredados

---

📜 Historia

En 1987 **Larry Wall** trabajaba como administrador de sistemas y programador en un proyecto que necesitaba generar informes a partir de árboles de ficheros de texto. Las herramientas disponibles no le servían: `awk` no manejaba bien ficheros ni estructuras complejas, `sed` era demasiado limitado, el shell era frágil y C era demasiado ceremonioso para un guion. Escribió **Perl** para cubrir ese hueco concreto, y lo publicó en el grupo de noticias `comp.sources.unix` el 18 de diciembre de 1987.

Wall es **lingüista** de formación, y eso explica la filosofía del lenguaje mejor que cualquier decisión técnica. Sus dos lemas son deliberadamente contrarios a la corriente:

- **TMTOWTDI** — *There's More Than One Way To Do It*. Frente al "debería haber una forma obvia" de Python, Perl asume que los lenguajes naturales tienen sinónimos y registros, y que quien escribe elige el que mejor comunica su intención.
- **"Perl facilita lo fácil y hace posible lo difícil."**

Perl 5, en 1994, fue la reescritura que lo convirtió en un lenguaje serio: referencias (y con ellas estructuras de datos anidadas), módulos, paquetes y orientación a objetos mediante `bless`. Un año después nació **CPAN**, el *Comprehensive Perl Archive Network*, que fue **el primer gran repositorio de módulos de la historia** — el antepasado directo de PyPI, npm, RubyGems y crates.io.

A finales de los 90, Perl era el lenguaje de la web: los guiones CGI en `cgi-bin` eran Perl casi por definición. Ese dominio se perdió frente a PHP, y luego frente a Python y Ruby.

El episodio Perl 6. Anunciado en el año 2000 como el rediseño total del lenguaje, se convirtió en uno de los desarrollos más largos de la historia del software. Durante quince años la comunidad quedó dividida entre un Perl 5 que "iba a ser sustituido" y un Perl 6 que no llegaba. Cuando por fin se estabilizó, era tan distinto que en **2019 se renombró como Raku** (https://raku.org/) y se declaró un lenguaje aparte. Perl 5 recuperó su nombre y su cadencia anual de versiones. La lección —el coste de anunciar un sucesor que no llega— es de las más citadas en gestión de proyectos de lenguajes, y conviene tenerla presente al estudiar cualquier migración mayor.

🏭 Dónde sobrevive hoy

- **Administración de sistemas Unix/Linux**: viene preinstalado, siempre está ahí, y procesa registros y configuraciones sin instalar nada.
- **Bioinformática**: **BioPerl** fue durante años la biblioteca de referencia para manipular secuencias genómicas, y sigue habiendo mucha canalización de datos científica en Perl.
- **Infraestructura conocida**: `git send-email` y varias utilidades de Git son guiones Perl; **Bugzilla**, **cPanel/WHM** (el panel de hosting más extendido) y el sistema de gestión de paquetes de Debian dependen de él.
- **Empresas de gran escala** con base histórica en Perl, siendo **Booking.com** el caso más documentado públicamente.
- **Procesamiento de texto y ETL** en general: sigue siendo excepcionalmente bueno en ello.

🧠 Por qué no ha muerto

1. Las regex son parte del lenguaje. `if ($linea =~ /^(\w+)\s+(\d+)$/)` no llama a una función: `=~` es un operador, `/.../` es literal de patrón, y `$1`, `$2` aparecen automáticamente. En Python o Java hay que importar un módulo, compilar el patrón y consultar un objeto de coincidencia. Para trabajo de texto intenso, la diferencia de densidad es enorme.

2. CPAN sigue siendo excepcional. Decenas de miles de distribuciones, con una cultura de pruebas inusualmente rigurosa: **CPAN Testers** ejecuta automáticamente la batería de pruebas de cada módulo en decenas de combinaciones de sistema operativo y versión de Perl, y publica los resultados. Pocos ecosistemas tienen algo comparable.

3. Compatibilidad hacia atrás casi obsesiva. Un guion de 1998 sigue funcionando. Cuando el lenguaje ha querido cambiar algo, lo ha hecho tras `use feature` o `use v5.36`, sin romper lo existente. Perl no tuvo su ruptura 2→3.

4. Ubicuidad. Está en cualquier sistema tipo Unix. Para un administrador que no puede instalar nada en una máquina ajena, eso lo convierte en la única opción.

🔄 Lo que se ha modernizado

El Perl que se escribe hoy no es el de los guiones CGI de 1998:

- **Firmas de subrutina** estables desde 5.36: `sub total ($precio, $cantidad, $descuento) { ... }`, en lugar del clásico `my (...) = @_;`.
- `try` / `catch` / `finally` como sintaxis del lenguaje (5.34–5.40), en vez del idioma `eval { }; if ($@)` que era fácil de escribir mal.
- **Clases nativas.** La palabra clave `class` con `field` y `method` (el proyecto *Corinna*) está entrando en el lenguaje desde 5.38 como característica experimental: orientación a objetos de verdad, en lugar del `bless` sobre un hash.
- `use v5.40` activa de golpe el conjunto de características modernas — `say`, firmas, `try/catch`, `strict`, `warnings` — con una sola línea.
- **Una versión estable al año**, con la serie 5.44 como documentación vigente, y compatibilidad hacia atrás mantenida a rajatabla.
- **Ecosistema actual**: **Carton** y **cpanfile** para dependencias fijadas por proyecto, **Perl::Critic** para análisis estático, **Mojolicious** y **Dancer2** como marcos web modernos con soporte asíncrono y WebSockets, y **PDL** para cálculo numérico vectorizado.

⚙️ Cómo se ejecuta hoy

```
perl --version          # ya está instalado en Linux y macOS

perl total.pl < entrada.txt
echo "15000 2 0.10" | perl total.pl
# Total: 27000.00

# Como una sola línea, que es donde Perl brilla:
perl -ne 'print if /ERROR/' registro.log
perl -pi -e 's/viejo/nuevo/g' *.conf
```

Ecosistema: `cpanm` (`cpanminus`) para instalar módulos, **Carton** para fijar versiones por proyecto (equivalente a un *lockfile*), **perlbrew** o **plenv** para gestionar varias versiones del intérprete, y **Perl::Critic** y **perltidy** para estilo y análisis estático. La documentación completa está en la propia máquina: `perldoc perlre`, `perldoc -f split`.

🧪 El programa de la clase 041 en Perl

```
use strict;
use warnings;

my $linea = <STDIN>;
chomp $linea;

my ($precio, $cantidad, $descuento) = split ' ', $linea;
my $total = $precio * $cantidad * (1 - $descuento);

printf "Total: %.2f\n", $total;
```

Recorrido, línea a línea.

- `use strict; use warnings;` son las **dos primeras líneas de cualquier Perl escrito después de 1995**, y su ausencia es la mejor señal de que estás ante código antiguo o descuidado. `strict` obliga a declarar las variables (sin él, un nombre mal tecleado crea una variable global nueva en silencio) y `warnings` avisa de usos sospechosos.

- `my` declara una variable **de ámbito léxico**, el equivalente a `let` de JavaScript.
- **Los *sigilos*** son la marca de la casa: `$` para un **escalar** (un valor), `@` para una **lista**, `%` para un **hash**. El sigilo no describe el tipo del dato sino la **forma en que lo estás usando**, que es una idea distinta y desconcertante al principio: `$lista[0]` lleva `$` —no `@`— porque un elemento suelto de la lista *es* un escalar.
- `<STDIN>` lee del canal de entrada. Y aquí aparece **el contexto**, el concepto más específicamente perlero: en **contexto escalar** (`my $linea = <STDIN>`) devuelve **una línea**; en **contexto de lista** (`my @todo = <STDIN>`) devuelve **todas las líneas**. La misma expresión, dos resultados, según lo que espere el lado izquierdo. Lo mismo ocurre con un array: `my $n = @lista` da el número de elementos, porque un array en contexto escalar es su longitud.
- `chomp` elimina el salto de línea final. **Es obligatorio y se olvida siempre.** En este programa el resultado saldría bien igualmente porque el `\n` desaparece al convertir a número, pero en cuanto se compara texto —`if ($palabra eq "Ada")`— la cadena mide un carácter de más y la comparación falla sin dar ningún error. Es exactamente el defecto que se encontró al ejecutar por primera vez los primos de este repositorio en CI.
- `split ' ', $linea` usa el **patrón mágico de un solo espacio**: no significa "parte por el carácter espacio", significa "parte por rachas de espacios en blanco e ignora los iniciales". Es un caso especial documentado, heredado de `awk`, y es lo que hace que funcione con entradas mal alineadas.
- La asignación a `my ($a, $b, $c) = ...` es **desestructuración**, treinta años antes de que JavaScript la incorporara.
- `printf "%.2f"` es el mismo formato de C, porque Perl viene de ahí.

Y ahora lo que hace de Perl lo que es. El procesamiento de texto:

```
while (my $linea = <STDIN>) {
    next unless $linea =~ /^(\d{4})-(\d{2})-(\d{2})\s+(\w+)\s+(\d+)$/;
    my ($anio, $mes, $dia, $producto, $unidades) = ($1, $2, $3, $4, $5);
    $ventas{$producto} += $unidades;
}
printf "%-15s %6d\n", $_, $ventas{$_} for sort keys %ventas;
```

Ocho líneas que analizan un registro con formato, validan cada fila contra un patrón, agregan por clave y emiten un informe ordenado. `next unless` es un modificador de sentencia —la condición va detrás, porque así se lee mejor en voz alta, decisión típicamente lingüística de Wall—, `$1..$5` son las capturas del patrón, `%ventas` es un hash que se crea solo al usarlo, y `for sort keys %ventas` al final es otro modificador. Reproducir esto en cualquier lenguaje del núcleo cuesta el doble de líneas. **Ese es el nicho donde Perl sigue siendo la mejor herramienta.**

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces...	En Perl es...
`x = 5`	`my $x = 5;`
Lista / array	`my @a = (1, 2, 3);` y el elemento es `$a[0]`
Diccionario	`my %h = (clave => 'valor');` y el elemento es `$h{clave}`
`len(a)`	`scalar @a` — o simplemente `@a` en contexto escalar
`re.match(p, s)`	`$s =~ /p/` — operador, no función
`s.replace(a, b)`	`$s =~ s/a/b/g;`
`a.split(",")`	`split /,/, $a`
`"".join(a)`	`join '', @a`
`def f(a, b):`	`sub f { my ($a, $b) = @_; ... }` — los argumentos llegan en `@_`
`try / except`	`eval { ... }; if ($@) { ... }` — o `Try::Tiny` de CPAN
`import modulo`	`use Modulo;`
Referencia / puntero	`\@lista`, `\%hash`, y se accede con `->`

⚠️ Errores comunes al leerlo

- **Olvidar `chomp`.** El defecto silencioso número uno.
- **No poner `use strict; use warnings;`.** Sin ellos, `$totl` es una variable global nueva y válida.
- **Confundir los comparadores.** `==` compara **números**, `eq` compara **cadenas**. `"10" == "10.0"` es cierto; `"10" eq "10.0"` es falso. Usar el equivocado es un error clásico y difícil de ver.
- **Ignorar el contexto.** `my $n = @lista;` da la longitud, no el primer elemento. Es correcto y desconcertante hasta que se interioriza.
- **Leer los sigilos como tipos.** `$a[0]` con `$` es correcto: describe la forma del **acceso**, no la del contenedor.
- **Creer que todo Perl es ilegible.** El Perl de golf y las líneas sueltas crípticas dieron fama al lenguaje, pero un Perl moderno con `use v5.36`, funciones con firma y módulos de CPAN es perfectamente ordenado.

📚 Fuentes y bibliografía

- perl.org (https://www.perl.org/) y perldoc.perl.org (https://perldoc.perl.org/) — sitio oficial y documentación completa, también disponible sin conexión con `perldoc`.
- MetaCPAN (https://metacpan.org/) — el buscador moderno de CPAN.
- CPAN Testers (http://www.cpantesters.org/) — los resultados de ejecutar las pruebas de cada módulo en decenas de plataformas; vale la pena verlo aunque no uses Perl.
- **Larry Wall, Tom Christiansen, Jon Orwant**, *Programming Perl*, O'Reilly — "el libro del camello", escrito por el autor del lenguaje; la referencia canónica.
- **chromatic**, *Modern Perl* — gratis en línea (http://modernperlbooks.com/); cómo escribir Perl hoy y no como en 1998. Es el libro por el que empezar.
- **Mark Jason Dominus**, *Higher-Order Perl* — gratis del autor (https://hop.perl.plover.com/); programación funcional avanzada, y uno de los mejores libros de programación de su década.
- **Jeffrey Friedl**, *Mastering Regular Expressions*, O'Reilly — la obra definitiva sobre expresiones regulares; útil sea cual sea tu lenguaje.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Tcl/Tk · Common Lisp · C

🐘 PHP — 1995

PHP no se diseñó: **creció**. Empezó como unos guiones en C para contar visitas a una página personal y acabó ejecutando la mayor parte de la web. Esa historia explica sus rarezas — y también por qué, después de una década de mala fama merecida, **PHP 8 es un lenguaje razonable que casi nadie ha vuelto a mirar**.

> 🎯 ***Por qué está en este programa***
>
> ***PHP es uno de los diez lenguajes del núcleo**, y comparte con Python la representación del **scripting dinámico** (Atlas).*
>
> *Está en el núcleo por una razón que no es sentimental: **es el lenguaje del que depende una parte desproporcionada de la web pública**, y quien vaya a mantener software real se lo encontrará. Y aporta al programa el caso de estudio más claro de **coerción débil de tipos** y de **cómo un ecosistema entero corrige un lenguaje desde fuera** (clases 100, 143 y 146).*

Año	1995; **PHP 5** con OO real (2004); **PHP 7** (2015); **PHP 8** (2020), anual
Autoría	**Rasmus Lerdorf**; el motor Zend, de **Andi Gutmans** y **Zeev Suraski**
Familia	Scripting dinámico; sintaxis de C y Perl
Paradigma	Imperativo y orientado a objetos; con rasgos y funciones de primera clase
Tipado	**Dinámico y débil**; con **declaraciones de tipo comprobadas** desde PHP 7
Memoria	Conteo de referencias + recolector de ciclos; **liberación al final de la petición**
Ejecución	Bytecode con caché (OPcache); **JIT** desde PHP 8
Estado	🟢 **Enorme cuota de la web**; muy mejorado y poco reconocido

---

📜 Historia

En **1994**, Rasmus Lerdorf escribió unos guiones en C para contar quién visitaba su currículum en línea. Los llamó **Personal Home Page Tools**. No pretendían ser un lenguaje.

La gente empezó a pedírselos, y en **1995** los publicó. En **1997**, dos estudiantes israelíes —**Andi Gutmans** y **Zeev Suraski**— reescribieron el analizador entero: nació el **motor Zend** y el nombre pasó a ser el acrónimo recursivo **PHP: Hypertext Preprocessor**.

Y ahí está el origen de casi todo lo que se le critica: **la biblioteca estándar creció por acumulación**, sin plan. De ahí la inconsistencia famosa — `strlen` pero `str_replace`, `in_array($aguja, $pajar)` pero `strpos($pajar, $aguja)` — que es real y que hoy no se puede arreglar sin romper media web.

Lo que sí se arregló:

- **PHP 5 (2004)**: un modelo de objetos de verdad, con excepciones e interfaces.
- **PHP 5.3 (2009)**: espacios de nombres y cierres — lo que hizo posible **Composer** (2012) y con él un ecosistema moderno de dependencias (clase 143).
- **PHP 7 (2015)**: el motor reescrito. **Duplicó el rendimiento y redujo a la mitad la memoria**, un salto que muy pocos lenguajes han dado. Y llegaron las **declaraciones de tipo escalares**.
- **PHP 8 (2020)**: **JIT**, atributos, `match`, promoción de propiedades del constructor, tipos unión, argumentos con nombre, `nullsafe`.

- **PHP 8.1-8.4**: enumerados, `readonly`, fibras (clase 134), propiedades con captadores.

🏭 Dónde vive hoy

- **WordPress**: mueve por sí solo una fracción enorme de todos los sitios web del mundo.
- **Comercio electrónico**: Magento, PrestaShop, WooCommerce.
- **Aplicaciones de empresa a medida**: **Laravel** y **Symfony** son marcos de primer nivel, con una calidad que sorprende a quien juzga PHP por su reputación de 2005.
- **Wikipedia**: MediaWiki está escrito en PHP (y sus plantillas ejecutan Lua, clase 163).
- **Sistemas de gestión de contenidos y foros**: Drupal, Joomla, Nextcloud.

🧠 Lo que enseña: el modelo de ejecución sin estado

PHP tiene un modelo que ningún otro del núcleo comparte y que explica la mitad de sus decisiones: **"nada compartido"**.

```
Llega una petición HTTP
  → se arranca un intérprete (o se reutiliza uno limpio)
  → se ejecuta el guion
  → se envía la respuesta
  → y SE DESTRUYE TODO el estado
```

Y esa propiedad, que parece un derroche, es exactamente la primera regla del cierre de la clase 168: un servicio que no guarda estado en el proceso escala, se reinicia sin que nadie lo note y no filtra datos de una petición a la siguiente.

Es lo contrario de `mod_perl` (clase 163) y de un servidor Java de larga vida — y es la razón por la que **PHP casi nunca sufre las fugas de estado entre usuarios** que aquellos tuvieron.

El coste es que no hay caché en proceso ni conexiones persistentes fáciles, y por eso el ecosistema depende de OPcache, Redis y agrupadores de conexiones.

Y el otro tema que PHP enseña mejor que nadie es **la coerción débil** (clase 100):

```
0 == "abc"        // false en PHP 8; TRUE hasta PHP 7  ← ¡lo arreglaron!
"1" == "01"        // true
"10" == "1e1"       // true
100 == "1e2"         // true
```

> ***La línea corregida merece destacarse**, porque es un ejemplo excelente de la clase 175: **PHP 8 cambió la comparación entre número y cadena no numérica**, rompiendo compatibilidad deliberadamente, porque **el comportamiento anterior causaba fallos de seguridad reales** —una comparación de contraseñas con `==` podía dar verdadero—. Fue una decisión difícil, documentada y correcta. Y la regla práctica sigue siendo la de la clase 146: **usar siempre** `===`.*

🔄 Lo que se ha modernizado

- **Tipos por todas partes**: parámetros, retornos, propiedades, uniones, intersecciones, `never`. Con `declare(strict_types=1)`, PHP se comporta como un lenguaje de tipado fuerte.
- **Enumerados** (8.1) con métodos, y `readonly` (8.1/8.2) para inmutabilidad (clase 102).
- **Fibras** (8.1): corrutinas en el lenguaje, base de los tiempos de ejecución asíncronos (clase 134).
- **JIT** (8.0) — decisivo en cálculo, poco relevante en web, donde el cuello es la base de datos.

- **Composer y PSR**: gestor de dependencias con fichero de bloqueo (clase 143) y estándares de interoperabilidad entre marcos — el ecosistema puso el orden que el lenguaje no tenía.
- **Herramientas de análisis excelentes**: **PHPStan** y **Psalm** hacen análisis estático por niveles y detectan lo que el tipado dinámico deja pasar (clase 146).

⚙️ Cómo se ejecuta hoy

```
php main.php < entrada.txt        # el comando de la clase 041
php -S localhost:8000              # servidor de desarrollo, integrado

composer install                    # dependencias, con composer.lock (clase 143)
vendor/bin/phpstan analyse -l 9     # análisis estático estricto (clase 146)
vendor/bin/phpunit                  # pruebas (clase 139)
```

🍼 El programa de la clase 041 en PHP

```
<?php
// PHP: dinámico y débilmente tipado; las variables llevan el prefijo $.
$linea = trim(fgets(STDIN));
[$precio, $cantidad, $descuento] = preg_split('/\s+/', $linea);

$precioUnitario = (float) $precio;
$cantidadInt = (int) $cantidad;
$descuentoFloat = (float) $descuento;

$subtotal = $precioUnitario * $cantidadInt;
$total = $subtotal * (1 - $descuentoFloat);

printf("Total: %.2f\n", $total);
```

Lo que hay que ver.

- **El `$` delante de cada variable** viene de Perl, y con él la sintaxis general del lenguaje. Es la marca visual de la familia.
- **Las conversiones explícitas `(float)` y `(int)` no son necesarias** —PHP convertiría solo— y están ahí a propósito: **hacen visible lo que el lenguaje haría en silencio**, que es exactamente la disciplina que la clase 146 recomienda en un lenguaje con coerción débil.
- **La desestructuración `[$a, $b, $c] = ...`** es de PHP 7.1; antes se escribía `list(...)`.
- **`printf` con `%.2f`** es la misma familia que C y Perl: PHP heredó la función y su formato.
- **Y con `declare(strict_types=1)` al principio**, este programa **fallaría** si algún argumento no fuera del tipo declarado — que es la forma moderna de escribirlo.

📚 Fuentes y bibliografía

- Manual de PHP (https://www.php.net/manual/es/) — está en español y es sorprendentemente bueno; los comentarios de usuario, con precaución.
- PHP: The Right Way (https://phptherightway.com/) — la guía comunitaria de buenas prácticas, y el antídoto contra los tutoriales de 2008 que siguen circulando.
- PHP-FIG y los PSR (https://www.php-fig.org/psr/) — los estándares de interoperabilidad del ecosistema (clase 160).
- **Josh Lockhart**, *Modern PHP*, O'Reilly — el libro que separa el PHP actual del de su reputación.
- **Matthias Noback**, *Object Design Style Guide* y *Advanced Web Application Architecture* — arquitectura seria en PHP, aplicable a la Parte 9 del curso.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Python · Perl · Ruby · JavaScript

🧩 PL/I — 1964

El lenguaje que quiso ser todos los lenguajes. IBM lo diseñó para unificar el mundo científico de Fortran y el mundo empresarial de COBOL en una sola herramienta. No lo consiguió, pero por el camino inventó buena parte de lo que hoy damos por sentado.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: PL/I sigue en producción sobre z/OS.*** *IBM mantiene y vende Enterprise PL/I for z/OS, integrado con CICS, Db2 e IMS, y publica documentación actual. Es un nicho mucho menor que el de COBOL, pero es un nicho real: bancos y aseguradoras con aplicaciones críticas que nunca se reescribieron.*
>
> *Entra por dos conceptos que **el núcleo no muestra**. El primero es técnico: PL/I tuvo **manejo estructurado de excepciones (`ON` conditions), punteros, multitarea y aritmética decimal a la vez, en 1964** — antes de que ninguna de esas ideas se considerara normal. Leerlo es ver el origen del `try/catch`. El segundo es una lección de **diseño de lenguajes**, que es de lo que trata la Parte 1: PL/I es el caso de estudio canónico de qué pasa cuando un lenguaje intenta cubrirlo todo. Dijkstra lo usó como advertencia; entender **por qué** es más útil que memorizar la anécdota.*

Año	1964 (como *NPL*); estándar ANSI en 1976
Autoría	**IBM** con el comité de usuarios **SHARE**, para el System/360
Familia	Propósito general — síntesis deliberada de ALGOL, FORTRAN y COBOL
Paradigma	Imperativo, procedimental, estructurado, con concurrencia
Tipado	Estático, con **conversión implícita muy permisiva** entre tipos
Memoria	Cuatro clases de almacenamiento: `STATIC`, `AUTOMATIC`, `CONTROLLED`, `BASED`
Ejecución	Compilado a nativo (z/OS, AIX, Windows, OS/2)
Estado	🟡 **Vivo pero minoritario** — mainframe empresarial, mucho menor que COBOL

📜 Historia

En 1963 IBM tenía un problema de estrategia. Sus clientes científicos usaban FORTRAN, sus clientes empresariales usaban COBOL, y la nueva línea **System/360** pretendía ser *una sola familia de máquinas para todos*. Tener dos lenguajes y dos comunidades separadas contradecía la idea. El comité *Advanced Language Development* del grupo de usuarios **SHARE** recibió el encargo de diseñar un lenguaje único.

El resultado, primero llamado **NPL** (*New Programming Language*) y renombrado **PL/I** por conflicto de marcas, se publicó en 1964. Su ambición era total: cálculo científico en punto flotante y aritmética decimal empresarial, procesamiento de cadenas, ficheros, punteros y estructuras dinámicas, multitarea, y **manejo de condiciones excepcionales** con `ON ERROR`, `ON ENDFILE`, `ON OVERFLOW`. Todo eso, en 1964, cuando FORTRAN no tenía siquiera estructuras de datos y COBOL no tenía punteros.

También llevó al extremo una idea peligrosa: **no hay palabras reservadas**. `IF IF = THEN THEN THEN = ELSE;` es una sentencia legal de PL/I, porque `IF`, `THEN` y `ELSE` pueden ser nombres de variable y el compilador lo resuelve por contexto. Sumado a la conversión automática entre casi cualquier par de tipos, el lenguaje se volvió difícil de compilar y, sobre todo, difícil de predecir.

Esa es la crítica que lo persigue. **Edsger Dijkstra** escribió en 1972, en su conferencia del Premio Turing *The Humble Programmer*, uno de los juicios más citados de la disciplina: describió a PL/I como un lenguaje cuya complejidad lo hacía inmanejable, comparándolo con una enfermedad fatal. La frase es célebre y algo injusta —PL/I hizo cosas notables— pero señala un problema real: **la generalidad tiene un coste cognitivo, y ese coste lo paga quien lee el código.**

Un dato que compensa la mala fama: **Multics**, el sistema operativo del MIT/Bell Labs que inspiró directamente a Unix, se escribió en un subconjunto de PL/I llamado EPL. Fue de los primeros sistemas operativos escritos en un lenguaje de alto nivel, y esa decisión influyó en que Ken Thompson y Dennis Ritchie escribieran Unix en C.

🏭 Dónde sobrevive hoy

- **Banca y seguros** con aplicaciones z/OS heredadas, típicamente conviviendo con COBOL en el mismo sistema.
- **Grandes corporaciones** con desarrollos de los 70 y 80 que resultaron más baratos de mantener que de reescribir.
- Integrado con **CICS** (transaccional), **Db2** (datos) e **IMS** (jerárquico), como el resto del ecosistema mainframe.

Un patrón habitual: PL/I para la lógica compleja y de cálculo, COBOL para el grueso del proceso por lotes, ambos orquestados por JCL.

🧠 Por qué no ha muerto

1. Estaba donde estaba el dinero, y funcionó. Las razones son las de COBOL: reglas de negocio sedimentadas, riesgo asimétrico y coste de migración.

2. IBM lo sigue manteniendo. *Enterprise PL/I for z/OS* recibe versiones nuevas, con explotación de las instrucciones modernas del procesador z e interoperabilidad con COBOL, C y Java en el mismo sistema. No es un compilador abandonado.

3. Es más expresivo que COBOL para cálculo. Donde COBOL necesita rodeos, PL/I tiene expresiones, recursión, punteros y estructuras dinámicas. En aplicaciones actuariales o de riesgo, eso importaba.

4. Su modelo de excepciones es genuinamente bueno. Las `ON` conditions permiten instalar manejadores por condición (`ZERODIVIDE`, `OVERFLOW`, `ENDFILE`, `KEY`) y, en muchos casos, **continuar** la ejecución. Es más parecido al sistema de condiciones y reinicios de Common Lisp que al `try/catch` que heredamos.

🔄 Lo que se ha modernizado

Menos que COBOL, pero más de lo que su fama sugiere:

- **Aritmética decimal de 64 bits** y explotación de las instrucciones vectoriales del procesador **z/Architecture**, con optimización específica para el hardware actual.
- **JSON y XML**: el compilador incorpora soporte para generar y analizar ambos formatos, igual que COBOL, para poder participar en integraciones modernas.
- **Interoperabilidad**: llamadas entre PL/I, COBOL, C/C++ y Java dentro del mismo entorno **Language Environment**, lo que permite mantener PL/I como componente dentro de un sistema mixto.
- **Direccionamiento de 64 bits** y soporte Unicode (UTF-8/UTF-16).
- **Herramientas actuales**: depuración desde **VS Code** con IBM Z Open Editor, integración con Git y pipelines de CI/CD sobre z/OS mediante IBM Dependency Based Build.

Lo que **no** ha ocurrido: no hay una comunidad libre significativa ni un compilador abierto de referencia. Esa es la diferencia práctica con COBOL, que sí tiene GnuCOBOL, y la razón por la que PL/I está en 🟡 y no en 🟢.

⚙️ Cómo se ejecuta hoy

En producción: *IBM Enterprise PL/I for z/OS*. Compilación mediante JCL invocando el compilador, enlace y ejecución.

Fuera del mainframe: existen implementaciones para otras plataformas —**Micro Focus / OpenText Open PL/I** y **Iron Spring PL/I** (un subconjunto para Linux y OS/2)— pero son minoritarias y parciales.

```
//COMPILA  EXEC PGM=IBMZPLI,PARM='OBJECT,SOURCE'
//SYSIN    DD   DSN=MI.FUENTE(TOTVTA),DISP=SHR
//SYSLIN   DD   DSN=&&OBJ,DISP=(NEW,PASS)
```

Es decir: para compilar PL/I hay que escribir JCL. Otro recordatorio de que en el mainframe ningún lenguaje se usa solo.

🧪 El programa de la clase 041 en PL/I

> ⚠️ ***Material de lectura, no verificado.*** *No hay compilador PL/I en los runners de CI. El código está escrito para ser correcto e idiomático, pero* ***sin el sello de la máquina****, y así se declara.*

```
total_venta: procedure options(main);

   declare precio     fixed decimal(11,2);
   declare cantidad   fixed decimal(11,2);
   declare descuento  fixed decimal(5,4);
   declare total      fixed decimal(15,2);
   declare presenta   picture 'ZZZZZZZZZ9V.99';

   on endfile(sysin) stop;

   get list (precio, cantidad, descuento);

   total = precio * cantidad * (1 - descuento);

   presenta = total;
   put skip list ('Total: ' || trim(presenta));

end total_venta;
```

Recorrido, línea a línea.

- `procedure options(main)` marca el punto de entrada. En PL/I **todo es un procedimiento**, incluido el programa principal; no hay una sintaxis especial para el `main`.
- `fixed decimal(11,2)` es aritmética **decimal de coma fija**: 11 dígitos, 2 decimales, exacta. Es el mismo tipo conceptual que el `PIC 9(9)V99 COMP-3` de COBOL, pero declarado como un tipo y no como una plantilla de imagen. Junto a él conviven `fixed binary`, `float decimal` y `float binary`: PL/I es de los pocos lenguajes que distingue explícitamente **base** (decimal o binaria) de **escala** (fija o flotante). Esa matriz de cuatro combinaciones es exactamente la unión de FORTRAN y COBOL que el lenguaje buscaba.
- `on endfile(sysin) stop;` **instala un manejador de condición**. No es un `try` que envuelve un bloque: es una declaración que dice "de aquí en adelante, si se alcanza el fin de fichero en `sysin`, ejecuta esto". El ámbito es dinámico y el manejador queda activo durante toda la ejecución. Esta línea es,

cronológicamente, uno de los primeros mecanismos de manejo estructurado de errores de la historia.

- `get list (...)` es la lectura dirigida por lista, igual que el `read(*,*)` de Fortran: lee tantos valores como variables haya, separados por espacios o comas.
- `picture 'ZZZZZZZZZ9V.99'` es un **dato numérico de imagen**, el mismo concepto que el campo editado de COBOL: cada `Z` suprime un cero a la izquierda, el `9` garantiza que el cero se imprima, y `V.` marca a la vez el punto decimal supuesto y el punto que se imprime. Asignar `total` a `presenta` **convierte automáticamente** de decimal a texto formateado.
- `||` es la concatenación de cadenas, y `trim` recorta los espacios que deja la supresión de ceros.

Lo que no se ve pero define al lenguaje: casi cualquier asignación entre tipos distintos funcionaría aquí. Asignar una cadena `'27000'` a un `fixed decimal` compila y convierte. Es cómodo y es, a la vez, exactamente la propiedad que hace que los errores de PL/I se manifiesten tarde. Compara con Ada, diseñado veinte años después con la filosofía contraria.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En PL/I es…
`int` / `double`	`fixed binary(31)` / `float binary(53)`
`decimal` / `BigDecimal`	`fixed decimal(15,2)` — nativo
`try { } catch (E e) { }`	`on condition ... ;` — instalado, no envuelto
`struct`	`declare 1 registro, 2 campo ...;` — niveles, como COBOL
Puntero	`pointer` con variables `based`
`malloc` / `free`	`allocate` / `free` sobre almacenamiento `controlled` o `based`
`static` / variable local	`static` / `automatic` — las clases de almacenamiento son explícitas
Hilos	`task` — multitarea en el lenguaje, en 1964
`printf("%s", x)`	`put skip list (x)` o `put edit (x) (formato)`

⚠️ Errores comunes al leerlo

- **Suponer palabras reservadas.** Una variable puede llamarse `IF` o `DO`. Al leer código antiguo, no des por hecho que una palabra clave lo es.
- **Ignorar las conversiones implícitas.** PL/I convierte casi todo a casi todo. Una operación entre `fixed decimal` y `fixed binary` produce reglas de precisión que hay que consultar en el manual; no son intuitivas y son fuente de errores de redondeo silenciosos.
- **Leer `on` como `try`.** El manejador se instala y permanece; no delimita un bloque. Dos `on` para la misma condición en el mismo ámbito significan que el segundo sustituye al primero.
- **Confundir `%INCLUDE` con un `import`.** Es una inclusión textual en tiempo de preproceso, como el `#include` de C, con las mismas consecuencias.
- **Olvidar la columna.** Igual que COBOL, el PL/I clásico usa formato de columnas fijas heredado de la tarjeta perforada (habitualmente 2–72).

📚 Fuentes y bibliografía

- IBM Enterprise PL/I for z/OS — documentación (https://www.ibm.com/docs/en/epfz) — referencia del lenguaje y guía de programación, versión vigente.
- IBM Enterprise PL/I (producto) (https://www.ibm.com/products/pli-compiler-zos) — el compilador que IBM mantiene y vende hoy.

- **Edsger W. Dijkstra**, *The Humble Programmer*, ACM Turing Award Lecture, 1972 — texto original (https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html). Léelo por el argumento sobre la complejidad, no por la frase suelta.
- **Joan K. Hughes**, *PL/I Structured Programming*, Wiley — el manual clásico del lenguaje.
- **Robin A. Vowels**, *Introduction to PL/I and Algorithms* — enfoque didáctico y accesible.
- **Multics History** (multicians.org (https://www.multicians.org/)) — la historia del sistema operativo escrito en PL/I que llevó, por reacción, a Unix y a C.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: COBOL · RPG · JCL · C

🔷 PowerShell — 2006

PowerShell parte de una idea que replantea cincuenta años de tradición Unix: **¿y si por las tuberías viajaran objetos en lugar de texto?** El resultado elimina de un golpe el análisis con expresiones regulares que domina el trabajo en Bash — y trae sus propias asperezas.

> 🎯 ***Por qué está en este programa***
>
> *PowerShell es un **primo de la familia históricos / shell** (Atlas), junto a Bash y JCL.*
>
> *Aporta al programa **la alternativa al contrato de texto** (clases 159 y 161): una tubería con datos estructurados y tipados, en lugar de con cadenas que hay que volver a analizar. Es la mejor demostración de que **el formato de la frontera decide el trabajo que hay que hacer en los dos lados**.*

Año	2006; **PowerShell Core 6** multiplataforma y libre (2018); **7.x** actual
Autoría	**Jeffrey Snover**, Microsoft — a partir del *Monad Manifesto* (2002)
Familia	Históricos / shell; sobre **.NET** (C#)
Paradigma	Imperativo y de canalización, **orientado a objetos**
Tipado	**Dinámico con tipos de .NET**; con anotaciones opcionales
Memoria	La del CLR: recolección de basura
Ejecución	Interpretado sobre .NET, con compilación a IL de los bloques de guion
Estado	🟢 **Estándar** en administración de Windows, Azure y Microsoft 365

📜 Historia

En **2002**, **Jeffrey Snover** escribió el ***Monad Manifesto***, un documento que empieza con un diagnóstico incómodo para Microsoft: **la administración de Windows por interfaz gráfica no escala**, y **el modelo de Unix —herramientas de texto compuestas por tuberías— no se puede copiar directamente**.

Y la razón que daba es interesante: **en Unix, la configuración está en ficheros de texto**, así que `grep` y `sed` bastan. **En Windows, la configuración está en el registro, en WMI y en APIs de objetos**, y convertir eso a texto para volver a analizarlo **pierde información y es frágil**.

Su propuesta fue **cambiar lo que viaja por la tubería**:

```
Get-Process | Where-Object CPU -gt 100 | Sort-Object CPU -Descending | Select-Object -First 5
```

Ahí no viaja texto: viajan objetos `Process`, con sus propiedades y sus métodos. **No hay que analizar nada**, porque `CPU` es un número y no una columna que haya que extraer.

PowerShell 1.0 salió en 2006, y su adopción fue lenta hasta que Microsoft tomó una decisión estratégica: **exigir que todo producto de servidor expusiera su funcionalidad como cmdlets**, y que **la interfaz gráfica se construyera encima**. Exchange fue el primero.

En 2016 se abrió el código y en 2018 llegó PowerShell Core, multiplataforma sobre .NET Core — que lo llevó a Linux y a macOS.

🏭 Dónde vive hoy

- **Administración de Windows**: es la forma estándar; el Administrador de Servidor genera PowerShell.
- **Azure y Microsoft 365**: `Az` y `Microsoft.Graph` son la vía de automatización.
- **Integración continua y despliegue**: pasos de canalización en Azure DevOps y GitHub Actions (clases 147 y 171).
- **Gestión de configuración**: DSC (*Desired State Configuration*) — el modelo declarativo de estado deseado que la clase 171 recomienda.
- **Y en Linux**, sobre todo en entornos mixtos y para gestionar servicios de Microsoft.

🧠 Lo que enseña: objetos por la tubería

La comparación con Bash es el contenido de esta ficha:

```
# Bash: TODO es texto, hay que extraer columnas y confiar en el formato
ps aux | awk '$3 > 50 {print $11}' | sort | head -5
```

```
# PowerShell: viajan objetos, se accede a propiedades con nombre
Get-Process | Where-Object CPU -gt 50 | Select-Object -First 5 -Property Name
```

Y las diferencias son de fondo:

	Texto (Bash)	**Objetos (PowerShell)**
Extraer un dato	**analizar** con `awk`, `cut`, `sed`	acceder a la propiedad
Si cambia el formato	**se rompe en silencio**	no hay formato que cambiar
Tipos	todo cadena; convertir a mano	`DateTime`, `Int`, `FileInfo`…
Interoperabilidad	**universal**: cualquier programa	dentro del ecosistema .NET
Rendimiento	procesos ligeros, texto barato	objetos y CLR: más pesado

Y las dos últimas filas son el precio. El contrato de texto de Unix es **universal**: cualquier programa, en cualquier lenguaje, de cualquier época, participa (clase 161). **Los objetos de PowerShell solo los entiende PowerShell** — y al llamar a `git` o a `docker`, **vuelve el texto**.

Es exactamente la tensión de la clase 159: **el formato que captura más información es el que menos gente entiende**.

Y hay una segunda cosa que PowerShell hace notablemente bien: la coherencia.

```
Get-Process   Get-Service   Get-ChildItem   Get-Content
Set-Location  Remove-Item   New-Item        Start-Service
```

Todos los comandos son `Verbo-Sustantivo`, con una **lista aprobada de verbos**. Eso significa que **se puede adivinar el nombre de un comando que no se conoce** — y `Get-Verb` la lista.

Es la misma virtud que la clase 167 señalaba en los comandos CL de IBM i: **la consistencia hace que aprender uno sea aprender todos**, frente a la libertad total de Unix donde cada herramienta inventa sus opciones.

🔄 Lo que se ha modernizado

- **PowerShell 7**: multiplataforma, con `&&` y `||`, operador ternario, `??` y paralelismo en `ForEach-Object -Parallel`.
- **PSScriptAnalyzer**: análisis estático con reglas, al estilo de la clase 146.
- **Pester**: marco de pruebas maduro para guiones (clase 139).
- `SecretManagement`: gestión de credenciales sin ponerlas en el guion (clase 153).
- **Clases y enumerados** en el lenguaje, y módulos con manifiesto y versión (clase 143).
- **Y `ConvertTo-Json` / `ConvertFrom-Json`**, que son el puente al mundo del texto estructurado (clase 159).

⚙️ Cómo se ejecuta hoy

```
pwsh main.ps1                              # PowerShell 7, multiplataforma
powershell.exe -File main.ps1               # Windows PowerShell 5.1

Invoke-ScriptAnalyzer -Path main.ps1        # análisis (clase 146)
Invoke-Pester                                # pruebas (clase 139)
```

🍼 El programa de la clase 041 en PowerShell

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
$campos = ($input | Select-Object -First 1) -split '\s+'

[double]$precio    = $campos[0]
[double]$cantidad  = $campos[1]
[double]$descuento = $campos[2]

$total = $precio * $cantidad * (1 - $descuento)

'Total: {0:F2}' -f $total
```

Lo que hay que ver.

- `[double]$precio` **es una conversión con tipo**, no una anotación: PowerShell **convierte y comprueba**, y falla si no puede. Es tipado dinámico **con los tipos de .NET** disponibles.
- `-split '\s+'` **usa una expresión regular** y devuelve **un arreglo de verdad**, no una cadena que haya que volver a partir.
- `-f` **es el operador de formato**, y `{0:F2}` es el formato de .NET — **el mismo `F2` que en C# y VB.NET** (clase 157). Y con la misma advertencia: **la cultura regional afecta al separador decimal**, así que en un guion serio conviene fijarla.
- **La última línea no lleva `Write-Output`: el resultado de una expresión suelta va a la tubería**. Es una decisión del lenguaje que ahorra ceremonia y que despista al principio.

- **Y este programa es lo menos representativo de PowerShell posible**: su terreno no es leer texto de la entrada estándar, sino **componer cmdlets que devuelven objetos**. Es un **contrato adaptado** (clase 040).

📚 Fuentes y bibliografía

- Documentación de PowerShell (https://learn.microsoft.com/powershell/) — extensa; y `Get-Help` con `-Examples` dentro del propio shell.
- **Jeffrey Snover**, *Monad Manifesto* (2002) — libre en línea; **el documento de diseño**, y una lectura excelente para la clase 175: plantea el problema, descarta alternativas y justifica la decisión.
- **Don Jones, Jeffery Hicks**, *Learn PowerShell in a Month of Lunches*, Manning — la introducción de referencia.
- **Bruce Payette, Richard Siddaway**, *Windows PowerShell in Action*, 3.ª ed., Manning — escrito por uno de los diseñadores del lenguaje; el más profundo.
- PSScriptAnalyzer (https://github.com/PowerShell/PSScriptAnalyzer) y Pester (https://pester.dev/) — calidad y pruebas.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Bash · C# · Tcl · JCL · Perl

🔗 Prolog — 1972

Prolog es el lenguaje que más lejos lleva la idea declarativa: **no se escriben instrucciones, se escriben hechos y reglas**, y el sistema busca por sí mismo las respuestas que se derivan de ellos. Escribir Prolog exige desaprender casi todo lo demás — y por eso enseña tanto.

> 🎯 ***Por qué está en este programa***
>
> *Prolog es un **primo de la familia lógica y declarativa** (Atlas), cuyo representante en el núcleo es SQL.*
>
> *Aporta al programa el paradigma lógico en su forma pura (clase 119): **unificación** en lugar de asignación, **vuelta atrás** en lugar de bucles, y **una relación que se puede recorrer en las dos direcciones**. Es el contraste más fuerte de todo el Atlas frente a los lenguajes imperativos.*

Año	1972; **ISO Prolog** en 1995
Autoría	**Alain Colmerauer** y Philippe Roussel, Marsella; teoría de **Robert Kowalski**
Familia	Lógica; basado en la **resolución SLD** sobre cláusulas de Horn
Paradigma	**Lógico declarativo**
Tipado	**Dinámico y sin tipos declarados**; los términos son la única estructura
Memoria	Recolección de basura, con pila de puntos de elección
Ejecución	Máquina abstracta de Warren (WAM); compilado o interpretado
Estado	🟢 **Vivo en nichos**: IA simbólica, procesamiento de lenguaje, verificación

---

📜 Historia

En **1972**, **Alain Colmerauer** trabajaba en Marsella en **procesamiento de lenguaje natural** — concretamente, en responder preguntas en francés—. Necesitaba una forma de expresar reglas gramaticales y deducir consecuencias, y con la base teórica de **Robert Kowalski** —que había demostrado que un

subconjunto de la lógica de primer orden, las **cláusulas de Horn**, se puede ejecutar— nació **Prolog**: *programmation en logique*.

La idea de Kowalski se resume en una ecuación famosa:

> ***Algoritmo = Lógica + Control.***

Es decir: **lo que se quiere calcular** y **cómo se busca** son cosas separables. El programa Prolog expresa la lógica; el motor aporta el control.

David H. D. Warren hizo Prolog práctico en 1983 con la **WAM**, una máquina abstracta que se convirtió en el objetivo de compilación estándar — y cuyo papel es el mismo que la JVM tendría después (clase 125).

El momento de máxima atención llegó con el **Proyecto de Quinta Generación japonés (1982-1992)**, que apostó mil millones de dólares por construir ordenadores basados en lógica y en paralelismo. **No alcanzó sus objetivos**, y su fracaso —junto con el del auge de los sistemas expertos— arrastró la reputación de Prolog durante décadas.

Y hoy sus ideas están más vivas que su nombre: **Datalog** en bases de datos, **la programación por restricciones** en logística e industria, y **la unificación** en los sistemas de tipos de casi todos los lenguajes con inferencia.

🏭 Dónde vive hoy

- **Procesamiento de lenguaje natural** y sistemas de reglas, sobre todo donde hay que explicar por qué se llegó a una conclusión.
- **Programación por restricciones**: SWI-Prolog y SICStus con **CLP(FD)** resuelven horarios, asignación de recursos, rutas y planificación industrial — es su uso más rentable hoy.
- **Verificación y análisis de programas**, con motores de deducción.
- **IBM Watson** usó Prolog para el análisis sintáctico en el sistema que ganó *Jeopardy!*.
- **Enseñanza**: sigue siendo la forma estándar de enseñar programación lógica.

🧠 Lo que enseña: unificación y relaciones reversibles

La unificación no es asignación (clase 119):

```
X = 3.          % ← liga X con 3, o comprueba que ya vale 3
persona(ana, 30).
persona(luis, 25).

?- persona(Quien, 30).      % ← ¿quién tiene 30 años?
Quien = ana.
```

El mismo `=` sirve para ligar y para comprobar, y **una variable solo se liga una vez** — es lo que la clase 041 llamaría una constante, no una variable.

Y de ahí sale la propiedad más asombrosa del paradigma: **una relación se puede recorrer en cualquier dirección**.

```
concat([], L, L).
concat([H|T], L, [H|R]) :- concat(T, L, R).

?- concat([1,2], [3], X).      % X = [1,2,3]        ← concatenar
?- concat(X, [3], [1,2,3]).     % X = [1,2]          ← ¡DESCONCATENAR!
?- concat(X, Y, [1,2,3]).       % TODAS las formas de partir la lista
```

Se define una vez y se usa de tres maneras. En un lenguaje imperativo habría que escribir tres funciones. **Esa reversibilidad es lo que ningún otro paradigma da**, y es la razón de que Prolog siga enseñándose.

Y la **vuelta atrás** es el mecanismo de control:

```
?- persona(X, E), E > 26.       % prueba ana → sí; prueba luis → no; devuelve ana
```

El motor prueba alternativas y deshace lo que no funciona, automáticamente. Es búsqueda con retroceso, integrada en el lenguaje.

> ***Y la honestidad exige decir dónde se rompe la abstracción**: **el orden de las cláusulas importa**, y un programa lógicamente correcto puede entrar en un bucle infinito según cómo esté escrito. Por eso existe **el corte (`!`)**, que poda la búsqueda — y con él vuelve el control imperativo por la puerta de atrás. **La ecuación de Kowalski es un ideal; la práctica es más sucia**, y saberlo forma parte de entender el paradigma.*

🔄 Lo que se ha modernizado

- **CLP(FD)** y la programación por restricciones: en lugar de generar y probar, **se propagan restricciones** y se poda el espacio de búsqueda. Es lo que hace útil a Prolog en optimización real.
- **SWI-Prolog** como implementación moderna: servidor web integrado, JSON, interfaz con Python (`janus`), depurador gráfico y gestor de paquetes (clase 143).
- **Tabulación (*tabling*)**: memorización de subobjetivos que **evita bucles infinitos** y acerca Prolog a Datalog en sus garantías.
- **Scryer Prolog** y **Trealla**: implementaciones nuevas escritas en Rust y en C, centradas en el estándar y en el rigor.
- **Y el renacimiento por la IA**: los sistemas neurosimbólicos combinan modelos de lenguaje con motores lógicos, precisamente porque **la deducción es explicable y el modelo no**.

⚙️ Cómo se ejecuta hoy

```
swipl -g main -t halt main.pl < entrada.txt     # ejecutar un guion
swipl                                            # consola interactiva: el uso natural

?- [main].          % cargar
?- trace.            % ← depurador de resolución, paso a paso (clase 141)
```

🧪 El programa de la clase 041 en Prolog

Es la versión que aparece en el `primos.md` de la clase 041.

```
:- initialization(main, main).

main :-
    read_line_to_string(user_input, Linea),
    split_string(Linea, " ", "", Partes),
    maplist([S, N]>>number_string(N, S), Partes, [Precio, Cantidad, Descuento]),
    Total is Precio * Cantidad * (1 - Descuento),
    format("Total: ~2f~n", [Total]).
```

Lo que hay que ver.

- `Total is ...` **NO es una asignación.** `is` evalúa la expresión aritmética y **unifica** el resultado con `Total`. Si `Total` ya estuviera ligado, **esto sería una comprobación**, no una escritura (clase 041).
- **Las mayúsculas son variables y las minúsculas son átomos.** `Precio` es una variable; `main` es un átomo. Es al revés de la convención de casi todos los lenguajes.
- **La coma es una conjunción lógica**, no un separador de sentencias: `A, B` significa "demuestra A **y** demuestra B", y si B falla, **se vuelve atrás sobre A** para probar otra alternativa.
- `maplist` **con** `[S,N]>>...` es una lambda (biblioteca `yall`), y aplica la relación `number_string` a cada elemento — la misma idea que `map` en el resto de las fichas (clase 115).
- **Y** `main` **no es una función: es un predicado que se demuestra.** Si algo falla, el predicado falla y el programa termina sin salida — que es la semántica lógica, no la imperativa.

📚 Fuentes y bibliografía

- The Power of Prolog (https://www.metalevel.at/prolog) — **Markus Triska**; libre en línea y de lo mejor escrito sobre el lenguaje, con mucha atención a hacerlo bien y no solo a que funcione.
- Learn Prolog Now! (http://www.learnprolognow.org/) — libre; la introducción académica estándar.
- Manual de SWI-Prolog (https://www.swi-prolog.org/pldoc/doc_for?object=manual) — la implementación más completa.
- **Leon Sterling, Ehud Shapiro**, *The Art of Prolog*, MIT Press — el clásico.
- **Robert Kowalski**, *Algorithm = Logic + Control* (1979) — el artículo que fundamenta el paradigma; corto y sigue siendo lúcido.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Datalog · SQL · Erlang · Common Lisp

Python — 1991

Python es hoy el lenguaje más enseñado del mundo y el que más gente escribe sin llamarse programadora. Su historia explica las dos cosas: **nació para ser legible**, y treinta años después esa decisión resultó valer más que casi cualquier otra característica técnica.

> 🎯 ***Por qué está en este programa***
>
> ***Python es uno de los diez lenguajes del núcleo****: se implementa y se verifica en CI en las 136 clases de código. Esta ficha no repite eso — cuenta su historia y responde a por qué se eligió como representante.*
>
> *Python es el* ***representante del scripting dinámico*** *(Atlas): tipado dinámico, poca ceremonia, y una biblioteca estándar que cubre casi todo. Quien entiende la versión Python de una clase reconoce después la de Ruby, Perl, Lua, Tcl o R sin haberlas visto. Y es, además, el lenguaje con el que hoy se pega el resto del sistema (clase 155).*

Año	1991; **2.0** en 2000; **3.0** en 2008; ciclo anual desde **3.9**
Autoría	**Guido van Rossum**, CWI (Ámsterdam), como sucesor de **ABC**
Familia	Scripting dinámico — con influencia de ABC, Modula-3, C y Lisp
Paradigma	Multiparadigma: imperativo, orientado a objetos y funcional
Tipado	**Dinámico y fuerte**; anotaciones opcionales comprobadas por herramientas
Memoria	Automática: **conteo de referencias** + recolector de ciclos
Ejecución	Compilado a bytecode e interpretado (CPython); PyPy con JIT
Estado	🟢 **Dominante** en ciencia de datos, IA, automatización y enseñanza

---

📜 Historia

Guido van Rossum trabajaba en el CWI de Ámsterdam en **ABC**, un lenguaje de enseñanza muy cuidado que nunca despegó. De ABC se llevó una obsesión —**que el código se lea**— y una lección: ABC era cerrado y no se podía extender, y por eso murió.

En las navidades de **1989** empezó Python como proyecto personal. El nombre no viene del reptil sino de **Monty Python's Flying Circus**, y esa broma marcó el tono de toda la documentación durante décadas.

Las decisiones que lo definieron estaban ahí desde el principio: **la indentación como sintaxis** —el bloque se marca con espacios, no con llaves—, **todo es un objeto**, y una biblioteca estándar grande, la política de las **"baterías incluidas"**.

Python 2.0 (2000) trajo la comprensión de listas y el recolector de ciclos. **Python 3.0 (2008)** rompió la compatibilidad para arreglar lo que estaba mal —sobre todo la distinción entre **bytes y texto**, que en Python 2 era una fuente constante de errores (clase 093)— y la migración costó **doce años**: el soporte de Python 2 terminó en 2020.

Esa transición es un caso de estudio de la clase 143: una ruptura necesaria, técnicamente correcta y socialmente carísima.

Desde **3.9** hay una versión anual con fecha fija, y el lenguaje ha ido incorporando **anotaciones de tipo** (2015), `async` / `await` (2015), el operador morsa (2018), el emparejamiento estructural (2021) y, en marcha, la eliminación del bloqueo global del intérprete.

🏭 Dónde vive hoy

- **Ciencia de datos e inteligencia artificial**: NumPy, pandas, scikit-learn, PyTorch, TensorFlow. Es el idioma común del campo, sin discusión.
- **Automatización y administración de sistemas**: el sucesor natural de los guiones de Perl y de shell.
- **Web de servidor**: Django, FastAPI, Flask.
- **Enseñanza**: es el primer lenguaje en la mayoría de las universidades del mundo.
- **Herramientas de desarrollo**: Ansible, dbt, buena parte de los sistemas de construcción y de despliegue.
- **Investigación científica**: la sustitución de MATLAB en muchos laboratorios, con el ecosistema SciPy.

🧠 La decisión que lo explica: legibilidad sobre todo lo demás

Python renuncia a cosas que otros lenguajes consideran irrenunciables, y **cada renuncia compra legibilidad**:

- **No hay llaves ni `end`**: la indentación *es* el bloque. Elimina de golpe la discusión de estilo y hace imposible que el sangrado mienta sobre la estructura.
- **No hay asignación dentro de una expresión** —hasta el operador morsa, y con reservas—, ni incremento `++`, ni operador ternario con `?:`.
- **`self` explícito** en los métodos: se ve de dónde viene cada atributo.
- **Solo hay una forma obvia de hacerlo**: es la línea más citada del *Zen de Python*, y es lo contrario del lema de Perl (clase 146).

Y el coste hay que decirlo con la misma claridad:

> ***El precio es el rendimiento y la concurrencia.*** *CPython interpreta bytecode y tiene un* ***bloqueo global del intérprete*** *(el GIL) que impide que dos hilos ejecuten bytecode a la vez. Por eso el Python que va rápido* ***casi nunca es Python****: NumPy, PyTorch y pandas son C, C++ y Fortran por debajo (clase 155), y Python es la capa que los compone. Entender eso es entender por qué el lenguaje "lento" domina el cálculo intensivo.*

🔄 Lo que se ha modernizado

- **Anotaciones de tipo y comprobadores estáticos**: `mypy`, `pyright`, `ty`. El tipado sigue siendo dinámico en ejecución, pero **el análisis previo caza una familia entera de errores** — es la misma idea que la clase 146 defiende para todos los lenguajes.
- **`async` / `await`** y el ecosistema asíncrono (asyncio, anyio, FastAPI) — clase 134.
- **Emparejamiento estructural** (`match` / `case`) desde 3.10, al estilo ML.
- **Herramientas de una generación nueva escritas en Rust**: `ruff` (análisis y formato), `uv` (gestor de paquetes y entornos), que han reducido de minutos a segundos operaciones cotidianas.
- **PEP 703: el intérprete sin GIL**, opcional desde 3.13 y en camino de ser el modo por defecto. Es el cambio más profundo del lenguaje desde Python 3.
- **Mejoras de rendimiento sostenidas** desde 3.11 (el proyecto *Faster CPython*), con intérprete especializado y un JIT experimental.

⚙️ Cómo se ejecuta hoy

```
python3 main.py < entrada.txt          # el comando de la clase 041

# Entorno y dependencias reproducibles (clase 143):
uv venv && uv pip install -r requirements.txt
uv run main.py

# Calidad, como en la clase 146:
ruff check . && ruff format --check .
mypy .
pytest
```

🪠 El programa de la clase 041 en Python

Es el que se ejecuta y se verifica en la clase 041.

```
import sys

# Literales y constantes: los valores se leen y se nombran.
precio_str, cantidad_str, descuento_str = sys.stdin.readline().split()

PRECIO_UNITARIO = float(precio_str)   # tipo dinámico, inferido en tiempo de ejecución
CANTIDAD = int(cantidad_str)
DESCUENTO = float(descuento_str)

subtotal = PRECIO_UNITARIO * CANTIDAD
total = subtotal * (1 - DESCUENTO)

print(f"Total: {total:.2f}")
```

Lo que hay que ver, comparando con las otras fichas.

- **La desestructuración en la primera línea** —tres nombres para tres trozos— es la misma que hacen Ruby, Kotlin y Rust, y que Java no permite.
- `PRECIO_UNITARIO` **en mayúsculas no es una constante**: es una convención. Python **no tiene constantes** (clase 041), y esa ausencia es una decisión: el lenguaje confía en el acuerdo y no en el compilador. Compárese con Ada, donde una constante es un tipo con dominio.
- **La conversión es explícita** (`float`, `int`) porque Python tiene **tipado fuerte**: `"3" + 4` es un error, a diferencia de JavaScript o PHP (clase 100).
- `f"{total:.2f}"` es interpolación con formato; la misma familia de `%.2f` que aparece en casi todas las fichas, con otra sintaxis.

🧶 Fuentes y bibliografía

- Documentación oficial de Python (https://docs.python.org/3/) — el tutorial y la referencia de la biblioteca estándar siguen siendo de lo mejor escrito en cualquier lenguaje.
- Índice de PEP (https://peps.python.org/) — cada decisión del lenguaje, con su discusión. **PEP 8** (estilo), **PEP 20** (el Zen), **PEP 484** (tipos), **PEP 703** (sin GIL).
- **Luciano Ramalho**, *Fluent Python*, 2.ª ed., O'Reilly — el libro que enseña a escribir Python como Python y no como Java con otra sintaxis.
- **Brett Slatkin**, *Effective Python*, 3.ª ed., Addison-Wesley — 125 consejos concretos.
- **David Beazley, Brian K. Jones**, *Python Cookbook*, 3.ª ed., O'Reilly — recetas con explicación.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Ruby · Perl · Lua · C · Julia

📊 R — 1993

R no es un lenguaje de programación de propósito general al que se le añadió estadística: **es estadística a la que se le añadió un lenguaje**. Esa inversión explica todo lo que resulta raro al llegar de otro sitio — y también por qué sigue siendo insustituible en su terreno.

> 🎯 ***Por qué está en este programa***
>
> *R es un **primo de la familia de scripting dinámico** (Atlas), cuyos representantes en el núcleo son Python y PHP.*

> *Aporta al programa la **programación vectorizada**: en R, la unidad de trabajo **no es el valor, es el vector**, y casi ningún programa lleva bucles (clase 089). Es la misma idea que APL, J, Fortran moderno y NumPy, y quien la entiende aquí la reconoce en los cuatro.*

Año	1993; **1.0** en 2000; el *tidyverse* a partir de 2014
Autoría	**Ross Ihaka** y **Robert Gentleman**, Universidad de Auckland — de ahí las dos erres
Familia	Scripting dinámico / array; implementación libre de **S** (Bell Labs, 1976)
Paradigma	Funcional y vectorizado; con varios sistemas de objetos (S3, S4, R5, R7)
Tipado	**Dinámico**, con vectores tipados y coerción automática
Memoria	Recolección de basura; **copia al modificar**
Ejecución	Interpretado, con partes críticas en C y Fortran
Estado	🟢 **Estándar** en estadística, bioinformática e investigación académica

---

📜 Historia

En **1976**, **John Chambers** creó **S** en los Laboratorios Bell —los mismos de C y Unix— para que los estadísticos pudieran analizar datos de forma interactiva sin escribir Fortran. S se comercializó como S-PLUS y era caro.

En **1993**, **Ross Ihaka** y **Robert Gentleman**, en Auckland, escribieron una implementación libre compatible con S. La llamaron **R** por sus dos nombres y por el juego con la letra anterior.

La decisión que lo hizo despegar fue **CRAN (1997)**: un archivo central de paquetes **con pruebas automáticas y revisión antes de publicar**. Es la misma idea de CPAN (clase 143), con una diferencia importante: **CRAN comprueba que cada paquete construya y pase sus pruebas en varias plataformas antes de aceptarlo**, y avisa al autor cuando un cambio en otro paquete lo rompe. Es integración continua a escala de ecosistema (clase 147).

Y en **2014** llegó el segundo cambio: **el *tidyverse*** de Hadley Wickham — `dplyr`, `ggplot2`, `tidyr` — que **redefinió el estilo del lenguaje** hacia canalizaciones legibles y una gramática de gráficos. Hoy hay, de hecho, **dos dialectos de R**: el base y el tidy, y conviene saberlo al leer código ajeno.

🏭 Dónde vive hoy

- **Estadística académica**: es el idioma común de la investigación cuantitativa.
- **Bioinformática y genómica**: **Bioconductor** es un ecosistema entero de miles de paquetes.
- **Farmacéutica y ensayos clínicos**: R está aceptado por las agencias reguladoras para análisis de ensayos, con validación documentada.
- **Análisis y visualización**: **ggplot2** sigue siendo la mejor biblioteca de gráficos estadísticos que existe, y **Shiny** permite publicar análisis como aplicaciones web.
- **Informes reproducibles**: **R Markdown** y **Quarto** mezclan texto, código y resultados — la respuesta directa a la deuda de reproducibilidad de la clase 154.

🧠 Lo que enseña: pensar en vectores

Este es el cambio mental que R produce y que se transfiere a NumPy, a Julia, a MATLAB y al Fortran moderno:

```
x <- c(1, 2, 3, 4, 5)
y <- x * 2 + 1          # ← se opera sobre TODO el vector. Sin bucle.
mean(x[x > 2])           # filtrar con un vector lógico, y agregar
```

No hay bucle porque no hace falta. La operación se define sobre el vector entero, y por debajo se ejecuta en C o Fortran (clase 155). Un bucle explícito en R **es lento y es señal de que hay una forma mejor** — lo contrario de lo que enseña casi cualquier otro lenguaje de esta lista.

Y hay tres rarezas de R que merecen explicación porque desconciertan a quien llega:

Una, los índices empiezan en 1, como Fortran, MATLAB, Lua y Julia — y a diferencia de casi todos los demás. Es una convención de las matemáticas, no un error.

Dos, la evaluación perezosa de los argumentos. R **no evalúa un argumento hasta que se usa**, y —más raro todavía— **una función puede ver la expresión que le pasaron, no solo su valor**:

```
graficar <- function(x) plot(x, main = deparse(substitute(x)))
graficar(altura_pacientes)   # el título sale "altura_pacientes"
```

Eso es evaluación no estándar, y es la magia que hace posible que `dplyr` escriba `filter(datos, edad > 30)` sin comillas. Es potentísimo y es la razón de que el código R sea difícil de analizar (clase 150).

Y tres, la copia al modificar: los argumentos se comportan como si se copiaran, así que **una función no puede modificar el objeto de quien la llama** (clase 081). Eso da seguridad y cuesta memoria, y el intérprete optimiza el caso común evitando copias reales.

🔄 Lo que se ha modernizado

- **El *tidyverse*** y el operador de canalización, hoy también nativo: `datos |> filter(...) |> summarise(...)`.
- **Quarto**: la evolución de R Markdown, que además publica desde Python y Julia — informes con código ejecutable y salida versionable (clase 154).
- `data.table` **y** `arrow`: rendimiento de nivel industrial sobre conjuntos grandes.
- `renv` **y** `pak`: entornos y dependencias reproducibles por proyecto, con fichero de bloqueo (clase 143) — un problema que R tuvo mal resuelto durante años.
- `Rcpp` **y** `extendr`: escribir las partes críticas en C++ o en Rust e integrarlas sin fricción (clase 156).

⚙️ Cómo se ejecuta hoy

```
Rscript main.R < entrada.txt        # el comando de la clase 041
R -q -e 'sessionInfo()'

R -e 'renv::restore()'               # entorno reproducible (clase 143)
R -e 'devtools::test()'               # pruebas con testthat (clase 139)
quarto render informe.qmd             # informe reproducible
```

🪠 El programa de la clase 041 en R

```
v <- as.numeric(strsplit(readLines("stdin", n = 1), " ")[[1]])
total <- v[1] * v[2] * (1 - v[3])
cat(sprintf("Total: %.2f\n", total))
```

Lo que hay que ver, y es lo que delata al lenguaje.

- **No hay tres variables: hay un vector `v` de tres elementos.** Todas las demás fichas de la clase 041 declaran `precio`, `cantidad` y `descuento`; R trata la línea como **un objeto vectorial** y accede por posición. **Ese reflejo es el lenguaje entero.**
- `v[1]` **es el primero**, no el segundo (clase 089).
- `<-` **es la asignación idiomática**, aunque `=` funcione. Es herencia de S y de los teclados APL que tenían una tecla con la flecha.
- `[[1]]` **frente a** `[1]`: `strsplit` devuelve **una lista de vectores** —porque podría partir varias cadenas a la vez—, y `[[1]]` saca el primer elemento **desenvuelto**. La distinción entre `[` y `[[` es una de las primeras cosas que hay que entender en R (clase 099).
- `as.numeric` **convierte el vector entero de golpe**, no elemento a elemento. Otra vez: la unidad es el vector.

📚 Fuentes y bibliografía

- R for Data Science (https://r4ds.hadley.nz/) — **Hadley Wickham** y Garrett Grolemund; libre en línea, y el punto de entrada estándar al R moderno.
- Advanced R (https://adv-r.hadley.nz/) — el mismo autor; **el libro que explica el lenguaje por dentro**: entornos, evaluación no estándar y los sistemas de objetos. Imprescindible para las clases 087, 088 y 122.
- CRAN Task Views (https://cran.r-project.org/web/views/) — el mapa de paquetes por dominio.
- **W. N. Venables, B. D. Ripley**, *Modern Applied Statistics with S* — el clásico, todavía útil.
- Bioconductor (https://www.bioconductor.org/) — el ecosistema de bioinformática, con su propio ciclo de publicación.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Python · Julia · MATLAB · APL · Fortran

🪁 Racket — 1995

Racket empezó como un Scheme para enseñar y acabó convirtiéndose en algo que ningún otro lenguaje de esta lista es: **una plataforma para construir lenguajes**. Su lema —*"un lenguaje de programación programable"*— se toma literalmente: en Racket, **crear un lenguaje nuevo con su propia sintaxis es una operación normal**.

> 🎯 ***Por qué está en este programa***
>
> *Racket es un **primo de la familia Lisp** (Atlas).*
>
> *Aporta al programa la forma más extrema de **metaprogramación** (clase 122) y de **lenguaje de dominio** (clase 163): no ya macros que añaden construcciones, sino **lenguajes completos con otra sintaxis y otra semántica**, definidos como bibliotecas. Y aporta **los contratos en ejecución** (clase 118) en un lenguaje dinámico.*

Año	1995 (como PLT Scheme); **Racket** desde 2010; **Racket 8** actual
Autoría	**Matthias Felleisen** y el grupo PLT (Northeastern, Utah, Brown, otras)
Familia	Lisp; descendiente directo de Scheme
Paradigma	Multiparadigma; funcional por defecto, con OO, lógico y lo que se defina
Tipado	**Dinámico**, con **contratos** y con **Typed Racket** para tipado estático gradual
Memoria	Recolección de basura
Ejecución	Compilado a bytecode sobre **Chez Scheme** desde Racket 8
Estado	🟢 **Vivo**: enseñanza, investigación en lenguajes y proyectos propios

---

📜 Historia

PLT Scheme nació en **1995** con un objetivo docente: **DrScheme** —después DrRacket— era un entorno diseñado para principiantes, con **niveles de lenguaje** que iban activando características según avanzaba el curso, y con mensajes de error pensados para enseñar.

De ahí salió **How to Design Programs (HtDP)**, un método de enseñanza de programación basado en **recetas de diseño** —del tipo de datos se deriva la forma de la función— que se usa en decenas de universidades y que es, en sí mismo, una aportación pedagógica seria.

Pero lo que convirtió a Racket en lo que es fue una decisión posterior: **hacer del sistema de macros una plataforma para definir lenguajes enteros**. En **2010** el cambio de nombre marcó ese giro, y hoy el primer renglón de un fichero Racket **dice en qué lenguaje está escrito**:

```
#lang racket          ; el lenguaje por defecto
#lang typed/racket     ; con tipos estáticos
#lang scribble/base     ; ← un lenguaje de DOCUMENTOS
#lang datalog            ; ← Datalog, dentro de Racket
#lang mi-lenguaje         ; ← el tuyo
```

Y todos interoperan, porque todos compilan al mismo núcleo.

🏭 Dónde vive hoy

- **Enseñanza**: HtDP y *Bootstrap* —un currículo que enseña álgebra programando— en institutos y universidades.
- **Investigación en lenguajes de programación**: es la plataforma de referencia para prototipar lenguajes y sistemas de tipos.
- **Scribble**: la documentación de Racket está escrita en un lenguaje hecho en Racket para escribir documentación (clase 154).
- **Herramientas y aplicaciones propias** de su comunidad, y algunos usos industriales de nicho.

🧠 Lo que enseña: definir lenguajes como bibliotecas

Este es el concepto, y no lo tiene ningún otro lenguaje de este Atlas con esta profundidad:

```
#lang racket

(provide (rename-out [mi-app #%app])    ; ← redefinir cómo funciona LA APLICACIÓN
         (except-out (all-from-out racket) #%app))
```

Se puede redefinir qué significa llamar a una función, qué significa un literal numérico, o cómo se lee el texto del fichero. Un `#lang` es un módulo que exporta el lector y las macros del lenguaje.

Y de ahí salen cosas como `#lang scribble` —donde el texto plano es lo normal y el código va marcado, al revés que en un lenguaje de programación— o `#lang datalog`, que es Datalog de verdad, con su semántica lógica, dentro del mismo sistema.

Es la clase 163 llevada al límite: en lugar de incrustar un lenguaje ajeno, **el lenguaje anfitrión permite fabricar el incrustado**.

Y la segunda aportación son **los contratos** (clase 118), que Racket implementa mejor que ningún otro lenguaje dinámico:

```
(provide (contract-out
          [dividir (-> number? (and/c number? (not/c zero?)) number?)]))
```

El contrato se comprueba en la frontera del módulo, en ejecución — y cuando falla, **el mensaje dice quién tiene la culpa**: si el que llamó pasó un argumento malo, o si la función devolvió algo que no debía.

Esa atribución de culpa es la aportación técnica: en un lenguaje dinámico, saber **de qué lado de la frontera está el error** es la mitad del diagnóstico (clase 137).

🔄 Lo que se ha modernizado

- **Racket CS (8.0)**: el tiempo de ejecución reescrito sobre **Chez Scheme**, con mejoras grandes de rendimiento.
- **Typed Racket**: tipado estático gradual que **convive con módulos sin tipar**, insertando contratos en la frontera — una implementación muy estudiada del tipado gradual (clase 146).
- **Rhombus**: un lenguaje nuevo sobre la plataforma **con sintaxis de expresiones en lugar de paréntesis**, para atraer a quien rechaza la notación Lisp. Es la demostración definitiva de la tesis: **cambiar la sintaxis del lenguaje es un proyecto dentro del propio lenguaje**.
- `raco` como herramienta única: construir, empaquetar, documentar, probar y publicar.

⚙️ Cómo se ejecuta hoy

```
racket main.rkt < entrada.txt        # ejecutar
raco test main.rkt                    # pruebas (clase 139)
raco exe main.rkt                     # ejecutable autocontenido (clase 174)
raco pkg install <paquete>
```

🧪 El programa de la clase 041 en Racket

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
#lang racket

(define campos (string-split (read-line)))
(define nums (map string->number campos))
(define total (* (first nums) (second nums) (- 1 (third nums))))

(printf "Total: ~a\n" (real->decimal-string total 2))
```

Lo que hay que ver.

- `#lang racket` **en la primera línea** es la característica del lenguaje: **declara en qué lenguaje está escrito el fichero**. Cambiarlo por `#lang typed/racket` haría el mismo programa estáticamente tipado.
- `first`, `second`, `third` en lugar de `car`, `cadr` y `caddr` de Scheme: Racket eligió nombres legibles, coherente con su origen docente.
- `real->decimal-string` en lugar de una directiva de formato: es más explícito sobre lo que hace, otra vez la vocación pedagógica.
- `(* a b (- 1 c))` **con notación prefija** y aridad variable, como todo Lisp.
- **Y no hay mutación**: `define` liga nombres, no asigna variables (clase 102).

📚 Fuentes y bibliografía

- docs.racket-lang.org (https://docs.racket-lang.org/) — la documentación, escrita con Scribble; el *Racket Guide* y la *Reference* son excelentes.
- How to Design Programs (https://htdp.org/) — **Felleisen et al.**, libre en línea; **es un método de diseño de programas**, no un manual de sintaxis, y merece leerse aunque no se use Racket (clase 166).
- Beautiful Racket (https://beautifulracket.com/) — **Matthew Butterick**; cómo crear lenguajes con `#lang`, explicado paso a paso. Material directo de las clases 122 y 163.
- **Matthias Felleisen et al.**, *The Racket Manifesto* — la tesis de la programación orientada a lenguajes.
- Rhombus (https://racket-lang.org/rhombus/) — el proyecto de sintaxis alternativa.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Scheme · Common Lisp · Clojure · Datalog

🏢 RPG — 1959

El lenguaje que factura. Miles de empresas medianas —distribuidoras, fábricas, cadenas de retail, cooperativas de crédito— llevan su ERP sobre IBM i, y ese ERP está escrito en RPG. Es el sistema que nadie ve y que emite la factura.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: RPG se ejecuta hoy en decenas de miles de instalaciones de IBM i**, IBM mantiene el compilador ILE RPG, publica Technology Refreshes con novedades del lenguaje y hay un ecosistema de herramientas modernas (VS Code, Git, SQL embebido) creciendo alrededor.*
>
> *Entra porque **deja a la vista un concepto que ningún lenguaje del núcleo tiene**: el **ciclo del programa**, un bucle principal implícito que el propio runtime ejecuta por ti —lee un registro, procesa, escribe, repite— sin que aparezca un solo `while` en el código. Y un segundo concepto: **la base de datos como parte del lenguaje**, no como una biblioteca. Ver RPG obliga a preguntarse algo que en Python o Java nunca te planteas: ¿quién escribe el bucle principal de mi programa?*

Año	1959 (RPG); **RPG IV / ILE RPG** desde 1994; formato totalmente libre desde 2013
Autoría	**IBM**, originalmente para el IBM 1401
Familia	Negocios / generación de informes sobre datos
Paradigma	Imperativo con **ciclo implícito**; procedimental modular en ILE
Tipado	Estático y declarativo, con **decimal empaquetado** como tipo natural
Memoria	Estática; automática en procedimientos ILE
Ejecución	Compilado a objetos nativos de **IBM i** sobre Power
Estado	🟢 **Legacy empresarial muy activo** — ERP, retail, logística, manufactura, banca

---

📜 Historia

RPG nació en 1959 como *Report Program Generator*: no era un lenguaje de propósito general, era una herramienta para describir informes. Le decías qué campos tenía la tarjeta perforada de entrada, qué totales querías y cómo debía salir impreso, y él generaba el programa. La idea central —**declaras la forma, no el bucle**— venía de que el trabajo real era siempre el mismo: recorrer un fichero, acumular y romper por grupos.

De ahí sale el **ciclo del programa (*RPG cycle*)**, su rasgo más peculiar: el runtime lee el siguiente registro, ejecuta tu lógica de detalle, comprueba los cambios de nivel de control, imprime totales y vuelve a empezar. El programador rellenaba las especificaciones y el bucle lo ponía la máquina. Es una idea sorprendentemente moderna —programación declarativa, inversión de control— con sesenta años encima.

La evolución fue larga: **RPG II** (1969), **RPG III** (1978, con el System/38 y su base de datos integrada), **RPG/400** (AS/400, 1988) y **RPG IV** (1994), que trajo el **ILE** (*Integrated Language Environment*): procedimientos con parámetros, prototipos, módulos enlazables y llamadas entre lenguajes. En **2001** llegó el formato libre parcial (`/FREE`) para la lógica de cálculo, y en **2013** el formato **totalmente libre** (`**FREE`), que eliminó por fin las columnas.

El resultado es un lenguaje con dos caras muy distintas. El RPG de 1985 es un cuadrícula de columnas ilegible para un ojo moderno. El RPG libre de 2013 en adelante se parece bastante a un Pascal con tipos decimales.

🏭 Dónde sobrevive hoy

- **ERP a medida** en distribución, manufactura y mayoristas — muchísimo software vertical que nunca se empaquetó ni se vendió, escrito en casa a lo largo de treinta años.
- **Retail y logística**: gestión de inventario, pedidos, almacén, facturación.
- **Banca y cooperativas de crédito**, sobre todo de tamaño medio.
- **Administración y seguros** de escala regional.

La plataforma es **IBM i** (antes AS/400, iSeries, System i) sobre hardware **Power**. Es un sistema operativo con una propiedad rara: la compatibilidad binaria hacia atrás es casi total, de modo que un objeto compilado en un AS/400 de 1990 se ejecuta en un Power actual. Esa es, literalmente, la razón comercial de que RPG siga vivo.

🧠 Por qué no ha muerto

1. La base de datos es el sistema operativo. En IBM i, **Db2 for i** no es un servicio que instalas: es parte del sistema. Un programa RPG declara un fichero y sus campos aparecen como variables del programa. No hay ORM, no hay cadena de conexión, no hay serialización. La distancia entre "el registro en disco" y "la variable en memoria" es cero.

2. Compatibilidad binaria de décadas. IBM i separa el programa de la máquina mediante una capa de abstracción (*TIMI*). Cuando IBM cambió de procesador CISC a Power en los 90, los objetos de los clientes siguieron funcionando tras un simple guardar/restaurar. Un sistema que nunca te obliga a migrar es un sistema del que nunca migras.

3. Decimal empaquetado nativo. Igual que COBOL: `packed(11:2)` es aritmética decimal exacta, no punto flotante. Para facturar, es el tipo correcto.

4. Sigue recibiendo funcionalidad. SQL embebido, servicios web REST, JSON con `DATA-INTO` y `DATA-GEN`, integración con Git y VS Code. RPG moderno consume una API REST y devuelve JSON sin salir del lenguaje.

5. El conocimiento del negocio está dentro. El mismo argumento que en COBOL, con un agravante: en IBM i mucho de ese software es **artesanal y único de cada empresa**, sin proveedor al que pedir la versión nueva.

🔄 Lo que se ha modernizado

RPG es de los casos más llamativos: un lenguaje de 1959 al que se le ha añadido, en los últimos quince años, casi todo lo que hace falta para un backend actual.

- **Formato totalmente libre** (`**FREE`, 2013): desaparecen las columnas. El RPG que se escribe hoy se parece más a Pascal que a una hoja de cálculo.
- **JSON y XML con `DATA-INTO` y `DATA-GEN`**: sentencias del lenguaje que convierten un documento JSON en una estructura de datos RPG y al revés. Consumir una API REST desde RPG es hoy tarea de media pantalla de código.
- **SQL embebido** (`exec sql`): el acceso a datos moderno en IBM i es SQL escrito dentro del RPG, con las variables del programa como parámetros. Y **IBM i Services** expone el propio sistema operativo — trabajos, usuarios, ficheros IFS, red— como **vistas SQL** consultables.
- **Servicios web en las dos direcciones**: `HTTPAPI`, las funciones `SYSTOOLS` y el motor **IWS** (*Integrated Web Services*) permiten publicar un programa RPG como API REST o consumir una externa.
- **`ctl-opt` y procedimientos con prototipos** desde ILE: ámbito local, parámetros con tipo, módulos enlazables y llamadas entre RPG, C, COBOL y Java en el mismo programa.
- **Git, VS Code y CI**: la extensión **Code for IBM i** y el constructor **`bob`** trajeron control de versiones, revisión de código y construcción reproducible a una plataforma que durante décadas guardó los fuentes en miembros de ficheros físicos.
- **Node.js, Python y PHP corren en IBM i**, de modo que lo habitual hoy es un frontend moderno llamando a lógica RPG a través de una API.

⚙️ Cómo se ejecuta hoy

RPG **solo** se ejecuta sobre IBM i: no hay compilador libre para Linux ni para Windows. Esto es importante y conviene decirlo sin rodeos — es el lenguaje de esta lista con la barrera de entrada más alta.

```
# Desde una línea de comandos de IBM i (CL):
CRTBNDRPG PGM(MILIB/TOTVTA) SRCSTMF('/home/vlad/totvta.rpgle') DFTACTGRP(*NO)
CALL PGM(MILIB/TOTVTA) PARM('15000' '2' '0.10')
```

Herramientas actuales. RDi (*Rational Developer for i*, basado en Eclipse) es el IDE comercial clásico; **Code for IBM i**, la extensión de VS Code, es hoy la vía moderna y gratuita — edita, compila y depura contra un IBM i remoto. **`bob`** (*Better Object Builder*) da construcción reproducible con Git.

Para practicar sin un IBM i propio: los programas de acceso público de IBM (IBM i Developer / Tech Refresh sandboxes (https://www.ibm.com/products/ibm-i)) o una cuenta en un proveedor de *cloud* IBM i. No hay atajo local.

🍼 El programa de la clase 041 en RPG

> ⚠️ ***Contrato adaptado, y declarado.*** *En IBM i un programa no recibe sus datos por* `stdin`*: los recibe por* ***parámetros****, por un* ***fichero de base de datos*** *o por una* ***pantalla****. Mantener aquí la ficción de* `stdin` *sería falsear el lenguaje. Así que el contrato es el mismo —tres valores de entrada, un total con dos decimales de salida— pero la entrada llega como parámetros. Este código* ***no se verifica en CI****: no hay compilador RPG en los runners.*

```
**free
ctl-opt dftactgrp(*no) actgrp(*caller);

dcl-pi TOTVTA;
  precio    packed(11:2) const;
  cantidad  packed(11:2) const;
  descuento packed(5:4)  const;
end-pi;

dcl-s total  packed(15:2);
dcl-s salida char(40);

total = precio * cantidad * (1 - descuento);

salida = 'Total: ' + %char(total);
dsply salida;

*inlr = *on;
return;
```

Recorrido, línea a línea.

- `**free` en la primera línea, columna 1, declara el fuente en **formato totalmente libre**. Sin esa marca, el compilador vuelve al formato de columnas de 1959.
- `ctl-opt` son las opciones de control del programa. `dftactgrp(*no)` es obligatorio para compilar como ILE moderno y no como RPG/400 heredado; es la línea que todo el mundo olvida y que produce el primer error de compilación de cualquiera que empieza.
- `dcl-pi TOTVTA ... end-pi` es la **interfaz de programa**: los parámetros que recibe. `const` indica que no se modifican, lo que además permite pasar expresiones y no solo variables.
- `packed(11:2)` es **decimal empaquetado**: 11 dígitos en total, 2 de ellos decimales. Cada dígito ocupa medio byte. Es el tipo con el que se factura, y la razón es la misma que en COBOL: la aritmética es decimal, no binaria.
- `%char(total)` convierte el decimal a su representación en texto sin ceros a la izquierda. Es una **función incorporada** (*built-in*): las de RPG empiezan por `%` y hay decenas (`%trim`, `%scan`, `%subst`, `%date`).
- `dsply` muestra un mensaje. En un programa real la salida iría a una pantalla (fichero de display), a un informe o a un fichero; `dsply` es el equivalente al `print` de depuración.
- **`*inlr = *on` es la línea que hay que entender.** `*INLR` es el **indicador de último registro**. Ponerlo a `*ON` le dice al **ciclo del programa** que este es el final: cierra los ficheros, libera el almacenamiento y termina. Si se omite, el programa **queda residente en memoria** con sus variables intactas y la siguiente llamada continúa donde quedó. Es una fuente inagotable de errores fantasma para quien viene de otros lenguajes, y es la huella visible de que aquí el bucle principal no lo escribes tú.

Y así se veía en formato fijo, que es como está la mayor parte del código en producción:

```
     H DFTACTGRP(*NO)
     D TOTVTA          PR
     D   precio                      11P 2 CONST
     D   cantidad                    11P 2 CONST
     D   descuento                    5P 4 CONST
     D total           S             15P 2
      /FREE
        total = precio * cantidad * (1 - descuento);
        *inlr = *on;
      /END-FREE
```

La letra de la columna 6 decide qué es cada línea: **H** de control, **F** de ficheros, **D** de definiciones, **C** de cálculo, **O** de salida. Ese carácter es todo el "parser" del RPG clásico.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En RPG libre es…
`int x;` / `decimal x;`	`dcl-s x packed(11:2);`
`const`	`const` en la interfaz, o `dcl-c` para una constante nombrada
Firma de función	`dcl-pr` (prototipo) y `dcl-pi` (interfaz)
`struct`	`dcl-ds` — estructura de datos, con solapamiento de campos incluido
`if / else`	`if ... elseif ... else ... endif;`
`for`	`for i = 1 to n; ... endfor;`
`x.substring(a, b)`	`%subst(x : a : b)`
`SELECT ... FROM`	`exec sql select ... into :variable from tabla;` — SQL **dentro** del RPG
El bucle principal	**No lo escribes**: lo pone el ciclo del programa

⚠️ Errores comunes al leerlo

- **Olvidar `*inlr = *on`.** El programa no termina de verdad; se queda residente y la siguiente llamada arranca con el estado anterior.
- **Confundir `packed` con `float`.** `packed(11:2)` es exacto. `float(8)` existe, pero usarlo para dinero en RPG sería tan erróneo como en cualquier otro sitio.
- **Suponer ámbito local.** En RPG clásico todas las variables son globales al programa. Solo los **procedimientos** de ILE (`dcl-proc`) introducen variables locales de verdad.
- **Leer los indicadores como booleanos normales.** `*IN01` … `*IN99` son variables globales numeradas heredadas del ciclo, y el código antiguo las usa para todo. Un `*IN15` sin comentario puede significar cualquier cosa; es la deuda técnica característica del lenguaje.
- **Creer que un fuente `.rpgle` es RPG libre.** La extensión no lo determina: lo determina `**free` en la columna 1 de la primera línea.

📚 Fuentes y bibliografía

- IBM i — documentación de ILE RPG (https://www.ibm.com/docs/en/i/7.5?topic=languages-ile-rpg) — referencia del lenguaje y guía del programador, versión vigente.
- Code for IBM i (https://codefori.github.io/docs/) — la cadena de herramientas moderna sobre VS Code.
- RPG Café (IBM) (https://www.ibm.com/support/pages/rpg-cafe) — dónde IBM publica las novedades del lenguaje en cada *Technology Refresh*.

- **Jim Buck, Bryan Meyers, Dan Cruikshank**, *Programming in ILE RPG*, 5.ª ed., MC Press — el manual de referencia del RPG moderno.
- **Jim Martin**, *Free-Format RPG IV*, MC Press — específicamente sobre el paso del formato de columnas al libre, que es el problema real de cualquiera que herede código RPG.
- **Scott Klement**, artículos y bibliotecas de código abierto para IBM i — la referencia práctica de la comunidad para HTTP, JSON y cifrado desde RPG.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: COBOL · PL/I · JCL

💎 Ruby — 1995

Ruby se diseñó con un criterio poco frecuente y declarado por su autor: **la felicidad de quien programa**. De esa decisión salieron un lenguaje extraordinariamente expresivo, un marco web que cambió la industria, y buena parte de las prácticas ágiles que hoy se dan por supuestas.

> 🎯 ***Por qué está en este programa***
>
> *Ruby es un **primo de la familia de scripting dinámico** (Atlas), cuyos representantes en el núcleo son Python y PHP. **Es uno de los tres primos que se verifican en CI** en cada clase, junto a Perl y Lua.*
>
> *Aporta al programa el modelo de objetos más puro de todos los lenguajes de uso masivo —**todo es un objeto, incluidos los números y** `nil`, como en Smalltalk (clase 111)— y **los bloques**, que son la forma más elegante de la clase 121 en un lenguaje mayoritario.*

Año	1995; **1.9** con la máquina virtual YARV (2007); **3.0** en 2020
Autoría	**Yukihiro Matsumoto ("Matz")**, Japón
Familia	Scripting dinámico; con Smalltalk, Perl, Lisp y Eiffel dentro
Paradigma	Orientado a objetos puro, con fuerte componente funcional
Tipado	**Dinámico y fuerte**, con tipos gradual y opcionalmente comprobados (RBS)
Memoria	Recolección de basura generacional e incremental
Ejecución	Bytecode sobre YARV, con **JIT (YJIT)** desde 3.1
Estado	🟢 **Muy vivo** en web y automatización; Rails sigue siendo referencia

---

📜 Historia

Yukihiro Matsumoto quería, en 1993, un lenguaje de guion verdaderamente orientado a objetos. Perl era potente y no era OO; Python era OO y —en su opinión— no lo bastante. Su frase resume el criterio de diseño:

> ***"Ruby está diseñado para hacer felices a los programadores."***

Eso no es marketing: es una decisión de ingeniería con consecuencias. **Cuando hay varias formas de escribir algo, Ruby suele permitir todas**, y elige como idiomática la que se lee mejor en voz alta. Es lo contrario del "solo una forma obvia" de Python, y la comparación entre los dos es uno de los contrastes más instructivos de este Atlas.

Ruby fue popular en Japón durante años y desconocido fuera. **En 2004, David Heinemeier Hansson publicó Ruby on Rails**, y eso lo cambió todo: Rails llevó al mundo entero **la convención sobre la configuración**, los **generadores de código**, las **migraciones de base de datos** y una forma de trabajar que hoy está en Django, Laravel, Phoenix y media docena más.

Y con Rails llegó también una **cultura de pruebas** —RSpec, Cucumber, el desarrollo dirigido por pruebas como norma— que salió del ecosistema Ruby hacia todos los demás (clase 139).

Después vino la corrección del talón de Aquiles: **YARV** (1.9) sustituyó al intérprete de árbol, **Ruby 3.0 (2020)** cumplió el objetivo *"Ruby 3x3"* —tres veces más rápido que 2.0— y **YJIT**, escrito en Rust por Shopify, ha dado saltos grandes de rendimiento en producción real.

🏭 Dónde vive hoy

- **Aplicaciones web**: Rails mueve GitHub, Shopify, Basecamp, Airbnb y decenas de miles de productos.
- **Automatización e infraestructura**: **Chef**, **Puppet**, **Vagrant**, **Fastlane** — la generación anterior de herramientas de configuración (clase 171).
- **Herramientas de desarrollo**: **Homebrew**, el gestor de paquetes de macOS, está escrito en Ruby.
- **Guiones de administración**, como sucesor natural de Perl.
- **Y como lenguaje de DSL**: Rakefile, Gemfile, Podfile y Vagrantfile son **programas Ruby que parecen ficheros de configuración** (clase 163).

🧠 Lo que enseña: todo es un objeto, y los bloques

Uno, la pureza del modelo de objetos (clase 111):

```
5.class          # Integer
5.times { |i| puts i }
nil.to_a          # []      ← hasta nil es un objeto con métodos
Integer.ancestors # [Integer, Numeric, Comparable, Object, Kernel, BasicObject]
```

No hay tipos primitivos. En Java `int` no es un objeto y hay que envolverlo; en Ruby, como en Smalltalk, **no existe esa distinción** — y eso hace el lenguaje uniforme a costa de rendimiento, que es lo que YJIT recupera con las cachés de envío de la clase 152.

Dos, los bloques, que son la aportación más reconocible:

```
[1, 2, 3].map { |x| x * 2 }
File.open("datos.txt") do |f|      # ← el fichero se cierra SOLO al salir del bloque
  f.each_line { |l| puts l }
end
```

Un bloque es un trozo de código que se pasa a un método, y el método decide cuándo y cuántas veces ejecutarlo. Eso da a la vez **iteración** (clase 115) y **gestión de recursos** (clase 132) — el `File.open` con bloque es RAII sin destructores, y el `with` de Python y el `try-with-resources` de Java persiguen lo mismo con más ceremonia.

Y tres, las clases abiertas, que son su característica más potente y más peligrosa:

```
class String
  def gritar = upcase + "!"
end
"hola".gritar        # "HOLA!"
```

Se puede añadir un método a cualquier clase, incluidas las del sistema. Es lo que hace posible la expresividad de Rails — `2.days.ago`, `"texto".pluralize` — y también lo que produce el problema que la comunidad llama *monkey patching*: **dos bibliotecas que parchean lo mismo, y un fallo que nadie sabe de dónde viene** (clase 150).

> ***Ruby lo reconoció y dio una solución*** *(clase 146): los* ***refinamientos*** *(`refine` / `using`) limitan el parche al ámbito donde se activa. Es un buen ejemplo de un lenguaje corrigiendo su propio exceso sin quitar la característica.*

🔄 Lo que se ha modernizado

- **YJIT** (Rust, desde 3.1): mejoras de rendimiento de dos dígitos en aplicaciones Rails reales.
- **RBS y Sorbet**: tipos **en ficheros aparte** o en anotaciones, comprobados por herramientas — el tipado gradual de la clase 146 llegando a Ruby.
- **Ractor** (3.0): actores con memoria aislada para paralelismo real, porque **Ruby también tiene un bloqueo global del intérprete** (clase 135). Y **fibras** para concurrencia asíncrona (clase 134).
- **Emparejamiento de patrones** (`case/in`, 2.7) con desestructuración de arreglos y `hash`.
- `Bundler` **con** `Gemfile.lock`: es de 2010 y fue **uno de los primeros ficheros de bloqueo populares** (clase 143), un modelo que copiaron después npm, Cargo y Composer.

⚙️ Cómo se ejecuta hoy

```
ruby main.rb < entrada.txt        # el comando de la clase 041

bundle install                     # dependencias, con Gemfile.lock (clase 143)
rubocop && rspec                   # estilo y pruebas (clases 139 y 146)
ruby --yjit main.rb                # con el JIT activado
```

🧹 El programa de la clase 041 en Ruby

Esta versión **se ejecuta y se verifica en CI** contra el mismo `casos.json` que el núcleo.

```
precio, cantidad, descuento = STDIN.gets.split.map(&:to_f)
total = precio * cantidad * (1 - descuento)
puts format("Total: %.2f", total)
```

Lo que hay que ver.

- **Tres líneas, y ninguna sobra.** Es la versión más corta de las veinte de la clase 041, y esa densidad legible es exactamente lo que el lenguaje persigue.
- `&:to_f` **es un símbolo convertido en bloque**: el idioma más característico de Ruby. Equivale a `{ |x| x.to_f }`, y aprovecha que `to_f` **es un método del objeto** — otra vez la pureza del modelo.
- `split` **sin argumentos** parte por espacios en blanco, como `strings.Fields` en Go y `split ' '` en Perl.
- **La asignación múltiple** desestructura los tres valores, como Python.
- `format` **con** `%.2f` viene de C, vía Perl: la herencia se ve.

📚 Fuentes y bibliografía

- ruby-lang.org (https://www.ruby-lang.org/es/) — en español; y la documentación de la biblioteca (https://docs.ruby-lang.org/en/master/).
- Ruby Style Guide (https://rubystyle.guide/) — la guía comunitaria que RuboCop hace cumplir (clase 146).
- **Dave Thomas, Andy Hunt, Chad Fowler**, *Programming Ruby* — "el libro del pico"; la referencia clásica.

- **Sandi Metz**, *Practical Object-Oriented Design in Ruby* — uno de los mejores libros de diseño orientado a objetos que existen, en cualquier lenguaje (clases 149 y 166).
- **Paolo Perrotta**, *Metaprogramming Ruby*, Pragmatic — el modelo de objetos y las clases abiertas, explicados de verdad (clase 122).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Python · Perl · Smalltalk · Elixir · Lua

🦀 Rust — 2010

Rust resolvió un problema que llevaba cincuenta años abierto: **conseguir seguridad de memoria sin recolector de basura y sin perder rendimiento**. Lo hizo con una idea —**la propiedad y el préstamo**— que no existía en ningún lenguaje mayoritario, y que hoy están copiando todos.

> 🎯 ***Por qué está en este programa***
>
> ***Rust es uno de los diez lenguajes del núcleo*** *y, junto a Go, el* ***representante de la familia de sistemas*** *(Atlas).*
>
> *Aporta al programa el concepto que da nombre a una clase entera:* ***propiedad, movimiento y préstamo*** *(clase 081 y clase 132), y el de* ***concurrencia sin carreras garantizada por el compilador*** *(clase 136). Es el contrapunto exacto de C++ y el complemento de Go.*

Año	2010 (público); **1.0** en 2015; **ediciones** 2015, 2018, 2021, 2024
Autoría	**Graydon Hoare**, como proyecto personal (2006); adoptado por Mozilla
Familia	Sistemas; con influencia fuerte de **ML**, OCaml y **Cyclone**
Paradigma	Multiparadigma: imperativo, funcional y con genéricos por rasgos
Tipado	**Estático, fuerte, con inferencia** y tipos algebraicos
Memoria	**Sin recolector**: propiedad, préstamos y tiempos de vida comprobados al compilar
Ejecución	Compilado a nativo sobre **LLVM**; sin tiempo de ejecución significativo
Estado	🟢 **En crecimiento fuerte**; en el núcleo de Linux, Windows y Android

---

📜 Historia

Graydon Hoare empezó Rust en **2006** como proyecto personal. La anécdota de origen es buena: el ascensor de su edificio se había averiado por un fallo de software, y le pareció absurdo que en 2006 siguiéramos escribiendo software de sistemas en lenguajes que permiten corromper memoria.

Mozilla lo adoptó en **2009** con un objetivo concreto: **Servo**, un motor de navegador paralelo. Porque el problema real era ese — **paralelizar el renderizado de una página en C++ era inviable** por las carreras de datos, y hacía falta un lenguaje donde eso fuera seguro.

Rust cambió mucho antes de la **1.0 (2015)**: llegó a tener recolector de basura, hilos ligeros y punteros con sigilo, y **todo eso se quitó**. Lo que quedó fue el sistema de propiedad, que es la aportación real.

El sistema de ediciones (2018, 2021, 2024) resuelve un problema de la clase 143 de forma elegante: **se pueden hacer cambios incompatibles de sintaxis sin romper nada**, porque cada *crate* declara su edición y **las ediciones interoperan**. Es una idea que merece copiarse.

Y desde **2021**, tras la reestructuración de Mozilla, el lenguaje lo gobierna la **Rust Foundation**, con AWS, Google, Microsoft, Meta y Huawei entre sus miembros.

🏭 Dónde vive hoy

- **Núcleos de sistemas operativos**: **Linux** admite controladores en Rust desde 2022; **Windows** y **Android** tienen componentes reescritos, con reducción medible de vulnerabilidades (clase 153).
- **Infraestructura de red**: Cloudflare (Pingora), Discord, Dropbox, AWS (Firecracker).
- **Herramientas de desarrollo rapidísimas**: `ripgrep`, `fd`, `bat`, y toda la generación nueva de herramientas de JavaScript y Python — `swc`, `ruff`, `uv`, `biome` — **escritas en Rust** (clase 167).
- **WebAssembly**: es el lenguaje con mejor soporte del ecosistema (clase 162).
- **Sistemas embebidos y criptografía**, donde la ausencia de recolector es un requisito.
- **Bases de datos**: TiKV, InfluxDB 3, Datafusion.

🧠 La idea: propiedad, préstamo y tiempos de vida

Es el concepto que hay que llevarse de esta ficha, y se resume en tres reglas:

```
let a = String::from("hola");
let b = a;              // MOVIMIENTO: ahora b es el dueño
// println!("{a}");      // ✗ error de compilación: a ya no vale

let c = &b;             // PRÉSTAMO inmutable: se puede tener muchos
let d = &mut b;          // PRÉSTAMO mutable: solo UNO, y ninguno inmutable a la vez
```

Las tres reglas del comprobador de préstamos:

1. **Cada valor tiene exactamente un dueño.**
2. **Puede haber muchas referencias inmutables, o una mutable — nunca las dos cosas.**
3. **Ninguna referencia puede vivir más que el valor al que apunta.**

Y de esas tres reglas salen, gratis, tres garantías que la clase 136 desarrolla:

- **No hay uso después de liberar**, porque la referencia no sobrevive al valor.
- **No hay doble liberación**, porque solo hay un dueño.
- **Y no hay carreras de datos**, porque no puede haber una escritura y una lectura simultáneas.

La tercera es la que asombra: el mismo mecanismo que gestiona la memoria **elimina las carreras de datos en tiempo de compilación**, sin coste en ejecución. Es lo que Mozilla necesitaba para Servo, y es la razón por la que existe el lenguaje.

> ***Y el precio hay que decirlo, porque es real: la curva de aprendizaje es empinada**. El comprobador de préstamos rechaza programas correctos que no sabe demostrar, y aprender a estructurar el código para que pase **es el trabajo de las primeras semanas**. A cambio, quien escribe Rust deja de depurar corrupciones de memoria (clase 153) — pero el cambio no es gratis.*

🔄 Lo que se ha modernizado

- `async` / `await` (2019) con tiempos de ejecución intercambiables (Tokio, async-std), y el ecosistema asíncrono maduro.
- **Edición 2024** con `gen`, mejoras del comprobador de préstamos y `impl Trait` en más sitios.
- **Rust en el núcleo de Linux**, con las abstracciones seguras sobre las interfaces C (clase 156).

- **Herramientas de verificación**: **Kani**, **Creusot**, **Prusti** — demostración formal sobre Rust, persiguiendo lo que SPARK tiene maduro (clase 164).
- `cargo` **como referencia**: gestor de paquetes, sistema de construcción, pruebas, documentación, formateo y análisis **en una sola herramienta**, con fichero de bloqueo (clase 143). Es el estándar con el que se comparan los demás ecosistemas.

⚙️ Cómo se ejecuta hoy

```
rustc main.rs -O && ./main < entrada.txt      # el camino de la clase 041
cargo run --release

cargo fmt && cargo clippy -- -D warnings       # calidad (clase 146)
cargo test                                      # pruebas, integradas
cargo doc --open                                  # documentación desde los comentarios (clase 154)
```

🍼 El programa de la clase 041 en Rust

```
use std::io::Read;

fn main() {
    let mut entrada = String::new();
    std::io::stdin().read_to_string(&mut entrada).unwrap();
    let campos: Vec<&str> = entrada.split_whitespace().collect();

    // Rust: inmutable por defecto (`let`), tipos explícitos, conversión con `as`.
    let precio_unitario: f64 = campos[0].parse().unwrap();
    let cantidad: i64 = campos[1].parse().unwrap();
    let descuento: f64 = campos[2].parse().unwrap();

    let subtotal = precio_unitario * cantidad as f64;
    let total = subtotal * (1.0 - descuento);

    println!("Total: {total:.2}");
}
```

Lo que hay que ver.

- `let` **es inmutable por defecto**; para poder modificar hay que escribir `let mut`. Es lo contrario de casi todos los lenguajes de esta lista, y la clase 102 explica por qué importa: **la inmutabilidad por defecto convierte cada mutación en una decisión visible**.
- `Vec<&str>` **es la clave del sistema de propiedad en este programa**: los campos **no son copias**, son **referencias prestadas** a la cadena `entrada`. Por eso `entrada` tiene que seguir viva mientras se usen — y el compilador lo comprueba. En Java o Go eso sería una copia.
- `.unwrap()` **es una decisión, no un descuido**: `parse` devuelve `Result`, y **el lenguaje obliga a tratarlo** (clase 116). `unwrap` dice "si falla, aborta" — aceptable en un ejemplo, y en producción se escribiría `?` o un `match`.
- `cantidad as f64` **es explícito**, igual que en Go: Rust no convierte números sin permiso.
- `{total:.2}` **interpola la variable directamente** en la cadena de formato, y el formato se comprueba **en compilación** — como `std::format` de C++20 (clase 142).

📚 Fuentes y bibliografía

- *The Rust Programming Language* (https://doc.rust-lang.org/book/) — "el libro", oficial, libre y uno de los mejores textos de introducción de cualquier lenguaje.
- Rust by Example (https://doc.rust-lang.org/rust-by-example/) y Rustlings (https://github.com/rust-lang/rustlings) — para practicar el comprobador de préstamos.

- El Rustonomicon (https://doc.rust-lang.org/nomicon/) — para cuando haga falta `unsafe`; explica lo que el compilador estaba garantizando.
- **Jon Gjengset**, *Rust for Rustaceans*, No Starch Press — el libro para después del primero; tipos, rasgos, `async` y `unsafe` a fondo.
- **Steve Klabnik, Carol Nichols**, *The Rust Programming Language*, 2.ª ed. impresa, No Starch Press.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: C++ · Go · Zig · OCaml · Ada

🔺 Scala — 2004

Scala es el intento más serio de **fundir la orientación a objetos con la programación funcional en un solo sistema de tipos coherente**. Lo consiguió — y el precio fue un lenguaje que se puede escribir de muchas formas distintas, lo que produjo a la vez ideas que hoy están en todas partes y una reputación de complejidad.

> 🎯 ***Por qué está en este programa***
>
> *Scala es un **primo de la familia JVM** (Atlas), cuyo representante en el núcleo es Java.*
>
> *Aporta al programa el sistema de tipos más expresivo de todos los lenguajes de esta lista, y en particular dos conceptos: **el emparejamiento de patrones sobre tipos algebraicos** (clase 100 y clase 116) y **las colecciones inmutables persistentes** (clase 102). Y aporta un caso de estudio de la clase 175: **Scala 3 rompió la compatibilidad para simplificarse**, y eso tiene un coste.*

Año	2004; **2.x** durante quince años; **Scala 3** en 2021
Autoría	**Martin Odersky**, EPFL — coautor de los genéricos de Java y del compilador javac
Familia	JVM; con influencia de Java, Haskell, ML y Erlang
Paradigma	**OO y funcional a la vez**, sin costuras
Tipado	**Estático muy expresivo**: tipos superiores, implícitos, dependientes de ruta
Memoria	La de la JVM
Ejecución	Bytecode JVM; también **Scala.js** y **Scala Native**
Estado	🟢 **Sólido** en datos y en sistemas distribuidos; en descenso frente a Kotlin

---

📜 Historia

Martin Odersky no es un desconocido en esta historia: **escribió el compilador javac** y **diseñó los genéricos de Java** con Philip Wadler. Sabía exactamente qué limitaciones tenía la plataforma y por qué.

En **2004** publicó Scala —*scalable language*— con una tesis: **la orientación a objetos y la programación funcional no son opuestas, y se pueden unificar**. En Scala **todo es un objeto** —como en Smalltalk— y **todas las funciones son valores** —como en Lisp—, y las dos cosas encajan en el mismo sistema de tipos.

Y su influencia fue enorme, sobre todo por dos proyectos:

- **Apache Spark (2012)**, escrito en Scala, que se convirtió en el estándar del procesamiento de datos masivos.
- **Akka**, que llevó el **modelo de actores** de Erlang a la JVM (clase 133).

También aportó al debate del lenguaje: **las lambdas de Java 8 llegaron en parte por la presión de Scala**, y su biblioteca de colecciones influyó en la de Kotlin y en la de Java.

Scala 3 (2021) fue una reescritura profunda —basada en la teoría DOT y en el compilador Dotty— que **simplificó** el lenguaje: sustituyó los `implicit`, que eran su rincón más difícil, por `given` / `using`; añadió sintaxis por sangría opcional; y unificó los tipos. **Rompió compatibilidad**, y la migración del ecosistema ha sido lenta — la lección de la clase 175 que también vivieron Python 3 y D.

🏭 Dónde vive hoy

- **Datos a gran escala**: **Spark**, Kafka Streams, Flink, Delta Lake.
- **Sistemas distribuidos**: **Akka** y Pekko, en telecomunicaciones, finanzas y videojuegos en línea.
- **Servicios financieros**: bancos de inversión con requisitos de corrección alta.
- **Empresas conocidas**: Twitter (histórico), LinkedIn, Netflix, Zalando, Disney Streaming.
- **Y en investigación**: por su sistema de tipos, es una plataforma habitual de experimentación.

🧠 Lo que enseña: patrones y tipos algebraicos

El emparejamiento de patrones sobre tipos sellados es la aportación más transferible:

```
sealed trait Forma
case class Circulo(r: Double) extends Forma
case class Rectangulo(a: Double, b: Double) extends Forma

def area(f: Forma): Double = f match
  case Circulo(r)        => math.Pi * r * r
  case Rectangulo(a, b)  => a * b
  // ← si falta un caso, el COMPILADOR AVISA
```

La exhaustividad comprobada es lo que hace valioso el patrón: **al añadir una forma nueva, el compilador señala todos los sitios que hay que actualizar** (clase 100). Es lo que Rust, Kotlin, Java 21, C# y Swift han adoptado después, y viene de ML y de Haskell.

Y las **colecciones inmutables persistentes**:

```
val a = List(1, 2, 3)
val b = 0 :: a        // ← b es una lista nueva... que COMPARTE la estructura de a
```

No se copia nada: las estructuras persistentes comparten las partes que no cambian (clase 102). Eso hace la inmutabilidad asequible, y es la base de las colecciones de Clojure y de las bibliotecas inmutables de JavaScript.

> ***Y el rincón difícil merece nombrarse con honestidad**: los **implícitos** de Scala 2 —valores que el compilador inserta buscándolos por tipo— son a la vez lo más potente del lenguaje y la razón principal de su fama. Permiten clases de tipos al estilo de Haskell, conversiones automáticas y extensión de tipos ajenos; y también producen código donde **no se ve de dónde sale lo que se ejecuta**. **Scala 3 los rediseñó** con `given` / `using`, precisamente para hacerlos explícitos.*

🔄 Lo que se ha modernizado

- **Scala 3**: `given` / `using` en lugar de `implicit`, uniones e intersecciones de tipos, `enum`, métodos de extensión y sintaxis por sangría opcional.
- **Scala Native** y **Scala.js**: el mismo lenguaje a binario nativo y al navegador.

- **Efectos tipados**: **ZIO** y **Cats Effect** modelan los efectos secundarios en el tipo —la idea de Haskell llevada a la práctica industrial (clase 118).
- `scala-cli`: ejecutar un fichero sin proyecto, con dependencias declaradas en un comentario — arranque de fricción cero (clase 167).
- **Y el compilador más rápido** de Scala 3, aunque sigue siendo lento comparado con Java o Kotlin.

⚙️ Cómo se ejecuta hoy

```
scala-cli run main.scala < entrada.txt      # sin proyecto ni configuración
scalac Venta.scala && scala Venta

sbt compile test                              # el sistema de construcción clásico
sbt scalafmtCheck scalafixAll                  # estilo y correcciones (clase 146)
```

🍼 El programa de la clase 041 en Scala

```
object Venta extends App {
  val Array(precio, cantidad, descuento) =
    scala.io.StdIn.readLine().split(" ").map(_.toDouble)
  val total = precio * cantidad * (1 - descuento)
  println(f"Total: $total%.2f")
}
```

Lo que hay que ver.

- `val Array(a, b, c) = ...` **no es una asignación múltiple: es un emparejamiento de patrones.** Se compara el valor con el patrón `Array(_, _, _)` y se extraen los tres. **Si la línea trajera cuatro campos, esto fallaría en ejecución** — y esa es la diferencia con la desestructuración de Kotlin o Python.
- `_.toDouble` es una función anónima con parámetro implícito: el guion bajo **es** el argumento. Es de los idiomas más reconocibles del lenguaje, y también de los que más confunden al principio.
- `f"Total: $total%.2f"` es un interpolador de cadenas **comprobado en compilación**: si el formato no encaja con el tipo, no compila (clase 142). El prefijo `f` selecciona el interpolador, y se pueden definir otros — es una característica del lenguaje, no de la biblioteca.
- `val`**, otra vez inmutable por defecto**, como en Kotlin y Rust (clase 102).

📚 Fuentes y bibliografía

- docs.scala-lang.org (https://docs.scala-lang.org/) — el tour del lenguaje y la guía de migración a Scala 3 (https://docs.scala-lang.org/scala3/guides/migration/).
- **Martin Odersky, Lex Spoon, Bill Venners**, *Programming in Scala*, 5.ª ed. — el libro del autor; la referencia.
- **Noel Welsh, Dave Gurnell**, *Essential Scala* / *Scala with Cats* — libres en línea; excelentes para entender los tipos y los efectos.
- Rock the JVM (https://rockthejvm.com/) — cursos y artículos con muy buenas explicaciones de tipos avanzados.
- **Odersky et al.**, *The Essence of Dependent Object Types* — el fundamento teórico de Scala 3, para quien quiera llegar al final.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Java · Kotlin · Haskell · OCaml · Clojure

○ Scheme — 1975

Scheme es el Lisp minimalista: **su especificación completa cabe en unas cincuenta páginas**, frente a las mil de Common Lisp. Fue durante décadas el lenguaje con el que se enseñaba a programar en el MIT, y de él salieron las continuaciones, la recursión de cola garantizada y las macros higiénicas.

> 🎯 ***Por qué está en este programa***
>
> *Scheme es un **primo de la familia Lisp** (Atlas), que no tiene representante en el núcleo — Common Lisp y AutoLISP son sus parientes entre los lenguajes vivos.*
>
> *Aporta al programa tres conceptos concretos: **las continuaciones de primera clase** (clase 127), **la recursión de cola garantizada por el estándar** (clase 083) y **las macros higiénicas** (clase 122). Y aporta la demostración de que **un lenguaje puede ser minúsculo y completo a la vez**.*

Año	1975; **R5RS** (1998) el más querido; **R7RS** (2013) el vigente
Autoría	**Guy L. Steele** y **Gerald Jay Sussman**, MIT
Familia	Lisp; con el alcance léxico del cálculo lambda de Church
Paradigma	Multiparadigma con base funcional; sin imponer nada
Tipado	**Dinámico y fuerte**; numéricos exactos y con torre de tipos
Memoria	Recolección de basura
Ejecución	Depende de la implementación: intérprete, bytecode o nativo
Estado	🟢 **Vivo**: enseñanza, investigación y **Guile** como lenguaje de extensión de GNU

📜 Historia

En **1975**, **Steele** y **Sussman** en el MIT escribieron un intérprete pequeño para estudiar el **modelo de actores** de Carl Hewitt. Y descubrieron algo que cambió el diseño de lenguajes: **al implementar los actores con cierres léxicos, actores y funciones resultaban ser lo mismo**.

De ahí salieron **los *Lambda Papers*** —una serie de informes con títulos como *Lambda: The Ultimate Imperative*— que demostraron que **con lambdas y alcance léxico se pueden construir todas las estructuras de control**: bucles, condicionales, excepciones y saltos.

Y de ahí las decisiones de Scheme:

- **Alcance léxico**, cuando Lisp usaba dinámico (clase 088).
- **Un solo espacio de nombres** para funciones y valores —lo que se llama *Lisp-1*—, a diferencia de Common Lisp.
- **Recursión de cola obligatoria en el estándar**: un bucle escrito como recursión **no consume pila** (clase 083).
- **Y minimalismo radical**: si algo se puede construir con lo que hay, no entra en el lenguaje.

SICP —*Structure and Interpretation of Computer Programs*, de Abelson y Sussman, 1985— usó Scheme para enseñar programación en el MIT durante veinte años, y es probablemente **el libro de informática más influyente jamás escrito**. Está libre en línea.

Y **R6RS (2007)** provocó la única crisis del lenguaje: era grande y prescriptivo, la comunidad se dividió, y **R7RS (2013)** volvió al minimalismo con un núcleo pequeño y módulos opcionales.

🏭 Dónde vive hoy

- **GNU Guile**: es **el lenguaje de extensión oficial del proyecto GNU** (clase 163) — GnuCash, GNU Make con Guile, Guix.
- **GNU Guix**: un gestor de paquetes y distribución **completamente definido en Scheme**, con construcciones reproducibles bit a bit (clase 144). Es uno de los usos más serios que existen.
- **Enseñanza**: sigue en cursos de introducción y de compiladores.
- **Racket**, que empezó como Scheme y creció hasta ser otra cosa.
- **Sistemas embebidos**: Chibi Scheme y otras implementaciones diminutas.
- **Y Chez Scheme**, uno de los compiladores más rápidos que existen para un lenguaje dinámico.

🧠 Lo que enseña: continuaciones y recursión de cola

Uno, `call/cc` —capturar la continuación—, que es la construcción de control más potente que existe (clase 127):

```
(define (buscar lista objetivo)
  (call/cc
    (lambda (salir)                    ; ← 'salir' ES el resto del cálculo
      (for-each (lambda (x)
                  (if (equal? x objetivo) (salir #t)))
                lista)
      #f)))
```

`call/cc` da como valor "todo lo que falta por hacer", y llamarlo salta ahí. Con eso se pueden construir —**dentro del lenguaje, sin sintaxis nueva**— excepciones, generadores, corrutinas, hilos cooperativos y vuelta atrás.

Es la demostración más pura de la tesis de los *Lambda Papers*, y es la razón de que Seaside pudiera hacer aplicaciones web con continuaciones (clase 168).

Dos, la recursión de cola garantizada:

```
(define (contar n acc)
  (if (= n 0) acc (contar (- n 1) (+ acc 1))))   ; ← NO crece la pila
(contar 10000000 0)                               ; funciona
```

El estándar obliga a que una llamada en posición de cola no consuma pila (clase 083). Eso hace que **el bucle y la recursión sean lo mismo**, y por eso Scheme no necesita `for` ni `while`.

Y tres, las macros higiénicas:

```
(define-syntax mi-si
  (syntax-rules ()
    ((_ c t e) (cond (c t) (else e)))))
```

"Higiénicas" significa que las variables de la macro no capturan las de quien la usa —el problema clásico de las macros de Common Lisp y del preprocesador de C (clase 123)—. Scheme lo resolvió con `syntax-rules`, y es la solución que después adoptaron Rust y Racket.

🔄 Lo que se ha modernizado

- **R7RS-small y R7RS-large**: núcleo pequeño más una biblioteca por capas, resolviendo la crisis de R6RS.
- **Guile 3** con JIT, y **Chez Scheme** como compilador de referencia por rendimiento.
- **Hoot**: **Guile compilado a WebAssembly con WasmGC** (clase 162).
- **Guix**: la aplicación más ambiciosa del lenguaje, con reproducibilidad verificable.
- **Y** `syntax-case`, macros higiénicas con toda la potencia procedural.

⚙️ Cómo se ejecuta hoy

```
guile main.scm < entrada.txt          # GNU Guile
chez --script main.scm                 # Chez Scheme, muy rápido
csi -s main.scm                         # Chicken Scheme (compila a C)
```

🧪 El programa de la clase 041 en Scheme

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
(use-modules (ice-9 rdelim))

(let* ((linea   (read-line))
       (campos  (string-split linea #\space))
       (nums    (map string->number campos))
       (total   (* (car nums) (cadr nums) (- 1 (caddr nums)))))
  (format #t "Total: ~,2f~%" total))
```

Lo que hay que ver.

- `(* a b c)` **toma tres argumentos**: en Lisp los operadores son funciones normales y **no tienen aridad fija**. Es la notación prefija, y es lo que hace que el programa sea una estructura de datos (clase 123).
- `car`, `cadr`, `caddr` —primero, segundo, tercero— son los nombres históricos de Lisp, de los registros del IBM 704 de 1958. **Sobreviven cincuenta años después de que el hardware desapareciera**, que es un buen ejemplo de la clase 154.
- `let*` **liga en secuencia**, permitiendo que cada nombre use los anteriores —a diferencia de `let`, que liga en paralelo—. Esa distinción explícita es muy de Scheme: **lo que en otros lenguajes es implícito, aquí se elige**.
- `~,2f` **es la directiva de formato** con dos decimales, la misma familia que `%.2f`.
- **Y no hay ni una asignación**: todo son enlaces (clase 102).

📚 Fuentes y bibliografía

- SICP (https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/index.html) — **Abelson y Sussman**, libre en línea; y las clases en vídeo de 1986 (https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/), que siguen siendo excelentes.
- The Scheme Programming Language (https://www.scheme.com/tspl4/) — **R. Kent Dybvig**, libre en línea; la referencia práctica.
- Estándares R7RS (https://small.r7rs.org/) — la especificación entera cabe en un rato de lectura.
- Manual de GNU Guile (https://www.gnu.org/software/guile/manual/) — para el uso real como lenguaje de extensión.
- **Steele y Sussman**, *Lambda: The Ultimate…* — los informes originales; históricos y todavía reveladores.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Common Lisp · Racket · Clojure · Emacs Lisp · AutoLISP

🎈 Smalltalk — década de 1970

El lenguaje del que salió casi todo. La orientación a objetos tal como la entendemos, el patrón MVC, las pruebas unitarias, el refactoring como práctica, las metodologías ágiles y hasta la interfaz gráfica con ventanas y ratón: todo eso se cocinó en el mismo sitio, alrededor de Smalltalk, en Xerox PARC.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Smalltalk se ejecuta hoy en producción**, en banca, seguros, trading y logística. Cincom desarrolla y vende **VisualWorks** y **ObjectStudio**, GemTalk mantiene **GemStone/S** (base de datos de objetos usada en sistemas financieros) y **Pharo** publica versiones nuevas cada año con una comunidad activa.*
>
> *Entra porque **lleva al extremo el concepto central de la Parte 7**: aquí la orientación a objetos no es una forma de organizar el código, es **la única forma que existe**. No hay tipos primitivos fuera del sistema de objetos, no hay operadores, y —esto es lo que sorprende— **no hay estructuras de control**. `if`, `while` y `for` no son sintaxis: son mensajes enviados a objetos. Ver un lenguaje donde `ifTrue:` es un método de la clase `Boolean` es lo que hace que "todo es un objeto" deje de ser un eslogan y pase a ser una consecuencia comprobable.*

Año	Smalltalk-71/72/76 en Xerox PARC; **Smalltalk-80** es la versión difundida
Autoría	**Alan Kay**, **Dan Ingalls**, **Adele Goldberg** y el Learning Research Group de PARC
Familia	Orientada a objetos pura, con influencia de Simula y Lisp
Paradigma	**OO puro**, basado en paso de mensajes
Tipado	**Dinámico** y fuerte; sin declaraciones de tipo
Memoria	Recolector de basura (Smalltalk fue pionero en GC generacional)
Ejecución	Máquina virtual con **imagen** de estado persistente; JIT en las VM modernas
Estado	🟡 **Nicho vivo** — banca, seguros, trading, telecomunicaciones, investigación

---

📜 Historia

Alan Kay llegó a Xerox PARC a comienzos de los 70 con una pregunta que no era de ingeniería sino casi filosófica: si los ordenadores iban a ser personales, ¿cómo debería ser un sistema que un niño pudiera entender y modificar? Su respuesta —influida por Simula, por Lisp, por LOGO y por la biología celular— fue construir un sistema donde todo estuviera hecho de **objetos autónomos que se comunican enviándose mensajes**, sin poder tocar el interior de los demás.

Kay ha insistido después en que la palabra "objeto" desvió la atención: lo importante era el **paso de mensajes**, no las clases. La distinción explica por qué Smalltalk se siente distinto de Java o C++, que tomaron las clases y dejaron el resto.

De ese laboratorio salió, en pocos años, una cantidad desproporcionada de lo que hoy es normal:

- La **interfaz gráfica** con ventanas superpuestas, iconos, menús y ratón, que Steve Jobs vio en su visita de 1979 y llevó al Lisa y al Macintosh.
- El patrón **MVC** (*Model-View-Controller*), formulado por **Trygve Reenskaug** durante su estancia en PARC en 1979.

- **SUnit**, el marco de pruebas unitarias que **Kent Beck** escribió en Smalltalk y que, portado a Java como **JUnit**, definió cómo se prueban hoy casi todos los lenguajes.
- El **refactoring** como disciplina con herramienta: el *Refactoring Browser* de Ralph Johnson y John Brant fue el primero que automatizó transformaciones de código con seguridad.
- La **programación extrema (XP)** y buena parte del movimiento ágil, nacidas en la comunidad Smalltalk de los 90 alrededor del proyecto C3 de Chrysler.
- Y, en gran medida, el libro ***Design Patterns*** de la Banda de los Cuatro, cuyos ejemplos y vocabulario vienen de esta tradición.

Smalltalk-80 fue la versión que salió de PARC al mundo, documentada en el célebre "Libro Azul" de Goldberg y Robson. Después vinieron las implementaciones comerciales —ParcPlace, Digitalk, IBM VisualAge— y, cuando el mercado se movió a Java a finales de los 90, la retirada a los nichos donde el software ya estaba escrito y funcionaba demasiado bien para tirarlo.

🏭 Dónde sobrevive hoy

- **Banca y finanzas**: sistemas de riesgo y valoración de derivados. El caso más documentado es **Kapital**, la plataforma de riesgo de JPMorgan, escrita en Smalltalk y mantenida durante décadas.
- **Seguros**: motores de tarificación y gestión de pólizas.
- **Logística y manufactura**: planificación y control de producción.
- **Telecomunicaciones**: sistemas de provisión y facturación.
- **Investigación y enseñanza**: **Pharo** y **Squeak** son entornos vivos; Squeak sostiene además **Scratch**, cuyo primer motor se escribió en Smalltalk.

🧠 Por qué no ha muerto

1. La imagen: un sistema vivo, no un fichero fuente. En Smalltalk no compilas y ejecutas: entras en un **entorno que ya está corriendo** y lo modificas. La *imagen* es una instantánea completa del estado del sistema —objetos, clases, ventanas abiertas, pilas de ejecución— que se guarda y se restaura. Puedes detener un proceso en producción, abrir el depurador, **cambiar el método que falló, y continuar la misma llamada** sin reiniciar. Ningún lenguaje del núcleo ofrece eso.

2. Uniformidad absoluta. Seis palabras reservadas y una sola regla: enviar mensajes. La sintaxis completa cabe en una postal. Esa uniformidad es la que permite herramientas tan potentes: si todo es un objeto y todo es un mensaje, el navegador de clases, el depurador y el refactorizador pueden razonar sobre cualquier cosa.

3. Productividad demostrada en dominios complejos. En modelado de negocio con reglas intrincadas —riesgo, seguros, planificación—, el ciclo de exploración interactiva de Smalltalk sigue siendo competitivo. Por eso los sistemas que quedaron no se han reescrito.

4. Reescribir tiene un coste desproporcionado. Décadas de reglas modeladas como objetos vivos, sin esquema externo que documente el dominio.

> ***Y una honestidad necesaria:*** *la imagen también es su mayor problema. Es difícil de meter en un flujo moderno de Git, revisión de código y CI, porque el "fuente" no es un árbol de ficheros sino un estado binario. Las herramientas actuales (Tonel, Iceberg, Metacello en Pharo) resuelven bastante, pero la fricción cultural con el resto de la industria es real y explica parte de su declive.*

🔄 Lo que se ha modernizado

- **Git de verdad.** El problema histórico —el código vive en una imagen binaria— está resuelto: **Tonel** guarda cada clase y método como ficheros de texto legibles, e **Iceberg** integra Pharo con Git y GitHub. Hoy un proyecto Smalltalk se revisa por *pull request* como cualquier otro.
- **Pharo publica una versión al año**, con máquina virtual JIT, compilación *ahead-of-time* experimental, soporte de ARM64 (Apple Silicon y Raspberry Pi) y 64 bits.
- **Web moderna**: **Seaside** (aplicaciones con estado sobre continuaciones), **Teapot** y **Zinc** para APIs REST y clientes HTTP, y **Pharo JS** para compilar a JavaScript.
- **Integración continua**: los proyectos Pharo se construyen en GitHub Actions con imágenes descargadas desde un guion, lo que hace posible el CI sobre un lenguaje basado en imágenes.
- **Metacello** como gestor de dependencias y versiones de proyecto.
- **Glamorous Toolkit**, construido sobre Pharo, propone el "entorno moldeable": herramientas de análisis y visualización creadas a medida del sistema que estás estudiando. Es de las ideas más originales que ha salido de la comunidad en la última década.

⚙️ Cómo se ejecuta hoy

Pharo (https://pharo.org/) es hoy la puerta de entrada: libre, activo y con excelente material de aprendizaje.

```
# Descargar el lanzador o la imagen desde pharo.org
./pharo Pharo.image eval "3 + 4"
# 7

# Ejecutar un script desde fichero
./pharo Pharo.image st total.st
```

Implementaciones actuales: Pharo (libre, la más activa), **Squeak** (libre, descendiente directa del Smalltalk-80 original con parte del equipo de PARC detrás), **Cincom VisualWorks** y **ObjectStudio** (comerciales, las de los grandes sistemas empresariales), **GemStone/S** (base de datos de objetos), **VA Smalltalk** de Instantiations, y **Glamorous Toolkit**, un entorno de "programación moldeable" construido sobre Pharo.

🧪 El programa de la clase 041 en Smalltalk

> ⚠️ ***Material de lectura, no verificado.*** *Ejecutar Pharo en modo headless dentro del CI es posible pero pesado; no está en los runners de este repositorio.*

```
| linea partes precio cantidad descuento total |

linea := stdin nextLine.
partes := linea substrings collect: [ :cada | cada asNumber ].

precio    := partes first.
cantidad  := partes second.
descuento := partes third.

total := precio * cantidad * (1 - descuento).

Transcript
    show: 'Total: ', (total printShowingDecimalPlaces: 2);
    cr.
```

Recorrido, línea a línea.

- `| linea partes ... |` declara las variables temporales. No llevan tipo: cualquier variable puede referirse a cualquier objeto.
- `:=` es la asignación. El `=` a secas es **igualdad**, y es un mensaje como cualquier otro.
- `linea substrings` es un **mensaje unario**: se envía `substrings` al objeto cadena y devuelve una colección de trozos. Los mensajes unarios se escriben pegados detrás del receptor, sin paréntesis ni punto.
- `collect: [ :cada | cada asNumber ]` es un **mensaje de palabra clave**, y el corchete es un **bloque**: un trozo de código que es a su vez un objeto, con su parámetro declarado como `:cada`. `collect:` es el `map` de Smalltalk. El bloque es la construcción que, cuarenta años después, reapareció como *lambda* en Java 8, C# y Python.
- `precio * cantidad` **no usa operadores**: `*` es un **mensaje binario** enviado al objeto `precio` con `cantidad` como argumento. La clase `Number` implementa `*`. Se puede definir `*` en tus propias clases sin ninguna sintaxis especial de "sobrecarga de operadores", porque nunca hubo operadores que sobrecargar.
- **El orden de evaluación es la trampa clásica.** Smalltalk no tiene precedencia aritmética: los mensajes unarios van primero, luego **todos** los binarios de izquierda a derecha, y por último los de palabra clave. Es decir, `2 + 3 * 4` da **20**, no 14. Por eso `(1 - descuento)` lleva paréntesis: no por costumbre, sino porque sin ellos el resultado sería otro.
- `printShowingDecimalPlaces: 2` es un mensaje al número, no una función de formateo externa. La responsabilidad de saber presentarse es del propio objeto.
- El `;` es la **cascada**: envía el siguiente mensaje al *mismo* receptor (`Transcript`). Es la forma idiomática de encadenar varias operaciones sobre un objeto sin repetir su nombre.
- El `.` separa sentencias, como el punto y coma de otros lenguajes.

Y ahora lo que de verdad hay que ver. En Smalltalk esto también es cierto:

```
(total > 0)
    ifTrue:  [ Transcript show: 'hay venta' ]
    ifFalse: [ Transcript show: 'venta vacía' ]
```

Eso **no es una estructura de control**. `ifTrue:ifFalse:` es un **método implementado en la clase `Boolean`**: `True` lo implementa evaluando el primer bloque, `False` evaluando el segundo. El condicional es polimorfismo. Lo mismo ocurre con `1 to: 10 do: [...]` (un método de `Number`) y `[...] whileTrue: [...]` (un método de `BlockClosure`). Puedes abrir el navegador de clases y **leer el código fuente del `if`**. Ese es el momento en que se entiende de verdad qué significa "todo es un objeto".

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces...	En Smalltalk es...
`obj.metodo()`	`obj metodo` — sin punto, sin paréntesis
`obj.metodo(a, b)`	`obj conA: a conB: b` — el nombre del mensaje se **intercala**
`lista.map(f)`	`lista collect: [ :x | f value: x ]`
`lista.filter(f)`	`lista select: [ :x | ... ]`
`lista.reduce(f)`	`lista inject: 0 into: [ :acc :x | acc + x ]`
`if / else`	`ifTrue: [...] ifFalse: [...]` — mensajes a un booleano
`while (c) { }`	`[ c ] whileTrue: [ ... ]`
Lambda / closure	Bloque `[ :x | ... ]`
`this` / `self`	`self`; y `super` para el método de la superclase
`null`	`nil` — que también es un objeto, instancia de `UndefinedObject`

⚠️ Errores comunes al leerlo

- **Aplicar precedencia aritmética.** `2 + 3 * 4` es `20`. Es el error número uno y no da ningún aviso.
- **Buscar el fichero fuente.** El código vive en la **imagen**, organizado por clases y categorías, y se explora con el navegador. Los ficheros `.st` son un formato de intercambio, no la fuente de verdad. Herramientas como **Iceberg** y el formato **Tonel** son las que reconcilian eso con Git.
- **Confundir `=` con `==`.** `=` es igualdad de valor (redefinible); `==` es identidad del objeto.
- **Olvidar el `^` del retorno.** Un método sin `^` devuelve `self`, no el valor de la última expresión. Es la diferencia con casi todos los lenguajes funcionales y con Ruby.
- **Creer que la clase es lo importante.** Lo importante es el mensaje. Si un objeto responde al mensaje, sirve: *duck typing* en su forma original.
- **Tratar el depurador como último recurso.** En Smalltalk, programar *dentro* del depurador —dejar que falle, implementar el método que faltaba y continuar— es el flujo normal, no una emergencia.

📚 Fuentes y bibliografía

- Pharo (https://pharo.org/) — la implementación libre más activa, con MOOCs gratuitos.
- Squeak (https://squeak.org/) — descendiente directa del Smalltalk-80 de PARC.
- Cincom Smalltalk (https://www.cincomsmalltalk.com/) — VisualWorks y ObjectStudio, en desarrollo comercial continuo.
- **Adele Goldberg, David Robson**, *Smalltalk-80: The Language and its Implementation* — el "Libro Azul", el documento fundacional; disponible libremente en línea.
- **Stéphane Ducasse et al.**, *Pharo by Example* — libro gratuito y actual, la mejor entrada práctica (books.pharo.org (https://books.pharo.org/)).
- **Sherman Alpert, Kyle Brown, Bobby Woolf**, *The Design Patterns Smalltalk Companion* — los patrones de diseño en el lenguaje del que salieron.
- **Alan Kay**, *The Early History of Smalltalk*, ACM HOPL-II, 1993 — el relato de primera mano de por qué el lenguaje es como es.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Common Lisp · M / MUMPS · Delphi

🗄️ SQL — 1974

SQL es el lenguaje más usado del mundo por gente que no se considera programadora, el único de esta lista que **no dice cómo hacer las cosas**, y el que más tiempo lleva sin ser sustituido pese a que cada década alguien lo intenta. Cincuenta años después, **sigue siendo la forma en que la humanidad le pregunta cosas a sus datos**.

> 🎯 ***Por qué está en este programa***
>
> ***SQL es uno de los diez lenguajes del núcleo** y el **representante de la familia declarativa** (Atlas), junto a Prolog y Datalog.*
>
> *Está en el núcleo porque enseña el paradigma que ningún otro del curso enseña: **describir el resultado y dejar que otro decida cómo obtenerlo** (clase 118). Y porque **es la frontera con los datos de casi cualquier sistema real** (clase 170): quien programa cualquier cosa acaba escribiendo SQL.*

Año	1974 (SEQUEL); **SQL-86** el primer estándar; **SQL:2023** el vigente
Autoría	**Donald Chamberlin** y **Raymond Boyce**, IBM San José, sobre la teoría de **Edgar Codd**
Familia	Declarativa / lógica — basada en el **álgebra relacional**
Paradigma	**Declarativo**: se describe el qué, el optimizador decide el cómo
Tipado	Estático, con tipos del esquema; `NULL` **es un tercer valor lógico**
Memoria	No aplica: la gestiona el motor
Ejecución	Se **compila a un plan de acceso** que un optimizador elige por coste
Estado	🟢 **Universal e insustituible**; todo lo que quiso reemplazarlo acabó añadiéndolo

---

📜 Historia

En **1970**, **Edgar F. Codd** publicó en IBM *A Relational Model of Data for Large Shared Data Banks*. La idea era matemática: **los datos son relaciones —conjuntos de tuplas— y se manipulan con un álgebra**. Nada de punteros, nada de recorridos: **conjuntos**.

Era una idea incómoda. La industria estaba en lo jerárquico (IMS) y en lo navegacional (CODASYL), donde el programa recorría los datos a mano (clase 170), y el modelo relacional parecía lento y académico.

En **1974**, **Chamberlin y Boyce** diseñaron **SEQUEL** —*Structured English Query Language*— como una forma legible de expresar el álgebra de Codd. Hubo que renombrarlo a **SQL** por un conflicto de marca con una empresa aeronáutica británica, y por eso muchas personas siguen pronunciándolo "sícuel".

System R, el prototipo de IBM, demostró lo que faltaba: **que un optimizador automático podía elegir un plan tan bueno como el que escribiría una persona**. Sin eso, el modelo relacional no habría ganado.

Larry Ellison leyó los artículos de IBM y sacó **Oracle** al mercado en **1979**, antes que la propia IBM. El estándar llegó en **1986**, y desde entonces cada revisión ha ido añadiendo: integridad referencial y transacciones (**SQL-92**), disparadores y objetos (**SQL:1999**), XML (**2003**), funciones de ventana (**2003**), `MERGE` (**2008**), tablas temporales (**2011**), **JSON** (**2016**) y **grafos con** `MATCH` (**2023**).

Y el intento de sustitución más serio —el movimiento **NoSQL** de 2009— acabó en un lugar instructivo: **casi todas esas bases de datos añadieron después un lenguaje de consulta parecido a SQL**, porque describir lo que se quiere resultó ser mejor que programar cómo obtenerlo.

🏭 Dónde vive hoy

- **Bases de datos relacionales**: PostgreSQL, MySQL/MariaDB, SQL Server, Oracle, **SQLite** —que está en todos los teléfonos y navegadores del mundo (clase 170)—, Db2.
- **Almacenes analíticos**: Snowflake, BigQuery, Redshift, ClickHouse, DuckDB.
- **Motores de datos masivos**: Spark SQL, Trino, Presto, Hive, Flink SQL.
- **Bases NoSQL que volvieron**: Cassandra (CQL), MongoDB (con su lenguaje de agregación y ahora SQL), Elasticsearch (ES|QL).
- **Y en el flujo de datos moderno**: dbt convirtió SQL en el lenguaje de la transformación analítica, con pruebas y control de versiones (clases 139 y 145).

🧠 Lo que enseña: declarar en vez de ordenar

La comparación que resume el paradigma:

```
SELECT cliente, SUM(importe) AS total
FROM pedidos
WHERE fecha >= '2026-01-01'
GROUP BY cliente
HAVING SUM(importe) > 10000
ORDER BY total DESC;
```

En ninguna parte se dice cómo. No se dice si usar un índice, si ordenar antes o después de agrupar, si hacerlo en paralelo o si leer la tabla entera. **Eso lo decide el optimizador**, con estadísticas de los datos, y **puede elegir un plan distinto mañana** si los datos cambian.

Y esa es la propiedad que la clase 118 quiere enseñar: **el programa dice el qué y el sistema elige el cómo**, lo que permite que el mismo programa mejore sin tocarlo.

Y las tres cosas de SQL que más se malinterpretan merecen estar aquí:

Uno, `NULL` no es un valor: es "no se sabe" (clase 100).

```
NULL = NULL        -- ni verdadero ni falso: NULL
WHERE x = NULL      -- nunca devuelve nada; hay que usar IS NULL
COUNT(*) vs COUNT(columna)   -- el segundo NO cuenta los NULL
```

Es lógica de tres valores, y es la causa número uno de resultados sorprendentes.

Dos, el orden lógico de evaluación no es el orden de escritura:

```
Se escribe:  SELECT … FROM … WHERE … GROUP BY … HAVING … ORDER BY
Se evalúa:   FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY
```

Por eso no se puede usar en `WHERE` un alias definido en `SELECT`, y sí en `ORDER BY`.

Y tres, las transacciones y el aislamiento (clases 161 y 172): `READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE` no son detalles de configuración — **cambian qué anomalías puede ver el programa**, y el nivel por defecto varía entre motores.

🔄 Lo que se ha modernizado

- **Funciones de ventana** (`OVER (PARTITION BY … ORDER BY …)`): agregados sin perder las filas. Probablemente la característica más infrautilizada del lenguaje.
- **CTE y CTE recursivas** (`WITH RECURSIVE`): consultas legibles y recorrido de jerarquías — que es Datalog dentro de SQL.
- **JSON como tipo de primera clase** (SQL:2016) con índices: la respuesta a NoSQL, dentro del modelo relacional (clase 159).
- **`MERGE`**, tablas temporales por sistema, `GENERATED` y restricciones más expresivas.
- **`GQL` y `MATCH` (SQL:2023)**: consultas de grafos en el estándar (clase 099).
- **Y el ecosistema alrededor**: **dbt** trajo control de versiones, pruebas y documentación al SQL analítico (clases 139, 145 y 154) — es decir, **convirtió el SQL en software**.

⚙️ Cómo se ejecuta hoy

```
sqlite3 :memory: < main.sql          # el comando de la clase 041
psql -f consulta.sql                  # PostgreSQL
duckdb -c "SELECT * FROM 'datos.parquet'"   # analítica local, sin servidor

EXPLAIN ANALYZE SELECT …              # ← el comando más importante de esta ficha (clase 152)
```

> `EXPLAIN ANALYZE` ***merece el subrayado**: enseña **el plan que el optimizador eligió y lo que costó de verdad**. Es la única forma de saber por qué una consulta va lenta, y es la herramienta de la clase 152 aplicada al componente de datos.*

💉 El programa de la clase 041 en SQL

```
-- SQL es declarativo: no lee stdin como los lenguajes imperativos. En vez de
-- una variable que se asigna, se describe el cálculo sobre una tabla de valores.
-- Esta consulta demuestra la misma fórmula para los tres casos de casos.json.
WITH ventas(precio_unitario, cantidad, descuento) AS (
    VALUES (15000.0, 2, 0.10),
           (999.9, 3, 0.0),
           (5000.0, 0, 0.20)
)
SELECT printf('Total: %.2f', precio_unitario * cantidad * (1 - descuento)) AS resultado
FROM ventas;
```

Lo que hay que ver, y es lo más interesante de esta ficha.

- **Este programa no puede tener la forma de los demás**, y esa imposibilidad es el contenido. **SQL no lee de la entrada estándar ni tiene variables que se asignan**: se declara un conjunto de filas y se describe qué se quiere de él. La clase 040 llama a esto **contrato adaptado**, y declararlo es más honesto que fingir.
- `WITH … AS (VALUES …)` construye una tabla al vuelo. En el resto de las fichas, los tres valores son tres variables; aquí son **tres filas**, y la consulta se aplica a las tres a la vez.
- **No hay bucle.** Esa ausencia es el paradigma: **la operación se define sobre el conjunto**, y el motor decide si recorre, si paraleliza o si usa un índice.
- `printf` **es de SQLite**; en PostgreSQL sería `to_char` y en SQL Server `FORMAT`. **El formateo es la parte menos estándar de SQL**, y es un recordatorio útil de que "SQL" son varios dialectos con un núcleo común (clase 160).

📚 Fuentes y bibliografía

- Documentación de PostgreSQL (https://www.postgresql.org/docs/) — la mejor documentación de base de datos que existe, y sirve para aprender SQL aunque uses otro motor.
- Use The Index, Luke! (https://use-the-index-luke.com/es) — en español; explica índices y planes de ejecución mejor que ningún libro (clase 152).
- Modern SQL (https://modern-sql.com/) — qué hay en el estándar desde SQL-92 y qué motor lo implementa. Funciones de ventana y CTE, explicadas de verdad.
- **Markus Winand**, *SQL Performance Explained* — corto, denso y práctico.
- **C. J. Date**, *SQL and Relational Theory*, 3.ª ed., O'Reilly — la teoría, incluido el problema de `NULL`, por uno de los que estuvieron desde el principio.
- **Edgar F. Codd**, *A Relational Model of Data for Large Shared Data Banks* (1970) — el artículo original; se lee en veinte minutos y explica de dónde viene todo.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Prolog · Datalog · M/MUMPS · COBOL · PL/I

🕊️ Swift — 2014

Swift es el sucesor de Objective-C, y su diseño se puede resumir en una frase: **tomar todo lo que la investigación en lenguajes había demostrado y meterlo en un lenguaje que millones de personas iban a usar el mes siguiente**. Tipos algebraicos, opcionales, inmutabilidad por defecto y concurrencia estructurada — con la interfaz gráfica de Apple detrás.

> 🎯 ***Por qué está en este programa***
>
> *Swift es un **primo de la familia móvil / moderno** (Atlas), junto a Dart, y desciende directamente de Objective-C.*
>
> *Aporta al programa **la gestión de memoria por conteo automático de referencias sin recolector** (clase 131), con el problema de los ciclos a la vista; y **la concurrencia estructurada con actores integrados en el sistema de tipos** (clases 133 y 136), que es de las implementaciones más completas que existen.*

Año	2014; **ABI estable** en 2019; **Swift 6** en 2024, con concurrencia estricta
Autoría	**Chris Lattner** —creador de LLVM— y equipo, Apple
Familia	Móvil / moderno; con Objective-C, Rust, Haskell y C# dentro
Paradigma	Multiparadigma: OO por protocolos, funcional e imperativo
Tipado	**Estático, fuerte, con inferencia** y opcionales en el tipo
Memoria	**ARC**: conteo automático de referencias, sin recolector y sin pausas
Ejecución	Compilado a nativo sobre **LLVM**
Estado	🟢 **Dominante** en el ecosistema Apple; creciendo en servidor y embebido

---

📜 Historia

Chris Lattner había creado **LLVM** como proyecto de máster y lo había llevado a Apple, donde se convirtió en la base de todas sus herramientas. En **2010** empezó Swift **en secreto**, y Apple lo presentó en la WWDC de **2014** sin previo aviso.

El encargo era difícil: **sustituir a Objective-C** —treinta años de marcos y de código— **sin romper nada**. La solución fue la interoperabilidad total: **Swift y Objective-C conviven en el mismo proyecto**, y Swift importa las APIs de Objective-C con tipos precisos gracias a las anotaciones de nulabilidad que Apple añadió a Objective-C **precisamente para eso**.

Los hitos:

- **Swift 3 (2016)**: rediseño de la biblioteca y de los nombres; **la última ruptura grande**.
- **Swift 5 (2019)**: **ABI estable** —el lenguaje deja de tener que incluir su biblioteca en cada aplicación (clase 157)—.
- **Swift 5.5 (2021)**: `async` / `await`, **actores** y concurrencia estructurada.
- **Swift 6 (2024)**: **comprobación estricta de concurrencia** — el compilador detecta las carreras de datos, con un modelo emparentado con el de Rust (clase 136).

Y desde **2015 es de código abierto**, con versiones para Linux y Windows y un empuje serio hacia el servidor y los sistemas embebidos.

🏭 Dónde vive hoy

- **Todo el ecosistema Apple**: iOS, macOS, watchOS, visionOS — con **SwiftUI** como marco declarativo de interfaz (clase 169).
- **Servidores**: **Vapor** y **Hummingbird**; Apple lo usa en su propia infraestructura.
- **Sistemas embebidos**: *Embedded Swift* (2024), un subconjunto sin tiempo de ejecución para microcontroladores — el mismo camino que Ada con su perfil reducido (clase 162).
- **Y en interoperabilidad con C++**, un desarrollo reciente que le abre bibliotecas existentes (clase 156).

🧠 Lo que enseña: ARC y concurrencia en el tipo

Uno, el conteo automático de referencias (clase 131):

```
class Nodo {
    var siguiente: Nodo?          // referencia FUERTE: mantiene vivo
    weak var padre: Nodo?          // DÉBIL: no cuenta, y se pone a nil al morir
    unowned let dueño: Documento    // sin dueño: no cuenta y NO se pone a nil
}
```

El compilador inserta `retain` **y** `release`, así que no hay recolector y **no hay pausas** — lo que importa mucho en una animación a 120 fotogramas por segundo (clase 152).

Y el precio es el que la clase 131 explica: los ciclos no se recogen. Dos objetos que se apuntan mutuamente con referencias fuertes **no se liberan nunca**, y **el programador tiene que romper el ciclo** con `weak` o `unowned`.

Es el mismo modelo que Objective-C con ARC, que Rust con `Rc` / `Weak` **y que el conteo de referencias de Python y PHP** — con la diferencia de que estos dos últimos **sí** tienen un detector de ciclos adicional.

Dos, la concurrencia en el sistema de tipos (clase 136):

```
actor Banco {
    private var saldo = 0.0           // ← nadie puede tocarlo desde fuera sin await
    func depositar(_ x: Double) { saldo += x }
}

let b = Banco()
await b.depositar(100)                 // ← el await marca el cruce de frontera
```

Un actor aísla su estado: solo se accede desde dentro, y desde fuera **hay que esperar**. El compilador lo garantiza (clase 133).

Y `Sendable` es el otro lado: un tipo marcado como `Sendable` **puede cruzar fronteras de concurrencia con seguridad**, y **el compilador comprueba que no se envía nada que no lo sea**. Es exactamente el papel de `Send` / `Sync` en Rust, integrado en un lenguaje con ARC.

Y tres, los opcionales, que ya no sorprenden a nadie precisamente porque Swift ayudó a popularizarlos:

```
var nombre: String = "Ana"      // no puede ser nil
var apodo: String? = nil         // puede
if let a = apodo { ... }          // desempaquetado seguro
guard let a = apodo else { return }
```

🔄 Lo que se ha modernizado

- **Swift 6 con concurrencia estricta**: las carreras de datos pasan a ser errores de compilación.
- **Macros** (5.9): metaprogramación con transformación del árbol sintáctico, comprobada y depurable (clase 123) — a diferencia de las macros de texto de C.
- **SwiftUI y Observation**: interfaz declarativa con estado observable (clase 169).
- **Embedded Swift**: sin recolector, sin reflexión y sin metadatos, para microcontroladores.
- **Interoperabilidad con C++** bidireccional, y **Swift en Windows y Linux** con soporte oficial.
- **Y `swift-format` y SwiftLint** para el estilo (clase 146).

⚙️ Cómo se ejecuta hoy

```
swift main.swift < entrada.txt        # ejecutar como guion
swiftc -O main.swift -o venta          # compilar optimizado

swift build && swift test               # con Swift Package Manager (clases 143 y 139)
swift package init --type executable
```

🧪 El programa de la clase 041 en Swift

Esta versión se escribe aquí y **no está verificada en CI** (clase 040).

```
import Foundation

guard let linea = readLine() else { exit(1) }
let v = linea.split(separator: " ").compactMap { Double($0) }
guard v.count == 3 else { exit(1) }

let total = v[0] * v[1] * (1 - v[2])
print(String(format: "Total: %.2f", total))
```

Lo que hay que ver.

- `guard let ... else` es la forma idiomática de Swift para el camino de error: **desempaqueta el opcional y sale si no hay valor**, dejando el resto de la función sin anidamiento. Compárese con el `!` de Kotlin y de Dart, que afirma en lugar de comprobar.
- `readLine()` **devuelve** `String?` porque puede no haber línea — **la posibilidad de fallo está en el tipo**, y el compilador obliga a tratarla (clase 116).
- `compactMap` **convierte y descarta los nulos a la vez**: `Double($0)` devuelve `Double?`, y `compactMap` se queda con los que valen. Es un idioma muy del lenguaje.
- `let` **es inmutable**, `var` es mutable, y la convención es `let` por defecto (clase 102) — como Rust, Kotlin y Scala.
- **Y** `String(format:)` **viene de Foundation**, que es **la biblioteca de Objective-C**: la herencia se ve en la primera línea del programa.

📚 Fuentes y bibliografía

- The Swift Programming Language (https://docs.swift.org/swift-book/) — el libro oficial, libre y muy bien escrito.
- Swift Evolution (https://github.com/swiftlang/swift-evolution) — **todas las propuestas con su discusión pública**; de lo mejor que hay para entender por qué un lenguaje toma sus decisiones (clase 175).

- Swift.org (https://www.swift.org/documentation/) — para el uso en servidor y multiplataforma.
- **Paul Hudson**, *Hacking with Swift* — libre en línea; el recurso práctico más usado.
- **Chris Lattner**, entrevistas y charlas sobre el diseño de Swift y de LLVM (clase 123).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: Objective-C · Rust · Kotlin · Dart · C++

🔧 Tcl/Tk — 1988

El lenguaje con el que se diseñan los chips. Cada procesador moderno pasa por un flujo de herramientas de diseño electrónico, y ese flujo se conduce con guiones Tcl. Synopsys, Cadence y Xilinx exponen sus herramientas como comandos Tcl. Es un lenguaje casi invisible que está debajo del silicio que estás usando ahora mismo.

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: Tcl se ejecuta hoy en la cadena de herramientas EDA de toda la industria del semiconductor**, en equipamiento de red (Cisco IOS lleva Tcl incrustado), en automatización de pruebas con **Expect**, y detrás de `tkinter`, la biblioteca gráfica que viene con Python. El lenguaje publicó **Tcl 9.0** en 2024: sigue en desarrollo.*
>
> *Entra porque **lleva al extremo un concepto que el núcleo esconde**: en Tcl **no hay sintaxis, solo comandos**. `if`, `while`, `for` y `proc` no son palabras clave del lenguaje: son procedimientos normales que reciben cadenas y las evalúan. Eso significa que **puedes escribir tus propias estructuras de control** sin macros ni compilador, algo que en el núcleo entero solo se aproxima con las lambdas. Y su regla "todo es una cadena" (EIAS) es la versión más pura del tipado débil que existe: aclara de golpe qué significan realmente la coerción de JavaScript o el tipado de PHP.*

Año	1988 (Tcl); **Tk** en 1991; versión vigente **Tcl 9.0** (2024)
Autoría	**John Ousterhout**, Universidad de California en Berkeley
Familia	Scripting dinámico / lenguajes de comandos incrustables
Paradigma	Imperativo y procedimental; OO desde 8.6 con **TclOO**
Tipado	**Dinámico y débil** — *Everything Is A String*
Memoria	Gestionada por conteo de referencias, con objetos con doble representación
Ejecución	Interpretado a bytecode; diseñado para **incrustarse** dentro de una aplicación C
Estado	🟡 **Nicho industrial muy sólido** — EDA, redes, testing, GUI multiplataforma

---

📜 Historia

A finales de los 80, **John Ousterhout** dirigía en Berkeley un grupo que construía herramientas de diseño de circuitos integrados. Observó un patrón repetido: cada herramienta acababa necesitando un lenguaje de comandos propio, cada uno se inventaba desde cero, y todos eran malos. Su propuesta fue escribir **un solo lenguaje de comandos, pequeño y empotrable**, que cualquier aplicación en C pudiera enlazar y extender con sus propios comandos.

Eso es **Tcl** (*Tool Command Language*, pronunciado "tickle"). Su diseño está subordinado por completo a ese objetivo: sintaxis mínima (doce reglas caben en una página de manual), una única representación de datos —la cadena—, y una API en C para registrar comandos nuevos que es tan simple que se aprende en una tarde.

En 1991 llegó **Tk**, el kit de widgets gráficos. Fue el primer sistema que permitió construir una interfaz gráfica multiplataforma con veinte líneas de guion, y su impacto fue enorme: **Python lo adoptó como** `tkinter` (que sigue siendo la GUI de la biblioteca estándar) y Perl y Ruby hicieron lo propio. Buena parte de la gente que ha usado Tk nunca ha escrito una línea de Tcl.

Un año antes, en 1990, **Don Libes** escribió **Expect** sobre Tcl para automatizar programas interactivos: guiones que "esperan" un texto en la salida de otro programa y responden. Sigue siendo la herramienta canónica para automatizar sesiones de consola, y es la razón de que Tcl viva en la administración de equipos de red.

La industria EDA lo adoptó y ya no lo soltó: hoy los flujos de síntesis y de disposición física (*place & route*) se describen en Tcl porque las herramientas de Synopsys, Cadence y AMD/Xilinx exponen su funcionalidad como comandos Tcl.

🏭 Dónde sobrevive hoy

- **Diseño electrónico (EDA)**: Synopsys Design Compiler, Cadence Innovus y Genus, AMD Vivado. Los guiones de síntesis, restricciones temporales (**SDC**, que es un formato Tcl) y flujos de fabricación se escriben en Tcl.
- **Equipos de red**: Cisco IOS incorpora un intérprete Tcl para automatización en el propio equipo.
- **Automatización de pruebas y de sesiones interactivas**: **Expect**.
- **Interfaces gráficas**: **Tk** sigue siendo la GUI multiplataforma con menos fricción, y llega a millones de usuarios a través de `tkinter` de Python.
- **Software científico e industrial**: incrustado como capa de guion en aplicaciones grandes de C++.

🧠 Por qué no ha muerto

1. Incrustarlo es trivial. Enlazar el intérprete y registrar comandos propios desde C son unas pocas llamadas. Para un fabricante de herramientas, eso significa exponer todo su producto como lenguaje de guion con un esfuerzo mínimo. Ninguna alternativa —ni Python ni Lua— es más simple de integrar a ese nivel.

2. Los flujos de fabricación no se tocan. Un guion de síntesis validado sobre un proceso de fabricación concreto es un activo que se hereda de proyecto en proyecto. Cambiar el lenguaje del flujo no aporta nada y arriesga mucho.

3. La sintaxis mínima resiste el paso del tiempo. Un guion Tcl de 1995 se ejecuta hoy. La compatibilidad hacia atrás ha sido casi total durante treinta años.

4. Estabilidad como característica, no como estancamiento. Tcl 8.6 trajo corrutinas y OO; Tcl 9.0 trajo soporte completo de Unicode y ficheros grandes. Evoluciona despacio y a propósito.

🔄 Lo que se ha modernizado

- **Tcl 9.0 (2024)**, la primera versión mayor en veinte años: **Unicode completo** más allá del plano básico, ficheros y cadenas **de más de 2 GB**, enteros de 64 bits, y sistema de ficheros virtual con soporte de ZIP integrado (una aplicación puede llevar sus recursos dentro del propio ejecutable).
- **TclOO** (desde 8.6): un sistema de objetos **en el núcleo del lenguaje**, con metaclases y mixins, que unificó los tres o cuatro sistemas OO que competían en la comunidad.

- **Corrutinas** (8.6): concurrencia cooperativa sin hilos, y un manejo de excepciones estructurado con `try` / `trap` / `finally`.
- **Tk moderno**: los *widgets* temáticos `ttk` adoptan el aspecto nativo de Windows, macOS y Linux; la interfaz de 2026 no se parece a la de los 90. Y sigue llegando a millones de usuarios a través de `tkinter`.
- **Starkit/Starpack**: empaquetar aplicación e intérprete en **un solo fichero ejecutable** — la misma idea del binario autocontenido de Go, dos décadas antes.
- **Vigencia en EDA**: las herramientas de Synopsys, Cadence y AMD siguen exponiendo su funcionalidad como comandos Tcl en sus versiones actuales. Aquí no hay migración a la vista.

⚙️ Cómo se ejecuta hoy

```
sudo apt-get install -y tcl

tclsh total.tcl < entrada.txt
echo "15000 2 0.10" | tclsh total.tcl
# Total: 27000.00

# Con interfaz gráfica:
wish miapp.tcl
```

Ecosistema: `tclsh` (intérprete), `wish` (intérprete con Tk cargado), **Tcllib** y **Tklib** (bibliotecas estándar de la comunidad), y **Starkit/Starpack**, un mecanismo para empaquetar una aplicación entera —intérprete, código y recursos— en un **único fichero ejecutable**, mucho antes de que eso fuera habitual.

🧪 El programa de la clase 041 en Tcl

```
gets stdin linea
lassign [split [string trim $linea]] precio cantidad descuento
set total [expr {$precio * $cantidad * (1 - $descuento)}]
puts [format "Total: %.2f" $total]
```

Recorrido, línea a línea.

- `gets stdin linea` — `gets` **no** es una palabra clave: es un comando, y `stdin` y `linea` son sus dos argumentos. Con dos argumentos, guarda la línea leída en la variable indicada y devuelve su longitud. Toda la sintaxis de Tcl es esta: `comando arg1 arg2 ...`, separados por espacios.
- Los **corchetes** `[...]` son **sustitución de comando**: se ejecuta lo de dentro y su resultado se inserta en su lugar. Equivalen a la `$(...)` del shell. `[split [string trim $linea]]` primero recorta espacios y luego parte por espacios en blanco.
- `lassign lista a b c` reparte los elementos de una lista en varias variables. Y aquí aparece la idea nuclear: **la lista es una cadena** con los elementos separados por espacios. `split` no construye una estructura nueva; produce una cadena que Tcl sabe interpretar como lista.
- `set total [...]` asigna. No hay operador `=`: `set` es un comando.
- **`expr` con llaves es la línea importante.** Tcl no tiene operadores aritméticos; `expr` es un comando que recibe una expresión y la evalúa. Las **llaves** impiden que Tcl sustituya las variables *antes* de pasar el texto a `expr`, de modo que `expr` recibe la expresión con los nombres y la compila una sola vez. Sin llaves — `expr $a * $b` — funciona, pero es más lento y, si una variable contiene texto con espacios o corchetes, **se reinterpreta**, lo que es un agujero de inyección clásico. La regla de la comunidad es absoluta: **`expr` siempre con llaves**.
- `format "%.2f" $total` es el `printf` de Tcl; `puts` escribe la línea.

Y ahora el motivo real de estudiarlo. En Tcl, `if` es un comando corriente. Puedes escribir el tuyo:

```
proc repetir {n cuerpo} {
    for {set i 1} {$i <= $n} {incr i} {
        uplevel 1 $cuerpo
    }
}

set contador 0
repetir 3 { incr contador ; puts "vuelta $contador" }
```

`repetir` recibe el bloque `{ ... }` como una **cadena** y `uplevel 1` la evalúa **en el ámbito de quien llamó**, de modo que `contador` es la variable del llamante y no una copia. Acabas de añadir una estructura de control al lenguaje con un procedimiento de cuatro líneas, sin macros y sin tocar el compilador. Su pareja, `upvar`, hace lo mismo con variables y es la forma de escribir un procedimiento que modifica los argumentos de su llamante.

Esa capacidad —que las estructuras de control sean código de usuario— la comparten muy pocos lenguajes: Lisp con macros, Smalltalk con bloques, y Tcl con la evaluación explícita de cadenas.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces…	En Tcl es…
`x = 5`	`set x 5`
`$(comando)` del shell	`[comando]` — sustitución de comando
`x = a * b`	`set x [expr {$a * $b}]`
`print(x)`	`puts $x`
`def f(a, b):`	`proc f {a b} { ... }`
`if cond:`	`if {$cond} { ... } else { ... }` — `if` es un comando
`for i in range(n)`	`for {set i 0} {$i < $n} {incr i} { ... }`
Lista / array	`list`, `lindex`, `lappend`, `lsort` — y todo es cadena
Diccionario	`dict get`, `dict set`, o los *arrays asociativos* `a(clave)`
`"%.2f" % x`	`format "%.2f" $x`
`import`	`package require nombre`
Paso por referencia	`upvar`

⚠️ Errores comunes al leerlo

- `expr` **sin llaves.** Además de ser lento, permite inyección. Es el error que más aparece en revisiones de código Tcl.
- **Confundir `{}` con un bloque de código.** Las llaves son **comillas que no sustituyen nada**. Que `if {...} { ... }` parezca C es una coincidencia afortunada: el segundo grupo es simplemente una cadena que `if` decidirá evaluar.
- **Espacios obligatorios.** `set x[expr 1]` no funciona: los argumentos se separan por espacios, siempre. `if{$a}` tampoco, porque el comando se llamaría `if{$a}`.
- **Creer que hay tipos.** `"10"`, `10` y `10.0` son la misma cadena hasta que un comando decide cómo leerla. Internamente Tcl cachea una representación numérica por rendimiento, pero eso es invisible y no cambia la semántica.
- **Usar comillas dobles donde van llaves.** `"..."` **sí** sustituye variables y comandos; `{...}` no. Elegir mal cambia el momento en que se evalúa algo.

- **Ignorar `Tcl_Eval` en código C.** Si lees el fuente de una herramienta EDA, la mitad del lenguaje no está en Tcl: son comandos registrados desde C. La documentación del comando está en el manual de la herramienta, no en el de Tcl.

📚 Fuentes y bibliografía

- tcl-lang.org (https://www.tcl-lang.org/) — sitio oficial, con el manual de referencia de cada comando.
- Tcl/Tk 9.0 — novedades (https://www.tcl-lang.org/software/tcltk/9.0.html) — el estado actual del lenguaje.
- The Tcler's Wiki (https://wiki.tcl-lang.org/) — la memoria colectiva de la comunidad; treinta años de recetas.
- **John Ousterhout, Ken Jones**, *Tcl and the Tk Toolkit*, 2.ª ed., Addison-Wesley — escrito por el autor del lenguaje; explica el porqué de cada decisión de diseño.
- **Brent Welch, Ken Jones**, *Practical Programming in Tcl and Tk*, 4.ª ed., Prentice Hall — el manual práctico de referencia.
- **Don Libes**, *Exploring Expect*, O'Reilly — automatización de programas interactivos, por el autor de Expect.
- **John Ousterhout**, *Scripting: Higher Level Programming for the 21st Century*, IEEE Computer, 1998 — el artículo que argumentó por qué los lenguajes de guion complementan a los de sistemas en lugar de competir con ellos.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: Perl · Common Lisp · AutoLISP

🔷 TypeScript — 2012

TypeScript es un experimento que salió bien: **añadir tipos estáticos a un lenguaje dinámico que ya tenía millones de líneas escritas**, sin romper ninguna. Su diseño está lleno de decisiones pragmáticas que un lenguaje nuevo no habría tomado, y esas decisiones son exactamente lo que lo hizo funcionar.

> 🎯 ***Por qué está en este programa***
>
> ***TypeScript es uno de los diez lenguajes del núcleo**, y está junto a JavaScript por una razón pedagógica: **es el mismo lenguaje con una capa de comprobación encima**, así que **la comparación entre los dos aísla exactamente lo que aportan los tipos**.*
>
> *Aporta al programa el concepto de **tipado gradual** —tipos donde compensan, dinámico donde no— y el de **borrado de tipos**: en TypeScript los tipos **desaparecen al compilar** y no existen en ejecución (clase 108), que es una decisión con consecuencias muy visibles.*

Año	2012; **2.0** con `strictNullChecks` (2016); versión cada 3 meses
Autoría	**Anders Hejlsberg** y equipo, Microsoft — el mismo autor de Delphi y C#
Familia	JavaScript / web — superconjunto sintáctico estricto de JS
Paradigma	El de JavaScript, con un sistema de tipos estructural muy expresivo
Tipado	**Estático, estructural y gradual**; **borrado** en tiempo de compilación
Memoria	La de JavaScript: recolector automático
Ejecución	Se **transpila** a JavaScript; también se ejecuta directo en Deno, Bun y Node 22+
Estado	🟢 **Estándar de facto** para JavaScript a escala

📜 Historia

Hacia 2010, Microsoft construía aplicaciones web grandes —Office en el navegador— y se encontraba con el problema que todo equipo grande encuentra en JavaScript: **sin tipos, refactorizar es adivinar** (clase 150).

Anders Hejlsberg, que ya había diseñado Turbo Pascal, Delphi y C#, dirigió el proyecto. Y la decisión de diseño clave, la que lo hizo posible, fue esta:

> ***TypeScript es un superconjunto de JavaScript.*** *Todo programa JavaScript válido es un programa TypeScript válido. Los tipos son* ***opcionales*** *y* ***se borran al compilar****.*

Eso permitió lo que ningún lenguaje nuevo consigue: **adoptarlo fichero a fichero**, en una base de código existente, sin reescribir nada.

TypeScript 2.0 (2016) añadió `strictNullChecks`, que separa `string` de `string | null` y elimina por comprobación la familia de errores más común de JavaScript. **2.8** trajo los tipos condicionales, y a partir de ahí el sistema de tipos creció hasta ser **Turing-completo**: se pueden escribir cosas como un analizador de rutas en el propio sistema de tipos.

El lenguaje se hizo estándar de facto con **Angular** (2016) y después con casi todo el ecosistema. Y en **2025** el compilador se está reescribiendo en Go —el proyecto *Corsa*— buscando un orden de magnitud de mejora en velocidad, que es la queja histórica.

🏭 Dónde vive hoy

- **Front-end**: React, Angular, Vue, Svelte — el ecosistema entero está tipado.
- **Node.js del lado del servidor**: NestJS, tRPC, y la mayoría de las bibliotecas publican tipos.
- **Herramientas de desarrollo**: VS Code está escrito en TypeScript.
- **Definiciones de tipos para bibliotecas JS**: el repositorio **DefinitelyTyped** tiene tipos para decenas de miles de paquetes que no los traen.
- **Contratos entre servicios**: generar tipos de TypeScript desde OpenAPI o Protobuf es una práctica estándar (clase 160).

🧠 Las tres decisiones que lo explican

Una, el tipado estructural. TypeScript no mira el nombre del tipo, mira **su forma**:

```
interface Punto { x: number; y: number }
function dibujar(p: Punto) { /* ... */ }

dibujar({ x: 1, y: 2, color: "rojo" });   // ✓ compatible: tiene x e y
```

Es lo contrario del **tipado nominal** de Java, C# y Ada, donde hay que **declarar** que se implementa la interfaz. Y encaja con JavaScript, donde los objetos se crean al vuelo (clase 112).

Dos, los tipos se borran. Esto es lo que más sorprende y lo que más consecuencias tiene:

```
interface Usuario { nombre: string }
const u = JSON.parse(texto) as Usuario;    // ← ¡NADIE comprueba esto!
```

En ejecución no hay tipos, así que un `as` es una promesa sin verificación (clase 108). La defensa del ecosistema es validar en la frontera con bibliotecas como **Zod** o **Valibot**, que **generan el tipo a partir del validador** — la aplicación exacta de la regla de la clase 153: **validar donde entra el dato**.

Y tres, la escotilla de escape. `any` desactiva la comprobación, y `strict` la vuelve exigente. El lenguaje deja elegir **cuánta garantía se quiere**, y esa es la definición del tipado gradual.

> ***El coste de esa flexibilidad es real.*** *Un proyecto sin `strict`, con `any` repartidos y con aserciones `as`,* ***tiene la ceremonia de un lenguaje tipado y las garantías de uno dinámico****. Por eso la primera decisión de cualquier proyecto TypeScript debería ser `"strict": true`, y las excepciones, justificadas una a una (clase 146).*

🔄 Lo que se ha modernizado

- `satisfies` (4.9): comprobar que un valor encaja con un tipo **sin perder su tipo concreto**.
- `const` **en parámetros de tipo** y tipos literales de plantilla, que permiten expresar contratos muy precisos.
- **Decoradores** del estándar (5.0), tras años con la versión experimental.
- **Ejecución directa**: Deno, Bun y Node 22+ ejecutan `.ts` **borrando los tipos sin comprobarlos**, lo que separa la comprobación —que se hace en la integración continua (clase 147)— de la ejecución.
- `tsgo`, el compilador reescrito en Go, con mejoras de un orden de magnitud en proyectos grandes.

⚙️ Cómo se ejecuta hoy

```
npx tsc main.ts --target es2022 && node main.js    # el camino clásico
npx tsx main.ts                                     # ejecutar directo
deno run --allow-read main.ts

npx tsc --noEmit                # solo COMPROBAR, sin generar (para CI, clase 147)
```

```
// tsconfig.json — la primera decisión del proyecto
{ "compilerOptions": { "strict": true, "noUncheckedIndexedAccess": true } }
```

🧹 El programa de la clase 041 en TypeScript

```
import { readFileSync } from "node:fs";

// TypeScript añade tipos estáticos sobre JavaScript: se comprueban al compilar.
const [precio, cantidad, descuento]: number[] = readFileSync(0, "utf8")
  .trim()
  .split(/\s+/)
  .map(Number);

const subtotal: number = precio * cantidad;
const total: number = subtotal * (1 - descuento);

console.log(`Total: ${total.toFixed(2)}`);
```

Lo que hay que ver, comparado con la versión JavaScript.

- **Es el mismo programa con anotaciones.** Quitando `: number[]` y `: number`, se obtiene JavaScript válido — que es la tesis del lenguaje.

- `number[]` **es una promesa que el compilador no puede cumplir del todo**: con `noUncheckedIndexedAccess`, `precio` sería `number | undefined`, porque **el compilador no sabe que la línea trae tres campos**. Esa opción, que casi nadie activa, es la que hace honesto el tipo.
- **Sigue habiendo un solo tipo numérico**, con lo que las consecuencias de la clase 045 son las de JavaScript: TypeScript **no añade enteros**, añade comprobación.
- **Y todas las anotaciones desaparecen** al compilar: el `.js` resultante es exactamente el de la ficha de JavaScript.

📚 Fuentes y bibliografía

- Manual oficial de TypeScript (https://www.typescriptlang.org/docs/handbook/intro.html) — y el playground (https://www.typescriptlang.org/play), que enseña el JavaScript generado.
- Notas de versión (https://www.typescriptlang.org/docs/handbook/release-notes/overview.html) — cada versión con ejemplos.
- **Marius Schulz**, *TypeScript Evolution* — la historia de cada característica y por qué se añadió.
- **Boris Cherny**, *Programming TypeScript*, O'Reilly — el sistema de tipos explicado a fondo.
- **Dan Vanderkam**, *Effective TypeScript*, 2.ª ed., O'Reilly — 83 recomendaciones concretas; el mejor libro para pasar de "compila" a "está bien tipado".

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: JavaScript · C# · Dart · Elm

📊 VBA — 1993

Probablemente el entorno de programación más usado del mundo, y casi nadie lo llama programar. Cada departamento financiero, cada área de control de gestión y cada oficina técnica tiene hojas de Excel con macros. Ese código mueve presupuestos, cierres contables y decisiones de inversión, y lo escribió alguien cuyo puesto no dice "desarrollador".

> 🎯 ***Por qué está en este programa***
>
> ***Criterio de inclusión: VBA viene instalado con Microsoft Office y Microsoft mantiene su documentación de referencia del lenguaje y del modelo de objetos.*** *No está obsoleto ni descatalogado: la base instalada es sencillamente demasiado grande. Las alternativas modernas —Office Scripts en TypeScript, Power Automate, Python en Excel— se han* ***añadido****, no lo han sustituido.*
>
> *Entra por dos motivos. El primero es el mismo que AutoLISP: es el ejemplo canónico de* ***lenguaje incrustado en una aplicación anfitriona****, donde la biblioteca estándar es el* ***modelo de objetos*** *del programa que lo hospeda. El segundo es más sutil y más valioso: VBA es donde millones de personas descubren que* ***una hoja de cálculo es un programa****. Toda la Parte 0 —qué es un algoritmo, qué es el estado, por qué importa la reproducibilidad— se juega ahí con consecuencias económicas reales, y ese es un contexto que ningún ejemplo de laboratorio iguala.*

Año	1993, con Excel 5.0; **VBA 7** (con soporte de 64 bits) desde Office 2010
Autoría	**Microsoft**, derivado de **Visual Basic** (1991) y este de QuickBASIC
Familia	BASIC
Paradigma	Imperativo y procedimental, con objetos **COM**
Tipado	Estático opcional (`Option Explicit`) y dinámico por defecto vía `Variant`
Memoria	Conteo de referencias de COM
Ejecución	**Interpretado dentro del anfitrión** (Excel, Word, Access, Outlook…)
Estado	🟢 **Base instalada enorme** — finanzas, administración, ingeniería, informes

---

📜 Historia

Visual Basic 1.0 salió en 1991 y su idea fue revolucionaria en su momento: construir la interfaz arrastrando controles y escribir el código detrás de los eventos. En **1993**, Microsoft incrustó el motor de ese lenguaje dentro de **Excel 5.0** con el nombre de **Visual Basic for Applications**, para sustituir al viejo lenguaje de macros XLM. La jugada era estratégica: si las empresas escribían sus procesos en VBA, migrar de Office a otra suite se volvía muy caro.

Funcionó. VBA se extendió a Word, Access, PowerPoint, Outlook y Visio, y Microsoft lo licenció a terceros: **AutoCAD**, **SolidWorks**, **CorelDRAW**, **ArcGIS** y muchas otras aplicaciones lo incorporaron como su lenguaje de automatización.

Los años 2000 trajeron las **macros maliciosas**: los virus de documento (Melissa, ILOVEYOU y descendientes) explotaron que abrir un fichero podía ejecutar código. La respuesta fue el modelo de seguridad actual —macros deshabilitadas por defecto, extensiones `.xlsm` frente a `.xlsx`, firma digital y **bloqueo automático de macros en ficheros descargados de Internet**—. Es un contexto que hay que conocer antes de escribir la primera línea.

Con .NET, Microsoft intentó reemplazarlo: **VSTO** y **Visual Studio Tools for Office** apuntaban a que las soluciones Office se escribieran en C# o VB.NET. Nunca desplazó a VBA, porque VBA tiene una ventaja imbatible: **está ya instalado, no requiere permisos de administrador y se abre con Alt+F11**. En una empresa con equipos bloqueados, eso lo decide todo.

El estado real, sin adornos: el lenguaje está **congelado** en VBA 7.1. No recibe características nuevas. Microsoft ha declarado que sigue soportado y ha ido añadiendo alternativas —**Office Scripts** (TypeScript, solo en Excel para web y escritorio con Microsoft 365), **Power Automate**, **Python en Excel**— pero ninguna cubre todavía lo que VBA cubre en escritorio y en las aplicaciones de terceros que lo licenciaron.

🏭 Dónde sobrevive hoy

- **Finanzas y control de gestión**: modelos de valoración, consolidación, presupuestos, cierres mensuales, informes regulatorios.
- **Administración y operaciones**: automatización de informes, conciliaciones, cruces de ficheros, generación masiva de documentos en Word desde datos de Excel.
- **Ingeniería y diseño**: macros en AutoCAD, SolidWorks e Inventor para automatizar tareas repetitivas y generar listas de materiales.
- **Access**: aplicaciones departamentales completas —formularios, informes, lógica— que en muchas organizaciones siguen siendo el sistema real de un área.
- **Ciencia de laboratorio y análisis**: instrumentación que exporta a Excel y se procesa con macros.

🧠 Por qué no ha muerto

1. Está donde están los datos y no hay que instalar nada. Es el argumento decisivo. Un analista financiero puede pulsar Alt+F11 y automatizar su trabajo esta misma tarde, sin pedir permiso a nadie. Ninguna alternativa moderna iguala esa fricción cero.

2. El modelo de objetos de Office es enorme y muy estable. `Range`, `Worksheet`, `PivotTable`, `Chart`, `Document`, `MailItem`: décadas de API que casi no ha cambiado. Código de 2002 sigue funcionando.

3. Los volúmenes acumulados son gigantescos. Y son **invisibles**: no están en repositorios, no tienen pruebas, no pasan por revisión. Están dentro de ficheros `.xlsm` en carpetas compartidas. Nadie puede siquiera inventariarlos, y mucho menos migrarlos.

4. `Currency`: un tipo decimal exacto, integrado. Es un entero de 64 bits escalado con cuatro decimales fijos. Para dinero es exacto, igual que el `COMP-3` de COBOL, y evita el error clásico de acumular céntimos en punto flotante.

5. Las alternativas tienen huecos reales. Office Scripts no llega a todas las plataformas ni a todo el modelo de objetos; Python en Excel se ejecuta en la nube y no automatiza la interfaz; ninguna de las dos sirve para AutoCAD o SolidWorks.

> ⚠️ ***Un aviso, porque este es el único lenguaje de la lista con implicaciones de seguridad***
>
> *Las macros de VBA son el vector histórico número uno de compromiso vía documento. Reglas prácticas:* ***nunca habilites macros de un fichero cuyo origen no controles****; mantén el bloqueo de macros en ficheros procedentes de Internet;* ***firma digitalmente*** *las tuyas y usa ubicaciones de confianza; y no confundas "esta hoja funciona" con "esta hoja es segura". Aprender VBA incluye aprender esto.*

🔄 Lo que se ha modernizado

Del lenguaje, poco; **de su entorno, bastante**:

- **VBA 7 con 64 bits.** Las declaraciones de API externas necesitan `PtrSafe` y el tipo `LongPtr`, y la compilación condicional `#If VBA7 Then` permite que el mismo código funcione en Office de 32 y 64 bits. Es el cambio que rompió miles de macros heredadas.
- **Convivencia con lo nuevo**: desde VBA se pueden consumir APIs REST (`MSXML2.XMLHTTP` o `WinHttpRequest`) y analizar JSON con bibliotecas de la comunidad, así que una macro puede hablar con un servicio moderno.
- **Power Query y el modelo de datos** absorbieron gran parte de lo que antes se hacía con macros de transformación: hoy, mucho del trabajo de limpieza no necesita VBA.
- **Office Scripts** (TypeScript) para escenarios web y automatización con Power Automate, y **Python en Excel** para análisis y visualización, ejecutado en la nube de Microsoft.
- **Herramientas de ingeniería sobre VBA**: **Rubberduck**, un complemento libre que añade refactorizaciones, análisis de código y **pruebas unitarias** al editor de VBA; y exportar los módulos a ficheros `.bas`/`.cls` permite versionarlos en Git.

El consejo práctico hoy: **mantener VBA para automatizar la aplicación de escritorio y las aplicaciones de terceros; usar Power Query, Office Scripts o Python para transformar datos.**

⚙️ Cómo se ejecuta hoy

VBA **solo** existe dentro de su anfitrión. No hay intérprete de línea de comandos ni ejecutable.

```
1. Abre Excel (o Word, Access, AutoCAD…)
2. Pulsa Alt + F11 para abrir el editor (VBE)
3. Insertar → Módulo
4. Escribe el código y pulsa F5, o llámalo desde una celda
5. Guarda el fichero como .xlsm (habilitado para macros)
```

Herramientas: el **VBE** integrado (que no ha cambiado desde 1998, con su Ventana Inmediato como REPL rudimentario y su Ventana Locales como inspector), y **Rubberduck** como complemento imprescindible para trabajar en serio. Para versionar, exportar los módulos a `.bas` y `.cls`.

🍼 El programa de la clase 041 en VBA

> ⚠️ ***Contrato adaptado, y declarado.*** *VBA no tiene* `stdin` *ni* `stdout`*: su entrada son las celdas y el usuario, y su salida son las celdas, la Ventana Inmediato o un cuadro de diálogo. El cálculo es el mismo; la forma de entrar y salir es la del anfitrión.* ***No se verifica en CI****: requiere una instalación de Office.*

```
Option Explicit

' Versión 1: un procedimiento que lee y escribe en la hoja.
Public Sub CalcularTotalVenta()
    Dim precio    As Currency
    Dim cantidad  As Double
    Dim descuento As Double
    Dim total     As Currency

    With ThisWorkbook.Worksheets("Ventas")
        precio    = .Range("A2").Value
        cantidad  = .Range("B2").Value
        descuento = .Range("C2").Value

        total = precio * cantidad * (1 - descuento)

        .Range("D2").Value = total
    End With

    Debug.Print "Total: " & Format(total, "0.00")
End Sub

' Versión 2: una FUNCIÓN DE HOJA. Esto es lo que hace único a VBA.
Public Function TOTALVENTA(precio As Currency, _
                           cantidad As Double, _
                           descuento As Double) As Currency
    TOTALVENTA = precio * cantidad * (1 - descuento)
End Function
```

Recorrido, línea a línea.

- `Option Explicit` **debe ser la primera línea de todo módulo**. Sin ella, VBA crea variables nuevas al vuelo cuando encuentra un nombre desconocido: un `cantiadd` mal tecleado se convierte en un `Variant` vacío con valor 0, el cálculo da cero y **no hay ningún error**. Es el mismo problema que `implicit none` resuelve en Fortran y `use strict` en Perl, y es la causa documentada de errores millonarios en modelos financieros.

- `Dim precio As Currency` — **`Currency` es el detalle importante.** Es un entero de 64 bits con cuatro decimales fijos, es decir, **aritmética decimal exacta**. `Double` sería punto flotante binario y acumularía error al sumar miles de importes. La regla es la misma de siempre: dinero en decimal, magnitudes físicas en flotante. VBA tiene el tipo correcto y casi nadie lo usa.
- `With ... End With` fija un objeto y permite escribir `.Range(...)` con el punto inicial. No es solo estética: cada acceso a la hoja cruza la frontera COM entre VBA y Excel y **es caro**. Reducir esos cruces es la primera regla de rendimiento en VBA, y es un caso muy concreto de la Parte 10: el coste de atravesar la frontera entre dos mundos.
- `ThisWorkbook.Worksheets("Ventas").Range("A2")` es el **modelo de objetos**: libro → hoja → rango. Aprender VBA es, en su mayor parte, aprender ese árbol. `ActiveSheet` y `Selection` también existen, y usarlos es la mala práctica más extendida del lenguaje: hacen que la macro dependa de dónde estaba el cursor.
- `Debug.Print` escribe en la **Ventana Inmediato** (Ctrl+G), que es el `print` de depuración.
- `Format(total, "0.00")` formatea. Ojo: `Format` **depende de la configuración regional**, así que en un equipo con coma decimal produce `27000,00`. Para salida estable conviene `Format$` con máscaras explícitas o construir la cadena a mano.
- El guion bajo `_` al final de línea es la **continuación de línea**.

Y la versión 2 es la que hay que entender. Al declarar una `Public Function` en un módulo estándar, esa función **aparece en la lista de fórmulas de Excel**. En cualquier celda se puede escribir:

```
=TOTALVENTA(A2; B2; C2)
```

Acabas de **extender el lenguaje de fórmulas de la hoja de cálculo** con una función propia, que se recalcula sola cuando cambian sus argumentos. Eso es una *UDF* (*User Defined Function*), y es el puente exacto entre "usuario de Excel" y "programador": la hoja pasa de ser una tabla a ser un lenguaje al que le has añadido una primitiva.

🔍 Qué reconocer si vienes de otro lenguaje

Si conoces...	En VBA es...
`let x = 5`	`Dim x As Long` … `x = 5`
`def f(a):` sin retorno	`Sub f(a) ... End Sub`
`def f(a):` con retorno	`Function f(a) As Tipo` … `f = valor` … `End Function`
`return valor`	Asignar al **nombre de la función**: `f = valor`
`decimal` / `BigDecimal`	`Currency` (4 decimales) o `Decimal` dentro de un `Variant`
`any` / tipado dinámico	`Variant` — el tipo por defecto si no declaras
`null`	`Empty`, `Null`, `Nothing` y `vbNullString`: **cuatro cosas distintas**
`for x in lista`	`For Each x In coleccion ... Next x`
`try / catch`	`On Error GoTo Manejador` — manejo de errores por salto, no por bloque
`print(x)`	`Debug.Print x` (Inmediato) o `MsgBox x` (cuadro de diálogo)
`import`	Referencias del proyecto (Herramientas → Referencias)
Objeto	Objeto COM, creado con `New` o `CreateObject("...")`

⚠️ Errores comunes al leerlo

- **Falta `Option Explicit`.** Si no está, desconfía de todo lo demás.

- `Select` y `Activate` **por todas partes.** El grabador de macros genera código que selecciona celdas antes de tocarlas. Es lento y frágil; el código correcto trabaja con referencias directas a `Range` sin seleccionar nada.
- `Double` **para dinero.** Error de dominio con consecuencias contables.
- **Confundir** `Empty`, `Null` y `Nothing`. `Empty` es una variable sin inicializar, `Null` es un valor ausente de base de datos, `Nothing` es una referencia de objeto sin asignar. `IsEmpty`, `IsNull` e `Is Nothing` son las tres pruebas, y no son intercambiables.
- **Olvidar** `Set` **al asignar objetos.** `Set hoja = Worksheets(1)` lleva `Set`; `x = 5` no. Omitirlo da el error 91, el más frecuente del lenguaje.
- `On Error Resume Next` **como red general.** Silencia todos los errores, incluidos los que indicaban que el resultado es incorrecto. Se usa para casos puntuales y se desactiva de inmediato con `On Error GoTo 0`.
- **Suponer que el índice empieza en 0.** Los arrays de VBA empiezan en **1** si hay `Option Base 1`, en 0 si no, y `Range.Value` devuelve un array **de base 1** en ambos casos. Usa siempre `LBound` y `UBound`.

📚 Fuentes y bibliografía

- Referencia del lenguaje VBA (Microsoft Learn) (https://learn.microsoft.com/office/vba/api/overview/language-reference) — la documentación oficial vigente del lenguaje.
- Modelo de objetos de Excel (Microsoft Learn) (https://learn.microsoft.com/office/vba/api/overview/excel/object-model) — el árbol de objetos que constituye, en la práctica, la biblioteca estándar.
- Bloqueo de macros en ficheros de Internet (https://learn.microsoft.com/deployoffice/security/internet-macros-blocked) — la política de seguridad actual; léela antes de distribuir nada.
- Rubberduck VBA (https://rubberduckvba.com/) — refactorizaciones, análisis de código y pruebas unitarias dentro del editor de VBA. Cambia por completo la experiencia.
- **Rob Bovey, Dennis Wallentin, Stephen Bullen, John Green**, *Professional Excel Development*, 2.ª ed., Addison-Wesley — el único libro que trata Excel + VBA como ingeniería de software seria: arquitectura, distribución, rendimiento y errores.
- **Chip Pearson**, cpearson.com (http://www.cpearson.com/excel/topic.aspx) — archivo de referencia técnica sobre el modelo de objetos y sus rincones.
- **European Spreadsheet Risks Interest Group**, eusprig.org (https://eusprig.org/) — recopilación de errores documentados en hojas de cálculo y sus consecuencias. Lectura obligada para entender por qué el rigor importa aquí tanto como en cualquier otro sitio.

⏮️ Volver al Atlas · 🧟 Los lenguajes que siguen vivos · 🔗 Relacionadas: AutoLISP · Delphi / Object Pascal · COBOL

🟦 VB.NET — 2002

VB.NET es el heredero de **Visual Basic**, el lenguaje que más gente ha usado para escribir su primer programa que hacía algo útil. Su historia contiene una de las rupturas de compatibilidad más traumáticas que ha vivido un ecosistema — y también la prueba de que **la accesibilidad es una característica técnica**, no un adorno.

> 🎯 ***Por qué está en este programa***
>
> *VB.NET es un **primo de la familia .NET** (Atlas), cuyo representante en el núcleo es C#; y es el pariente moderno de VBA, que sí tiene ficha entre los lenguajes vivos.*

> *Aporta al programa dos cosas:* ***la demostración de que en el CLR el lenguaje es intercambiable*** *—VB.NET y C# compilan al mismo IL y comparten biblioteca (clase 157)— y el* ***caso de estudio de una migración forzada****, VB6 a VB.NET, que la clase 175 usa como advertencia.*

Año	2002 (VB.NET); **VB1** era de 1991 y **VB6**, de 1998
Autoría	Microsoft; VB original inspirado en **QuickBASIC** de Alan Cooper
Familia	.NET; descendiente de BASIC por vía de Visual Basic
Paradigma	Orientado a objetos, imperativo y dirigido por eventos
Tipado	**Estático**, con `Option Strict Off` opcional que lo relaja
Memoria	La del CLR: recolección de basura
Ejecución	Bytecode IL sobre el CLR, con JIT
Estado	🟡 **En mantenimiento**: Microsoft no añade características desde 2020

📜 Historia

Visual Basic (1991) hizo algo que ningún lenguaje había hecho: **puso el diseñador visual y el lenguaje juntos**, de forma que alguien sin formación podía arrastrar un botón, hacer doble clic y escribir lo que debía pasar. Es el modelo que después copió Delphi —y lo hizo mejor— y del que salió toda la programación dirigida por eventos (clase 120).

Millones de personas escribieron su primer programa útil en VB, y buena parte del software interno de las empresas de los noventa está escrito en él. **VB6 (1998)** fue el punto culminante.

Y entonces llegó **.NET (2002)**, y con él **VB.NET** — que **no era compatible con VB6**. El modelo de objetos cambiaba, los formularios cambiaban, los tipos cambiaban. Había herramientas de migración y no bastaban.

La reacción fue una de las mayores rebeliones de usuarios de la historia de Microsoft: hubo peticiones firmadas por decenas de miles de desarrolladores pidiendo que se mantuviera VB6, y el resentimiento duró años. Muchos equipos **no migraron**: se quedaron en VB6 —cuyo tiempo de ejecución Microsoft siguió soportando en Windows durante dos décadas— o se fueron a otro sitio.

> ***Es exactamente la lección de la clase 175 y de la 143: una ruptura de compatibilidad técnicamente justificada puede costar el ecosistema****. Lo mismo le pasó a Python 2→3 —doce años de migración—, a D con D2, a Scala 3 y a Perl con la saga de Perl 6.*

En **2020**, Microsoft anunció que **VB.NET seguirá soportado pero ya no evolucionará**: las características nuevas del lenguaje irán a C# y a F#.

🏭 Dónde vive hoy

- **Aplicaciones internas de empresa** escritas entre 2002 y 2015, muchas todavía en mantenimiento activo.
- **Administración pública** en varios países, con aplicaciones de gestión de largo recorrido.
- **Como puente desde VBA**: quien automatiza Excel reconoce la sintaxis inmediatamente.
- **Y en enseñanza**, en algunos programas de formación profesional.

🧠 Lo que enseña: el lenguaje como decisión intercambiable

Este es el punto que hace interesante la ficha (clase 157):

```
VB.NET, C# y F# compilan al MISMO bytecode (IL).
Comparten:
  - el sistema de tipos común (CTS)
  - la biblioteca de clases base entera
  - el recolector de basura y el modelo de excepciones
  - y las herramientas: depurador, perfilador, gestor de paquetes
```

Una biblioteca escrita en C# se usa desde VB.NET sin envoltorio, sin conversión y sin fricción — lo mismo que ILE en IBM i y Language Environment en z/OS (clase 157).

Y la consecuencia es la de la clase 155: en el CLR, elegir lenguaje es una decisión de equipo, no de arquitectura. Se pueden mezclar en el mismo sistema, componente a componente.

Y VB.NET tiene una característica propia que merece conocerse:

```
Option Strict On       ' ← comprobación estricta: conversiones explícitas obligatorias
Option Strict Off       ' ← permite conversiones implícitas y enlace tardío
```

El mismo lenguaje, con o sin red. `Option Strict Off` permite escribir código dinámico al estilo de VB6 — cómodo y peligroso—, y `On` lo convierte en un lenguaje estático estricto.

Es tipado gradual por fichero (clase 146), y la recomendación universal del ecosistema es **activarlo siempre**, con las excepciones justificadas.

🔄 Estado actual

- **Soporte indefinido**, sin características nuevas desde 2020; sigue funcionando en .NET moderno, multiplataforma.
- **Interoperabilidad total** con lo que se escriba en C# o F#, que es la vía de modernización recomendada: **no migrar el lenguaje, escribir lo nuevo al lado** (clases 150 y 175).
- **Herramientas de conversión** VB.NET → C# que funcionan razonablemente bien, precisamente porque el modelo subyacente es el mismo.
- **Y VB6**, que sigue ejecutándose en Windows por compatibilidad, aunque el entorno de desarrollo lleve más de veinte años sin soporte.

⚙️ Cómo se ejecuta hoy

```
dotnet run                            # con un proyecto .vbproj
vbc Venta.vb && ./Venta                # compilador de línea de comandos

dotnet test                            # pruebas (clase 139)
```

🍼 El programa de la clase 041 en VB.NET

```
Imports System.Globalization

Module Venta
    Sub Main()
        Dim v = Console.ReadLine().Split(" "c)
        Dim precio = Double.Parse(v(0), CultureInfo.InvariantCulture)
        Dim cantidad = Double.Parse(v(1), CultureInfo.InvariantCulture)
        Dim descuento = Double.Parse(v(2), CultureInfo.InvariantCulture)
        Console.WriteLine("Total: " & (precio * cantidad * (1 - descuento)).ToString("F2",
CultureInfo.InvariantCulture))
    End Sub
End Module
```

Lo que hay que ver.

- **La sintaxis con palabras en lugar de símbolos** — `Module` / `End Module`, `Sub` / `End Sub`, `Dim` — es la herencia de BASIC, y su motivo es el de la ficha: **legibilidad para quien empieza**. Compárese con la versión de C#, que hace exactamente lo mismo con llaves.
- `&` **es la concatenación de cadenas**, no `+`. Es una decisión de VB para **evitar la ambigüedad** de que `"1" + "2"` pueda ser `"12"` o `3` — un problema real en JavaScript y PHP (clase 100). Es una buena idea que casi nadie copió.
- `CultureInfo.InvariantCulture`**, otra vez**, y por la misma razón que en C# y Java: en configuración regional española el separador decimal es la coma, y sin esto el programa falla o imprime mal. **Es la trampa número uno de .NET.**
- `Dim v = ...` **sin tipo** usa inferencia, no tipado dinámico: con `Option Strict On`, el tipo se deduce y se comprueba igual que con `var` en C#.

📚 Fuentes y bibliografía

- Documentación de Visual Basic (https://learn.microsoft.com/dotnet/visual-basic/) — referencia oficial, incluido el comunicado de estrategia de 2020 (https://devblogs.microsoft.com/vbteam/visual-basic-support-planned-for-net-5-0/), que conviene leer como documento de decisión (clase 175).
- Guía de conversión VB.NET ↔ C# (https://learn.microsoft.com/dotnet/csharp/) — útil para leer código de los dos.
- **Michael Halvorson**, *Visual Basic Step by Step*, Microsoft Press — la línea clásica de introducción.
- **Anne Boehm, Bryan Syverson**, *Murach's Visual Basic* — orientado a aplicaciones de empresa.
- Y para el contexto histórico: los archivos de la campaña *Save VB6* (2005), que documentan lo que cuesta romper la compatibilidad de un ecosistema.

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: C# · VBA · BASIC · Delphi · F#

⚡ Zig — 2016

Zig es la propuesta más radical de las que intentan suceder a C: **no añadir seguridad automática, sino hacer visible todo lo que C esconde**. Cada reserva de memoria, cada operación que puede fallar y cada rama del flujo de control aparecen escritas — y a cambio no hay nada oculto.

> 🎯 ***Por qué está en este programa***
>
> *Zig es un* ***primo de la familia C / llaves*** *(Atlas) y aparece junto a los representantes de sistemas del núcleo, Go y Rust.*

> *Aporta al programa dos ideas que ningún otro lenguaje del curso enseña igual:* ***el asignador de memoria como parámetro explícito*** *—quien llama decide de dónde sale la memoria (clases 128 y 130)— y* `comptime`*, ejecutar código del propio lenguaje en tiempo de compilación en lugar de tener un lenguaje de macros aparte (clase 122).*

Año	2016; en **0.x** — aún no ha llegado a 1.0
Autoría	**Andrew Kelley**; hoy con la Zig Software Foundation
Familia	Sistemas / C; sin herencia sintáctica de C++
Paradigma	Imperativo y procedimental; sin clases ni herencia
Tipado	**Estático y fuerte**, con genéricos vía `comptime` y tipos opcionales
Memoria	**Manual y explícita**, con asignadores que se pasan como argumento
Ejecución	Compilado a nativo (LLVM y motor propio); **compilador cruzado universal**
Estado	🟡 **Preestándar**: se usa en producción con cuidado, y la API cambia

---

📜 Historia

Andrew Kelley empezó Zig en **2015** por frustración con C y con C++: quería un lenguaje donde **no hubiera control de flujo oculto** —nada de constructores, destructores, sobrecarga de operadores ni excepciones— y donde **cada reserva de memoria fuera visible**.

El proyecto creció con una propiedad inesperada que lo hizo conocido antes que el lenguaje: `zig cc` **es un compilador de C excelente**, con **compilación cruzada a cualquier plataforma sin instalar nada más**.

```
zig cc -target aarch64-linux-musl hola.c    # desde cualquier máquina, sin cadena cruzada
```

Eso resolvió, de rebote, uno de los problemas más molestos de la clase 174, y hizo que mucha gente usara Zig como herramienta antes que como lenguaje. **Bun**, el tiempo de ejecución de JavaScript, está escrito en Zig.

El lenguaje sigue **antes de la 1.0**, con cambios incompatibles en cada versión —el sistema de entrada y salida se está rediseñando ahora mismo— y con el compilador migrando de LLVM a un motor propio.

🏭 Dónde vive hoy

- **Bun**: el tiempo de ejecución de JavaScript, escrito en Zig.
- **TigerBeetle**: base de datos financiera de alto rendimiento — un caso de uso serio y exigente.
- **Como cadena de herramientas de C**: `zig cc` y `zig build` en proyectos que no son de Zig.
- **Sistemas embebidos y desarrollo de núcleo**, por la ausencia de tiempo de ejecución.
- **Y en proyectos nuevos de infraestructura** que aceptan el riesgo de una API que aún cambia.

🧠 Lo que enseña: hacer visible lo que C esconde

Uno, el asignador explícito. En Zig, **una función que reserva memoria recibe de dónde sacarla**:

```
fn crearLista(asignador: std.mem.Allocator, n: usize) ![]u32 {
    return try asignador.alloc(u32, n);
}
```

Y eso cambia el diseño entero de la biblioteca estándar: no hay un montón global implícito. Quien llama decide si la memoria viene del montón, de una arena que se libera de golpe, de un búfer en la pila o de un asignador que detecta fugas en las pruebas (clases 130 y 139).

Es la idea que la clase 128 persigue —**saber de dónde sale la memoria**— convertida en firma de función.

Dos, los errores son valores y el flujo está escrito:

```
const valor = try puedeFallar();    // ← 'try' propaga el error; SIEMPRE se ve
```

No hay excepciones, como en Go, pero el error forma parte del tipo (`!u32`), así que **ignorarlo no compila** (clase 116).

Y tres, `comptime`:

```
fn Lista(comptime T: type) type {          // los genéricos son FUNCIONES sobre tipos
    return struct { items: []T };
}
```

El mismo lenguaje se ejecuta en tiempo de compilación para generar tipos, desplegar bucles o calcular tablas. Es la metaprogramación de la clase 123 **sin un lenguaje de macros aparte** — lo contrario del preprocesador de C y de las plantillas de C++.

> ***Y la comparación con Rust es la que importa*** *(clase 164):* ***Rust hace imposible el error de memoria; Zig lo hace visible.*** *Zig detecta el desbordamiento y el uso de memoria no inicializada* ***en modo depuración****, y en modo seguro comprueba límites — pero* ***no tiene comprobador de préstamos****, así que un uso después de liberar sigue siendo posible. Son dos apuestas distintas sobre el mismo problema, y merece conocer las dos antes de opinar.*

🔄 Hacia dónde va

- **Camino a la 1.0**, con el sistema de entrada y salida rediseñado y la API estabilizándose.
- **Compilador propio sin LLVM** para las plataformas principales, con compilación mucho más rápida.
- **`zig build`** como sistema de construcción escrito en el propio Zig — sin lenguaje de configuración aparte (clase 144).
- **Y el papel de puente**: `zig cc` y `zig c++` como cadena de herramientas cruzada para proyectos C y C++ existentes, que es hoy su uso más extendido.

⚙️ Cómo se ejecuta hoy

```
zig run main.zig < entrada.txt
zig build-exe main.zig -O ReleaseSafe        # con comprobaciones; o ReleaseFast
zig build test                                # pruebas, integradas en el lenguaje

zig cc -target x86_64-windows-gnu hola.c      # ← compilar C para otra plataforma
```

🍼 El programa de la clase 041 en Zig

```
const std = @import("std");

pub fn main() !void {
    var buf: [128]u8 = undefined;
    const linea = (try std.io.getStdIn().reader().readUntilDelimiterOrEof(&buf, '\n')).?;
    var it = std.mem.tokenizeScalar(u8, std.mem.trim(u8, linea, " \r"), ' ');
    const precio = try std.fmt.parseFloat(f64, it.next().?);
    const cantidad = try std.fmt.parseFloat(f64, it.next().?);
    const descuento = try std.fmt.parseFloat(f64, it.next().?);
    const total = precio * cantidad * (1 - descuento);
    try std.io.getStdOut().writer().print("Total: {d:.2}\n", .{total});
}
```

Lo que hay que ver, y es la ficha donde más se nota.

- `var buf: [128]u8` **reserva el búfer a mano, en la pila.** Ningún otro programa de la clase 041 hace eso: los demás piden una cadena y el lenguaje se ocupa. Aquí **el tamaño es una decisión visible**, y si la línea no cupiera, se vería (clase 128).
- `try` **aparece cinco veces**, y cada una es una operación que puede fallar. **El flujo de error está escrito en el código**, no escondido en excepciones (clase 116).
- `.?` **desempaqueta un opcional** y aborta si es nulo: Zig **distingue en el tipo** lo que puede faltar, como Rust con `Option` (clase 100).
- `!void` **en la firma de** `main` declara que la función puede devolver un error. El tipo de retorno **incluye el fallo**, que es la idea central del manejo de errores del lenguaje.
- **Comparado con la versión de C**, que hace lo mismo en seis líneas: Zig es más largo **a propósito**. Lo que C oculta —el búfer, el fallo de conversión, el nulo— aquí está escrito.

📚 Fuentes y bibliografía

- ziglang.org/documentation (https://ziglang.org/documentation/master/) — la referencia oficial; al ser preestándar, **usar siempre la de la versión que se tenga instalada**.
- Zig Learn (https://ziglearn.org/) y zig.guide (https://zig.guide/) — introducciones comunitarias.
- Notas de versión (https://ziglang.org/download/) — imprescindibles: cada versión trae cambios incompatibles y los documenta bien.
- **Andrew Kelley**, charlas *The Road to Zig 1.0* y *Practical Data Oriented Design* — la segunda explica la relación entre disposición de datos y rendimiento mejor que casi cualquier texto (clase 152).

⏮️ Volver al Atlas · 🗂️ Todas las fichas · 🔗 Relacionadas: C · Rust · Go · Nim · D
