

**Módulo 3: Compiladores: Tarea 2**

**Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)**

**Estudiante**

Bo Hyeon Cha- A01023804

**30/08/2025**

**Profesores:**

Andres Saldaña Rodriguez

Elda Guadalupe Quiroga González

Mauricio González Soto

**Tabla de contenidos**

[**1. Introducción 3**](#_7phrouydprbj)

[**2. Fundamentos teóricos 4**](#_wx6e1sa0dvpp)

[**3. Herramientas analizadas 5**](#_xxstdnv0mcik)

[3.1 Flex & Bison 5](#_tqbfwuo8ggg9)

[3.2 ANTLR 4 (ANother Tool for Language Recognition) 7](#_gsomuq3jngvk)

[3.3 Gocc (Go Compiler Compiler) 9](#_ijz06fawsv8w)

[**Bibliografía 10**](#_4vhi9rgb9as)

# 

# 1. Introducción

El presente trabajo analiza y compara tres herramientas de generación automática de analizadores léxicos y sintácticos (scanners y parsers). Se examinan aspectos fundamentales de cada una, como la plataforma y el lenguaje base, las características técnicas (formato de entrada y tipo de ejecución), el modelo de licenciamiento, los fundamentos teóricos (expresiones regulares y métodos sintácticos empleados), el tipo de interfaz y la facilidad de integración con código personalizado.

Las herramientas seleccionadas para el análisis son Flex & Bison, que representan las soluciones clásicas en la construcción de compiladores. También se incluye ANTLR 4 (ANother Tool for Language Recognition), reconocida como una alternativa moderna y multiplataforma. Finalmente, se estudia Gocc (Go Compiler Compiler), una herramienta orientada específicamente al ecosistema del lenguaje Go.

Este análisis comparativo busca ofrecer una guía práctica para la elección de herramientas en el desarrollo de un compilador de lenguaje imperativo, tomando en cuenta tanto los aspectos técnicos como los de implementación.

# 

# 2. Fundamentos teóricos

El proceso de compilación se compone de varias fases que permiten transformar un programa fuente en un programa ejecutable. Entre ellas, el análisis léxico y el análisis sintáctico y análisis semántico representan las etapas inicial, las cuales garantizan que el código de entrada se ajuste tanto al vocabulario como a las estructuras gramaticales definidas por el lenguaje (Aho, Lam, Sethi, & Ullman, 2006, p. 25). En este reporte se enfoca en los dos analizadores, analizador léxico y sintáctico.

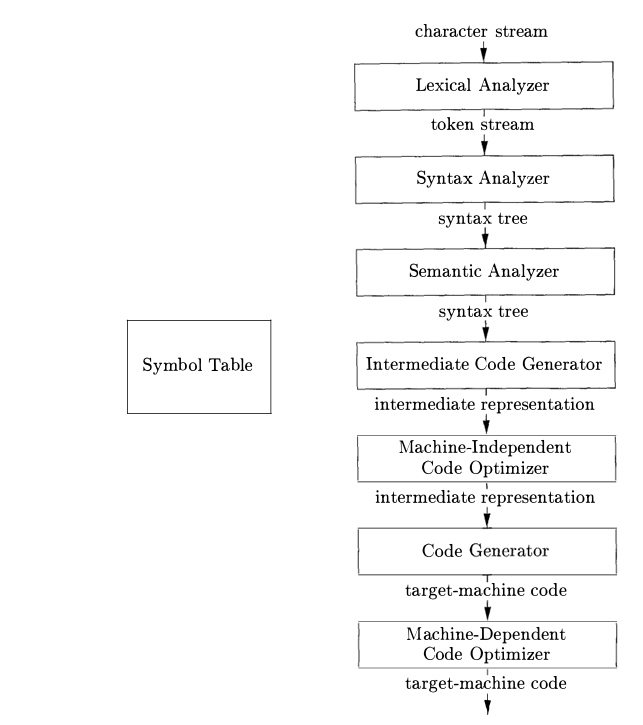


Figura 1. Fases del proceso de compilación. Adaptado de la Figura 1.6 del libro *Compilers: Principles, Techniques, and Tools (2ª ed.)*, por Aho, Lam, Sethi y Ullman (2006, p. 5).

**Análisis léxico:** Esta fase se encarga de leer el programa fuente y dividirlo en unidades mínimas con significado, llamadas tokens. Los tokens se describen mediante expresiones regulares y suelen clasificarse en identificadores, constantes, operadores y palabras reservadas. Para reconocerlos, se emplean autómatas finitos deterministas (DFA), que permiten un escaneo eficiente del texto.

**Análisis sintáctico:** Una vez reconocidos los tokens, el compilador verifica que su secuencia respete las reglas de una gramática libre de contexto. Estas gramáticas, que se pueden expresar de gramática de contexto libre o en BNF (Backus–Naur Form), permiten describir la estructura jerárquica de un lenguaje (Aho, Lam, Sethi, & Ullman, 2006, p. 191-197). A partir de ellas, el parser construye árboles de derivación o árboles sintácticos, que muestran cómo se organizan los elementos dentro de una instrucción.

Existen dos enfoques principales para el análisis sintáctico. El primero es el parsing descendente (Top-Down), que parte del símbolo inicial de la gramática y genera derivaciones hacia las hojas, con variantes como los parsers LL(1). El segundo es el parsing ascendente (Bottom-Up), que inicia desde los tokens de entrada y reduce hasta alcanzar el símbolo inicial, empleando métodos LR, SLR o LALR (Grune & Jacobs, 2008, caps. 7–10).

Finalmente, es importante destacar que tanto el análisis léxico como el sintáctico sientan las bases para las fases posteriores del compilador: validaciones semánticas, generación de código intermedio y optimización. Sin una correcta verificación en estas primeras etapas, las fases siguientes carecerían de consistencia y confiabilidad.

# 3. Herramientas analizadas

En este apartado se describen tres herramientas de generación automática de analizadores léxicos y sintácticos. La selección incluye una herramienta clásica, una alternativa moderna multiplataforma y una opción propia del ecosistema Go.

## 3.1 Flex & Bison

* **Plataforma y lenguaje base:** Flex (Fast Lexical Analyzer) y Bison (GNU Bison) son herramientas clásicas en sistemas tipo Unix. Flex y Binson fueron desarrollados como una alternativa compatible de Lex&Yacc (GNU Project, 2021; Paxson, 1995).
  + De acuerdo con Paxson (1995), está escrito en C y genera código en C por defecto, aunque también permite generar código en C++ mediante Flex++.
  + Bison, por su parte, es una herramienta escrita en C que genera código en C por defecto, heredando el enfoque de Yacc, pero ha evolucionado para ofrecer soporte a otros lenguajes como C++, Java y D (GNU Project, 2021).
* **Características básicas:** 
  + Flex lee un conjunto de patrones lexicográficos (expresiones regulares) con código en C, llamado “reglas”(en el documento de manual de Flex decía rules). Flex genera como salida un programa en C que reconoce esos patrones (un scanner o lexer, en forma de archivo “lex.yy.c”).
  + Bison lee una especificación de gramática (formato estilo Yacc, similar a BNF con acciones en C) y genera un parser LALR(1) en C. Típicamente se usan en conjunto: el lexer generado por Flex proporciona tokens al parser generado por Bison. El parser usa un esqueleto de código (que implementa la máquina de parsing) combinado con las reglas de gramática del usuario.
* **Tipo de licenciamiento:** 
  + Flex es un software libre bajo licencia BSD. Sin embargo, de acuerdo con el dueño de repositorio de Flex actual (Will Estes, 2025), Flex posee los derechos de autor utilizados para el software BSD, ligeramente modificados porqué se originó en el Laboratorio Lawrence Berkeley, que opera bajo un contrato con el Departamento de Energía:
    - Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006, 2007 The Flex Project.
    - Copyright (c) 1990, 1997 The Regents of the University of California.
  + Bison se distribuye bajo GNU GPL (General Public License) pero con una excepción especial que permite utilizar el código generado en proyectos no GPL sin contagio (esta excepción cubre el skeleton incluido en el código generado).
    - Para comprobar que el proyecto está en la excepción especial, solo necesita revisar que el archivo de salida de bison incluya la frase “As a special exception…”.
* **Fundamentos teóricos:** 
  + El scanner de Flex se basa en expresiones regulares que se convierten en un autómata finito determinista (DFA) para reconocer tokens. Bison implementa un algoritmo LR(1) por defecto optimizado a LALR(1) (Look-Ahead LR con un símbolo de anticipación).
  + Genera tablas de análisis sintáctico para un parser ascendente (bottom-up). Bison también puede producir variantes canonical LR, GLR (Generalized LR) para gramáticas más complejas (GNU Project, 2021). Las gramáticas aceptadas son gramáticas libres de contexto LR.
    - Bison/Yacc manejan recursión a la izquierda directamente, pero requieren gramáticas libres de conflictos (aunque Bison reporta y puede generar contraejemplos para resolver conflictos shift/reduce, estos rasgos se puede consultar con mayor detalle en el capítulo 5 sección 2 a 7).
* **Tipo de interfaz:** 
  + Son herramientas de línea de comandos. El flujo típico es escribir un archivo de especificación léxica (.l) para Flex y un archivo de gramática (.y) para Bison. Luego se ejecuta flex ejemplo.l y bison -d ejemplo.y (el -d opcional genera un header con definiciones de tokens). Flex produce código C (ej. lex.yy.c) y Bison produce código C (ej. ejemplo.tab.c y ejemplo.tab.h). Estos se compilan con un compilador C junto con el código de apoyo del usuario (p. ej., una función main) (Flex Manual, 2016; GNU Bison Manual, 2021).
    - A pesar de que en el manual más nuevo de de flex menciona yacc en lugar de Bison, esta parte es compatible con Bison también. Por tanto, no debe provocar ningún problema.
* **Facilidad para añadir código propio:** 
  + Flex permite incrustar código en C asociadas a patrones léxicos (entre { ... } después de cada patrón) (Flex Manual, 2016). El usuario puede acceder al texto del lexema mediante yytext y su longitud con yyleng.
  + En Bison, se pueden agregar acciones semánticas en C encerradas en { ... } después de las producciones (GNU Project, 2021, sección Semantic Actions). A través de estas acciones se construyen ASTs, se manejan conteos, etc. Bison proporciona mecanismos como una unión %union para definir tipos de valor de símbolos ($$, $1, etc.) y directivas %token y %type para declarar el tipo de cada token o no-terminal. Además, se pueden insertar bloques de código C en la sección de definiciones (entre %{ %} al inicio) para incluir bibliotecas, definiciones de estructuras, etc. Esta integración permite al usuario extender la funcionalidad del analizador con lógica propia (por ejemplo, construir una tabla de símbolos en las acciones, manejar errores personalizados con yyerror, etc.).
    - Después de enlazar con C se puede enlazar directamente a lenguaje de programación go mediante CGO (Go Project, 2025). Existen alternativas como la comunicación vía socket.

## 

## 3.2 ANTLR 4 (ANother Tool for Language Recognition)

* **Plataforma y lenguaje base:** ANTLR v4 es un generador de analizadores léxicos/sintácticos escrito en Java, de modo que su herramienta de generación (ANTLR Tool) corre en la JVM (Java 8+) (ANTLR Project, 2023). Es multiplataforma (cualquier sistema con Java). ANTLR soporta múltiples lenguajes destino, incluyendo Go. Es decir, puede generar código de lexer y parser en Go, además de otros como Java, C#, C++, Python, JavaScript, etc.
  + Por ejemplo, al ejecutar ANTLR se especifica la opción -Dlanguage=Go para obtener un parser Go. El runtime correspondiente (una librería en Go con las clases de apoyo) es provisto también por el proyecto ANTLR .
* **Características básicas:** ANTLR utiliza un único archivo de gramática (.g4) para definir tanto las reglas léxicas (lexer rules) como las reglas sintácticas (parser rules).
  + La sintaxis de gramática de ANTLR 4 se basa en EBNF, lo que la hace bastante legible: permite notación de operadores \* (cero o más), + (uno o más), ? (opcional) y agrupaciones (Parr, 2013, p. 89). Las reglas léxicas se identifican porque comienzan con mayúscula, y las sintácticas con minúscula.
  + ANTLR genera como salida código fuente en el lenguaje elegido, junto con un runtime o biblioteca que implementa el algoritmo de parsing.
    - En Go, el runtime de ANTLR se distribuye como un paquete llamado *antlr*, que proporciona utilidades para manejar tokens, árboles de sintaxis, errores y flujos de entrada (ANTLR Project, 2025).
  + Puede generar diferentes tipos de reconocedores: solo lexer, solo parser, o ambos combinados según la gramática escrita. Por defecto, produce clases de Lexer y Parser que pueden construir un árbol de sintaxis concreto (parse tree). Opcionalmente, ANTLR puede construir automáticamente un árbol abstracto (AST) o el usuario puede implementar listeners/visitors para recorrer el árbol y extraer semántica.
* **Tipo de licenciamiento:** ANTLR v4 es software libre bajo licencia BSD 3-clause (BSD de 3 cláusulas). Esto incluye tanto la herramienta como el código runtime. Las versiones antiguas (<v3) estaban en dominio público. La licencia BSD permite uso amplio con pocas restricciones.
* **Fundamentos teóricos:** Parr, Harwell y Fisher (2014) indican queANTLR 4 emplea un algoritmo LL(\*) adaptativo, conocido como ALL(\*).
  + Es un algoritmo de análisis descendente (top-down recursive-descent) que generaliza LL(k): en lugar de un k fijo, utiliza lookahead no acotado de forma dinámica y estrategias de backtracking con predicción. Esto le permite reconocer lenguajes más allá de LL(1) típicas, evitando al programador tener que hacer mucho left-factoring. ANTLR 4 incluso soporta recursión izquierda directa en las gramáticas (transformándose internamente para evitar el bucle infinito).
    - La misma fuente señala que, aunque ALL(\*) teóricamente tiene complejidad de O(n⁴) en muchos casos su rendimiento observado es lineal en gramáticas reales. Sin embargo, no maneja recursión izquierda indirecta (mutua) automáticamente. En general, las gramáticas aceptadas por ANTLR son gramáticas libres de contexto que pueden ser analizadas por un parser LL(). Muchas gramáticas ambiguas pueden manejarse con árboles de sintaxis alternativos o predicados semánticos si es necesario.
  + En la práctica, ANTLR cubre la mayoría de los lenguajes de programación modernos. El repositorio oficial de ANTLR contiene más de 200 gramáticas de ejemplo en su repositorio, incluyendo C, Java, Go, Python, etc. (ANTLR Project, 2025). Las reglas léxicas en ANTLR se basan en expresiones regulares tradicionales (definidas en el mismo archivo de gramática), y generan autómatas DFA optimizados con soporte completo Unicode. Para la fase sintáctica, ANTLR genera un parser recursivo-descent en Go que utiliza predicción LL().
* **Tipo de interfaz:** La herramienta ANTLR se usa por línea de comando (normalmente invocando java -jar antlr-4.xx-complete.jar). También hay plugins de IDE y GUIs que facilitan escribir y depurar gramáticas, pero el uso básico es: escribir archivo .g4, ejecutarlo con ANTLR para generar código, luego compilar este código junto con el runtime.
  + En contexto Go, el flujo sería:
    - (1) Definir gramática X.g4.
    - (2) Ejecutar antlr -Dlanguage=Go X.g4 que genera archivos Go (ej. X\_lexer.go, X\_parser.go, etc.).
    - (3) Usar esos archivos en un programa Go, importando el paquete runtime github.com/antlr/antlr4/runtime/Go/antlr.
  + La interacción con el parser típicamente consiste en crear un *InputStream* de ANTLR a partir de la entrada, alimentar ese stream al lexer generado, obtener un stream de tokens, y pasarlo al parser para producir el árbol de análisis (Brampton, 2017).
  + ANTLR ofrece dos modos de trabajar con el resultado: usando el patrón Listener/Visitor que el parser genera eventos al entrar/salir de cada regla que pueden ser manejados por nuestros tipos (implementando interfaces) o insertando acciones en línea en la gramática (en el lenguaje destino) si se desea lógica inmediata. En general, se recomienda el enfoque listener/visitor para separar la gramática de la semántica (Tomassetti, .·
* **Facilidad para añadir código propio:** ANTLR permite acciones semánticas embebidas dentro de las reglas (encerrándolas entre { } dentro de la gramática). Estas acciones son código Go si el destino es Go, y se insertarán tal cual en el parser generado (Parr, 2013).
  + No obstante, un enfoque más limpio es generar el árbol de parse completo y luego escribir un listener o visitor en Go que recorra ese árbol.
  + ANTLR genera por default interfaces como YourGrammarListener con métodos Enter/Exit para cada regla, y structs base con implementaciones vacías. El usuario puede implementar dichos métodos para agregar comportamiento (por ejemplo, construir un AST manualmente, o interpretar las construcciones). Esto hace muy fácil añadir código propio sin tocar la gramática: uno simplemente escribe una estructura en Go que embeba BaseYourGrammarListener y sobreescriba los métodos de las reglas de interés.
  + Adicionalmente, ANTLR soporta *predicados semánticos*: condiciones en línea (en { ... }?) que se evalúan durante el parseo para decidir alternativas, lo cual permite incorporar lógica específica del lenguaje en el proceso de reconocimiento (por ejemplo, verificar contextos semánticos sencillos). En resumen, ANTLR brinda flexibilidad: se puede optar por mezclar código en la gramática (útil para prototipos), o mantener gramática pura y manejar la semántica en una fase separada usando los patrones de diseño proveídos.

## 3.3 Gocc (Go Compiler Compiler)

* **Plataforma y lenguaje base:** Gocc es una herramienta escrita en Go (golang) diseñada para generar analizadores léxicos y sintácticos en Go. Funciona en cualquier plataforma soportada por Go. El código resultante (scanner, parser) está en Go puro, facilitando su integración directa en proyectos Go.
* **Características básicas:** Gocc toma como entrada un archivo de gramática en una combinación de EBNF/BNF que incluye reglas léxicas y sintácticas en un mismo fichero (extensión .bnf).
  + El formato permite separar una sección léxica (definiciones de tokens con expresiones regulares entre comillas o rangos de caracteres) y una sección sintáctica (producciones de gramática libre de contexto). Las acciones semánticas en Gocc se escriben directamente en código Go embebido entre marcadores << >> dentro de las producciones.
  + Al ejecutar gocc <archivo>.bnf, la herramienta genera varios archivos Go: un paquete lexer, un paquete parser y un paquete token (más cualquier código de apoyo necesario) (Ackerman, 2025).
  + También produce un archivo AST de ejemplo si se definieron acciones. El lexer generado es un autómata finito determinista (DFA) que reconoce los patrones regulares definidos, aceptando texto UTF-8 (con soporte opcional para case-insensitive). El parser generado es un autómata de pila (PDA) que reconoce lenguajes LR(1) (por lo tanto puede manejar gramáticas LR más generales que LALR, reduciendo la necesidad de factorizar o eliminar recursión izquierda). Gocc incluye manejo automático de conflictos LR(1): si se detectan conflictos shift/reduce o reduce/reduce, puede aplicar resoluciones por prioridad o elegir la primera producción para resolver ambigüedades (Gocc Project, 2025).
* **Tipo de licenciamiento:** Es software de código abierto bajo licencia Apache 2.0. Esto permite uso libre en proyectos open source o propietarios con mínimas restricciones (requiere mantener avisos de licencia).
* **Fundamentos teóricos:** Gocc soporta gramáticas libres de contexto LR(1) completas, dado que implementa un algoritmo LR(1) clásico (con tablas de parsing similares a Yacc/Bison pero con un lookahead completo de 1 símbolo). Al ser LR(1) real, puede resolver más casos sin transformaciones que un LALR(1). Las reglas gramáticales se expresan en (E)BNF. Se permiten operadores EBNF como { } (repetición) o [...] (opcionales) en la gramática. Para el análisis léxico, Gocc utiliza expresiones regulares clásicas para definir tokens. Internamente genera un DFA eficiente para el scanner (Gocc Project, 2025).. Las acciones semánticas incrustadas se ejecutan en puntos definidos de la derivación, permitiendo construir estructuras o ASTs conforme se va reconociendo la entrada. De hecho, Gocc recomienda definir tipos Go para representar nodos del AST y funciones constructoras en un paquete separado (por ejemplo, un paquete ast proporcionado por el usuario) e invocarlas desde las acciones semánticas.
* **Tipo de interfaz:** Gocc se usa por línea de comando. Tras instalarlo (go install github.com/goccmack/gocc@latest), el comando gocc archivo.bnf genera el código en el directorio actual (o en el mismo módulo Go). No es una biblioteca a la que se le pase la gramática en tiempo de ejecución, sino una herramienta de generación en tiempo de desarrollo. El output son ficheros Go que el desarrollador luego integra y compila en su proyecto. El parser generado expone una interfaz sencilla: típicamente una función Parse(lexer) que recibe una instancia de lexer (el cual a su vez recibe la entrada a analizar). También se genera código para los tokens (token.Token) y utilidades para posicionamiento o errores léxicos/sintácticos.
* **Facilidad para añadir código propio**: Gocc facilita la inclusión de código del usuario en varios niveles. En la gramática .bnf, se pueden incluir bloques de código Go arbitrario usando la sintaxis << ... >> dentro de las producciones (acciones semánticas). Ese código se inserta tal cual en el parser generado. En la sección de gramática también se pueden incluir declaraciones de import (<< import "pkg" >>) para traer paquetes necesarios en las acciones. Adicionalmente, el usuario suele proporcionar un paquete (por convención llamado ast) con definiciones de estructuras y funciones para construir los nodos del árbol sintáctico abstracto. Las acciones en la gramática invocan esas funciones (como << ast.NewNode($1, $3) >>) pasando atributos $n correspondientes a símbolos reconocidos. Esto permite escribir lógica de construcción semántica de manera limpia. Por otra parte, el código fuente generado es en Go legible, y el usuario podría modificarlo o extenderlo si fuese necesario (aunque normalmente no se edita manualmente el código generado). Gocc también permite personalizar el lexer generado: por ejemplo, uno puede regenerar solo el parser (gocc con flags) y mantener un lexer modificado a mano, si se desea un manejo más complejo en la fase léxica. En resumen, Gocc ofrece puntos de extensión claros: acciones en la gramática y uso de paquetes auxiliares para la lógica del compilador.

# 

# Bibliografía

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles,*

*Techniques, and Tools* (2nd ed.). Pearson. (Véase cap. 4 para LL y LR).

ANTLR Project. (2023). *ANTLR v4 documentation*. Recuperado de https://www.antlr.org

ANTLR Project. (2025). *Package antlr – Go ANTLR runtime*. Go Packages.

https://pkg.go.dev/github.com/antlr/antlr4/runtime/Go/antlr

ANTLR Project. (2025). *ANTLR grammars repository*. GitHub.

https://github.com/antlr/grammars-v4

Brampton, A. (2017, diciembre 10). Parsing with ANTLR 4 and Go. *Gopher Academy Blog*.

https://blog.gopheracademy.com/advent-2017/parsing-with-antlr4-and-go/

Estes, W. (2025). *Perfil profesional en LinkedIn*. LinkedIn. Recuperado el 3 de octubre de

2025, de https://www.linkedin.com/in/willestes/

GNU Project. (2021, septiembre 11). *Bison Manual: Yacc* (Versión 3.8.1). Free Software

Foundation. Recuperado de https://www.gnu.org/software/bison/manual/html\_node/Yacc.html

Go Project. (2025). *cgo command – cmd/cgo*. Go Packages. <https://pkg.go.dev/cmd/cgo>

Goccmack. (2025). \*gocc: Parser / Scanner Generator\* [Repositorio GitHub]. GitHub.

<https://github.com/goccmack/gocc>

Gocc Project. (2014). Learn You a gocc for Great Good (User Guide).

https://raw.githubusercontent.com/goccmack/gocc/master/doc/gocc\_user\_guide.pdf

Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide* (2nd ed.).

Springer.

Parr, T. (2013). *The definitive ANTLR 4 reference* (p. 89). Pragmatic Bookshelf.

Parr, T., Harwell, S., & Fisher, K. (2014). Adaptive LL(\*) parsing: The power of dynamic

analysis. *ACM SIGPLAN Notices, 49*(10), 579–598. https://doi.org/10.1145/2714064.2660202

Paxson, V. (1995). *Flex: A fast lexical analyzer generator* (Versión 2.5). The Regents of the

University of California. https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html\_mono/flex.html

Tomassetti, G. (2025). *Listeners and visitors*. Strumenta.

https://tomassetti.me/listeners-and-visitors/?utm\_source=chatgpt.com