Compiladores
Unidad 1: Introducción

Historia de los Compiladores

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

De forma intuitiva, podemos decir que un compilador es un programa que traduce código escrito en un lenguaje de programación a otro lenguaje, usualmente de más bajo nivel, para que pueda ser entendido y ejecutado por una computadora. Este proceso de traducción no es trivial: implica comprender la estructura, el significado y el propósito del programa original, para luego generar un equivalente correcto y eficiente. Los compiladores son fundamentales en el desarrollo de software, ya que permiten a las personas programadoras escribir en lenguajes de alto nivel, más cercanos al pensamiento humano, y delegar la tarea de generar instrucciones comprensibles para la máquina.

La teoría de compiladores se dedica al estudio formal y sistemático de cómo se lleva a cabo esta traducción. Analiza las etapas fundamentales del proceso de compilación (como el análisis léxico, sintáctico, semántico, la generación de código intermedio y final, y la optimización), y proporciona los fundamentos matemáticos, algorítmicos y prácticos para implementarlas correctamente. Esta teoría se apoya en conceptos de otras disciplinas, como las gramáticas formales, los autómatas, las estructuras de datos y la teoría de lenguajes formales, para modelar y resolver problemas relacionados con la traducción automática de programas.

Además de su aplicación directa en la construcción de compiladores, esta teoría permite entender cómo se implementan herramientas como intérpretes, analizadores estáticos, asistentes de programación y lenguajes intermedios. Nos ayuda también a reflexionar sobre la eficiencia, corrección y portabilidad del software que escribimos. En pocas palabras, la teoría de compiladores es un componente esencial en las ciencias de la computación, pues actúa como puente entre el diseño de lenguajes de programación y su ejecución efectiva en los sistemas reales.

En esta nota de clase, exploraremos las ideas centrales de esta teoría, desde sus orígenes históricos hasta sus aplicaciones contemporáneas, con el fin de entender qué principios la sustentan, cómo se ha desarrollado a lo largo del tiempo y cuál es su papel en el futuro del desarrollo de software.

1. Orígenes

Década de 1930

- ⋆ **1936.** Alan Turing presenta la máquina de Turing, sentando las bases teóricas de la computación y proporcionando un modelo abstracto para entender la capacidad de los lenguajes formales.
- ⋆ **1936.** Alonzo Church introduce el cálculo λ, otro modelo fundamental para describir computación, que más tarde influiría en la semántica de lenguajes y en el diseño de compiladores para lenguajes funcionales.

Década de 1950

- ⋆ **1952.** Se desarrolla el primer compilador para el lenguaje AUTOCODE en la Universidad de Manchester, marcando el inicio de la traducción automática de lenguajes de alto nivel.
- ⋆ **1954.** John Backus lidera el desarrollo de FORTRAN y posteriormente de su compilador, uno de los primeros en incorporar optimizaciones automáticas.
- ⋆ **1959.** Nace el lenguaje ALGOL, cuyo desarrollo motivó el uso de gramáticas formales para describir su sintaxis.

Década de 1960

- ⋆ **1960.** Se introduce la notación BNF (Backus-Naur Form) para describir la sintaxis de lenguajes de programación, un estándar que permanece vigente.
- ⋆ **1961.** Robert W. Floyd y Donald Knuth inician el estudio de los análisis sintácticos, sentando las bases para los algoritmos descendentes y ascendentes.
- ⋆ **1965.** Peter Wegner propone formalmente las fases del proceso de compilación: análisis léxico, sintáctico, semántico y generación de código.
- ⋆ **1969.** Stephen Johnson desarrolla `yacc` (Yet Another Compiler Compiler), una herramienta para generar analizadores sintácticos automáticamente.

Década de 1970

- ⋆ **1970.** Se desarrollan los autómatas LR por Donald Knuth, que permiten el análisis sintáctico eficiente de lenguajes no deterministas.
- ⋆ **1977.** Alfred Aho y Jeffrey Ullman publican *Principles of Compiler Design*, un texto fundacional que sistematiza la teoría y práctica de los compiladores.

Década de 1980

- ⋆ **1986.** Se lanza el compilador de MODULA-2, uno de los primeros compiladores en aplicar de forma sistemática optimizaciones globales de código.
- ⋆ **1988.** Aparece el compilador de GCC (GNU Compiler Collection), un proyecto de código abierto que permite explorar técnicas de compilación en arquitecturas reales.

Década de 1990

- ⋆ **1995.** Surge el concepto de compilación JIT (Just-In-Time) con el lanzamiento de la JVM de Java, combinando compilación y ejecución dinámica.
- ⋆ **1997.** Scott Muchnick publica *Advanced Compiler Design and Implementation*, consolidando décadas de técnicas de optimización y diseño de compiladores industriales.

Década de 2000

- ⋆ **2003.** Chris Lattner desarrolla el proyecto LLVM, una infraestructura modular de compiladores que revolucionará la forma en que se construyen backends modernos.
- ⋆ **2006.** Aparece la segunda edición de *Compilers: Principles, Techniques, and Tools* (el "Dragón Azul"), actualizando la teoría clásica con nuevos enfoques y herramientas.

Década de 2010

- ⋆ **2010.** Se popularizan lenguajes como Rust y Go, que introducen nuevos desafíos para los compiladores debido a sus sistemas de tipos avanzados y requerimientos de seguridad.
- ⋆ **2015.** Empiezan a surgir herramientas asistidas por IA para ayudar en la optimización y análisis de código fuente.

Década de 2020

- ⋆ **2020.** Surgen compiladores que integran modelos de lenguaje para tareas como generación de código, refactorización y detección de errores.
- ⋆ **2023.** Se publican trabajos sobre *neuro-symbolic compilation* y uso de LLMs para asistir en la creación automática de compiladores y DSLs.

2. ¿Por qué estudiar Compiladores?

Es fundamental que quienes se dedican a la ciencia de la computación comprendan la teoría de compiladores por varias razones importantes:

- ⋆ Proporciona una comprensión profunda del proceso mediante el cual los lenguajes de programación son interpretados o traducidos a instrucciones ejecutables por una máquina. Este conocimiento permite escribir código más eficiente y predecible, con un entendimiento claro de lo que ocurre "tras bambalinas".
- ⋆ La teoría de compiladores integra conceptos fundamentales como gramáticas formales, autómatas, estructuras de datos y algoritmos, desarrollando habilidades analíticas clave para cualquier perfil enfocado en la computación.
- ⋆ Permite diseñar e implementar compiladores, intérpretes y traductores de lenguajes, lo cual es esencial para construir nuevos lenguajes de programación, DSLs (lenguajes de dominio específico) o herramientas que interactúan directamente con el código fuente.
- ⋆ Estudiar compiladores ayuda a optimizar el rendimiento del software, ya que permite aplicar técnicas de análisis estático, optimización de código y administración eficiente de memoria en tiempo de compilación.
- ⋆ Contribuye al desarrollo de herramientas modernas como asistentes inteligentes de programación, editores con resaltado semántico, analizadores de errores en tiempo real y plataformas de verificación formal, todas basadas en principios de compilación.
- ⋆ Una sólida formación en esta teoría brinda una ventaja competitiva en campos avanzados como la criptografía y seguridad, verificación de software, diseño de lenguajes, desarrollo de sistemas embebidos y compilación para arquitecturas heterogéneas (GPU, dispositivos móviles, etc.).

3. Conclusión

A lo largo de esta nota hemos explorado los orígenes, fundamentos y razones por las cuales la teoría de compiladores ocupa un lugar central en la formación de cualquier persona dedicada a la ciencia de la computación. Desde las bases formales establecidas por Turing y Church hasta los desarrollos contemporáneos en infraestructuras como LLVM y la integración de modelos de lenguaje, la evolución de los compiladores refleja el avance mismo de la disciplina. Comprender cómo se traduce un programa, cómo se analiza su estructura, cómo se optimiza su ejecución y cómo se construyen las herramientas que usamos día a día, no solo es un ejercicio técnico, sino una puerta de entrada al pensamiento profundo sobre cómo funciona el software que mueve al mundo. Estudiar compiladores es, en última instancia, aprender a pensar con rigor, eficiencia y visión sobre el corazón mismo de la programación.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compiladores: Principios, Técnicas y Herramientas*, 2.ª ed. Madrid: Pearson Educación, 2008, Conocido como el "Libro del dragón azul".

[2] D. Grune, H. E. Bal, C. J. H. Jacobs y K. G. Langendoen, *Modern Compiler Design*, 2.ª ed. London: Springer, 2012, Incluye teoría y práctica de compiladores, con temas modernos.

[3] K. D. Cooper y L. Torczon, *Engineering a Compiler*, 2.ª ed. Boston: Morgan Kaufmann, 2011, Énfasis en diseño práctico y optimización.

[4] C. Lattner y V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, págs. 75-86, 2004, Base del backend moderno de compiladores.

[5] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. Brown University, 2007, Explora la conexión entre semántica, interpretación y compilación.