Compiladores
Unidad 1: Introducción
Otros componentes auxiliares

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota abordaremos de manera introductoria algunos de los módulos externos que complementan el funcionamiento de un compilador: el preprocesador, el ligador y el soporte en ejecución. Aunque a menudo se piensa en el compilador como el único responsable de transformar el código fuente en un programa ejecutable, la realidade es que este proceso depende de la colaboración de distintas herramientas que amplían ensamblan y sostienen la ejecución final del software.

El objetivo es ofrecer una primera noción de qué papel juega cada uno de estos módulos en la cadena de traducción. El preprocesador actúa como un filtro inicial que prepara el código antes de la compilación; el ligador se encarga de unir piezas de código y bibliotecas para formar un programa completo; y el soporte en ejecución proporciona los procesos necesarios para que el código compilado pueda ejecutarse de manera coherente sobre la máquina. Conocer estas etapas auxiliares permite comprender mejor el entorno en el que opera un compilador y apreciar la complejidad real de transformar un programa desde su forma textual hasta su ejecución final.

1. Tabla de símbolos

En el estudio de los compiladores, uno de los conceptos más fundamentales y a la vez más reveladores para comprender la "inteligencia" de estas herramientas es la tabla de símbolos. Aunque a primera vista puede sonar como un simple listadao de nombres y valores, en realidad constituye el corazón administrativo del compilador: es la estructura encargada de registrar, mantener y proveer información sobre las entidades declaradas en un programa. Dicho de otro modo, si el compilador fuese una institución académica, la tabla de símbolos sería su archivo central, donde se almacenan los expedientes de cada estudiante, docente y curso, con los detalles necesarios para operar de manera consistente y evitar contradicciones.

Desde una perspectiva conceptual, la tabla de símbolos surge como una abstracción de memoria organizada. Cuand una persona programadora declara una variable, una función, una clase o incluso un parámetro, el compilador debe responder a múltiples preguntas:

- ⋆ ¿ya se había declarado antes?
- ⋆ ¿qué tipo tiene?
- ⋆ ¿dónde vive en la memoria?
- ⋆ ¿es accesible desde este punto del programa?
- ⋆ ¿qué reglas semánticas se le aplican?

En términos de implementación, una tabla de símbolos suele diseñarse como una estructura de datos jerárquica (generalmente basada en diccionarios, árboles objetivo tablas *hash*) que permite manejar el alcance (*scope*). Así, no es lo mismo una variable declarada en el cuerpo de una función que una declarada de manera global. El compilador necesita distinguir estas situaciones para resolver adecuadamente las referencias y evitar errores semánticos. Este manejo del contexto es esencial para asegurar que los programas no sólo sean sintácticamente correctos, sino también coherentes en su interpretación.

La relevancia de la tabla de símbolos se entiende mejor al considerar la cadena completa de un compilador. Durante el análisis léxico y síntáctico, los constructores se identifican y organizan; pero es el análisis semántico donde la tabla de símbolos brilla, pues es allí donde se coteja la corrección de lo que el código describe. Posteriormente, en fases como la generación de código intermedio u optimización, la información almacenada en la tabla (por ejemplo, direcciones de memoria, alineación de datos o firmas de funciones) resulta indispensable.

Más allá de los técnico, el concepto de tabla de símbolos representa una idea central de la teoría de compiladores: la necesidad de formalizar y sistematizar la administración del conocimiento dentro de un lenguaje de programación. Cada identificador en el código es, en esencia, una referencia a un concepto que el compilador debe entender y ubicar de manera precisa. La tabla de símbolos es, pues, la materialización de esa comprensión: el puente entre la abstracción del lenguaje humano y la rigurosidad de la máquina.

Una tabla de símbolos no es solamente un repositorio de datos, sino un mecanismo organizador y garante de coherencia semántica en los lenguajes de programación. Comprender su función otorga a las personas estudiantes de compiladores una primera visión clara de cómo los lenguajes se transforman en sistemas vivos y verificables, y abre la puerta al estudio de estructuras más sofisticadas que permiten a la computación mantener su presición en medio de la complejidad.

Ejemplo 1

Imaginemos el siguiente fragmento de código en un lenguaje imperativo sencillo:

```
int x = 5;

void foo(int y) {
    int z = x + y;
    print(z);
}
```

Al procesar este código, el compilador construye una tabla de símbolos que podría representarse de manera simplificada así:

Nombre	Categoría	Tipo	Alcance
`x`	Variable	`int`	Global
`foo`	Función	`int` $\rightarrow$ `void`	Global
`y`	Parámetro	`int`	Local a `foo`
`z`	Variable	`int`	Local a `foo`

La tabla de símbolos permite al compilador responder, en este caso, a preguntas como:

- ⋆ Cuando dentro de `foo` aparece `y`, ¿qué tipo tiene y cuál es su alcance?
- ⋆ Al evaluar `x + y`, ¿cómo distinguir que `x` no es una variable local sino una global ya declarada?
- ⋆ ¿Qué parámetros espera la función `foo` y cómo se verifican las llamadas a ella?

Este ejemplo muestra cómo la tabla de símbolos no sólo es un "catálogo de identificadores", sino

una estructura activa que permite **resolver referencias, verificar tipos y controlar el alcance**, garantizando que el programa sea semánticamente correcto antes de traducirse a código ejecutable.

2. Gestión de errores

Dentro del vasto universo de la construcción de compiladores, uno de los aspectos que con frecuencia se pasa por alto en los primeros acercamientos, pero que resulta esencial para cualquier sistema robusto, es la gestión de errores. En términos generales, este concepto se refiere al conjunto de técnicas, estrategias y mecanismos que un compilador emplea para detectar, informar y, en la medida de lo posible, recuperse de los errores presentes en un programa fuente. Si bien puede sonar como un detalle meramente técnico, en realidad la gestión de errores constituye una de las características más visibles de un compilador para sus personas usuarias: un buen sistema de diagnóstico puede marcar la diferencia entre una experiencia frustrante y un proceso de desarrollo productivo.

El papel de la gestión de errores

Desde la perspectiva teoríca, un compilador es un traductor formal: recibe una cadena en un lenguaje fuente y debe determinar si esta cadena pertenece al lenguaje (según las reglas gramaticales y semánticas) y, en caso afirmativo, producir una representación equivalente en otro nivel (código intermedio, ensamblador, etc.). Sin embargo, en la práctica, los programas escritos por seres humanos rara vez son perfectos en su primer intento. De aquí surge la necesidad de que el compilador no se limite a rechazar la entrada ante el primer error, sino que sea capaz de proporcionar información clara y útil para guiar a la persona programadora hacia la corrección.

La gestión de errores involucra, entonces, tres funciones principales:

1. **Detección** Identificar en qué punto del programa ocurre una violación a las reglas sintácticas o semánticas.

2. **Diagnóstico** Describir con precisón la naturaleza del error, su ubicación y, en lo posible, su causa.

3. **Recuperación** Permitir que el compilador continúe procesando el resto del programa, evitando que un único error sature la salida con mensajes confusos o, peor aún, detenga por completo el análisis.

Clasificación y técnicas

Los errores en un programa fuente pueden clasificarse de múltiples maneras: léxicos (uso indebido de caracteres o componentes léxicos), sintácticos (violaciones a la gramática del lenguaje), semánticos (inconsistencias de tipos, usos indebidos de identificadores, etc.) o incluso lógicos, aunque estos últimos superan el alcance directo del compilador.

En términos de técnicas de recuperación, la teoría de compiladores ha propuesto distintos enfoques:

- **Recuperación inmediata** Intenta corregir o ignorar el error en el punto detectado para seguir con el análisis.

- **Recuperación mediante sincronización** Consiste en saltar a un punto "seguro" del programa (como un delimitador de bloque) para retomar el análisis.

- ⋆ **Corrección de cadenas** El compilador propone cambios mínimos en la entrada con el fin de obtener una cadena válida.

Cada técnica implica un compromiso entre precisión diagnóstica, facilidad de implementación y experiencia del usuario.

La importancia de la experiencia de las personas programadoras

Desde un punto de vista más humano, la gestión de errores refleja la interacción entre el lenguaje de programación y quienes lo utilizan. Un compilador que únicamente responde con mensajes crípticos o interrumpe la compilación al menor error cumple con la teoría formal, pero fracase en su propósito pedagógico y práctico. En cambio, un compilador que informa con claridad, ofrece pistas sobre la posible solución e incluso sugiere correcciones mínimas se convierte en una herramienta formativa y accesible.

Ejemplo 2

Consideremos el siguiente código en C:

```
int main() {
    int x = 5
    printf("%d\n", x);
}
```

El error aquí es evidente: falta un punto y coma al final de la asignación de x. Un compilador con buena gestión de errores podría mostrar un mensaje como:

```
error: expected ';' before 'printf'
 3 |     printf("%d\n", x);
   |     ^~~~~~
```

Obsérvese que el compilador no sólo señala el lugar aproximado donde detectó la anomalía, sino que también sugiere cuál podría ser la causa ("se esperaba un ';'"). Además, intenta continuar procesando el resto del programa en lugar de detenerse abruptamente.

La gestión de errores no es un aspecto secundario de los compiladores, sino un componente esencial que vincula la riguorosidad formal de la teoría con la usabilidad práctica del software. Desde la detección hasta la recuperación, pasando por la claridad en el diagnóstico, esta área constituye un puente entre la disciplina matemática de los lenguajes formales y la experiencia cotidiana del desarrollo de software. Comprenderla en un nivel introductorio otorga a las personas estudiantes de compiladores una visión clara del compromiso entre teoría y práctica, y muestra como incluso un "fracaso" (un error en el código fuente) puede transformarse en una oportunidad de aprendizaje gracias a un compilador bien diseñado.

3. Módulos externos

En el imaginario común, un compilador se concibe como un artefacto cerrado: un programa que recibe código fuente y produce código ejecutable. Sin embargo, en la práctica, el proceso de traducción y ejecución de un programa rara vez es responsabilidad exclusiva del compilador en sentido estricto. A su al rededor existen otros módulos, también de gran relevancia, que intervienen en etapas previas o posteriores a la compilación y que complementan su funcionamiento. Entre estos módulos externos, destacan el preprocesador, el ligador (*linker*) y el soporte en ejecución (*runtime support*). Conocerlos permite tener una visión más

clara del ecosistema que rodea al compilador y de cómo se articula el camino que recorre un programa desde su concepción hasta su ejecución.

El preprocesador: el filtro inicial

El preprocesador es un módulo que actúa antes de la compilación propiamente dicha. Su función consiste en preparar el código fuente mediante transformaciones textuales o simbólicas que amplían la expresividad del lenguaje y facilitan la labor de las personas programadoras. Un ejemplo clásico es el preprocesador de C, encargado de interpretar directivas como `#include`, que inserta el contenido de archivos externos, o `#define`, que permite definir macros de sustitución.

En términos teóricos, el preprocesador no altera la semántica del lenguaje núcleo, pero sí modifica la superficie del programa que finalmente recibirá el compilador. Así, el código que se analiza no siempre corresponde de manera directa al que escribió la persona programadora, lo cual explica por qué comprender esta fase resulta indispensable para depurar programas y entender sus dependencias.

El ligador: ensamblando piezas

El ligador o *linker* entra en acción una vez que el compilador ha generado código objeto a partir de los módulos fuente. Su misión es unir esos fragmentos de código (posiblemente provenientes de distintos archivos y bibliotecas) en un sólo programa ejecutable. Durante este proceso, el ligador resuelve referencias externas, es decir, identifica qué símbolos definidos en un archivo corresponden a símbolos utilizados en otro.

En un sentido más profundo, el ligador encarna la idea de composición de programas: permite que el software se construya a partir de piezas modulares y reutilizables, garantizando que funciones y variables puedan invocarse más allá de los límites del archivo en el que fueron originalmente declaradas. De no existir este módulo, cada programa debería compilarse como una unidad monolítica, lo que anularía buena parte de la ingeniería de software moderna.

Soporte en ejecución: la base invisible

Finalmente, el soporte en ejecución o *runtime support* es el conjunto de procesos y estructuras que permiten que un programa compilado pueda efectivamente ejecutarse en la máquina destino. Aunque en ocasiones se percibe como un "detalle de implementación", el soporte en ejecución es decisivo: se encarga de gestionar aspectos como la inicialización de variables globales, la creación y destricción de registros de activación en la pila, el manejo de excepciones, la admninistración de memoria dinámica o incluso la interacción con el sistema operativo.

Desde la perspectiva teórica, el soporte en ejecución representa el puente entre el modelo abstracto de ejecución que define un lenguaje (funciones, llamadas, excepciones, objetos, etc.) y los recursos concretos de la máquina subyacente. Sin este módulo, las construcciones de alto nivel carecerían de correlato práctico y el programa, aunque compilado, no podría ejecutarse de manera coherente.

El estudio de los compiladores no puede limitarse a la traducción de código fuente a objeto; requiere también atender a losm módulos que lo rodean y lo hacen posible. El preprocesador amplía la expresividad inicial del código, el ligador asegura la modularidad y la integración de componentes, y el soporte en ejecución garantiza que las abstracciones del lenguaje encuentren soporte en la realidad de la máquina. Juntos, estos módulos conforman un andamiaje impresicidible que, aunque muchas veces invisible para la persona programadora, resulta vital para que el ciclo completo (desde la escritura del código hasta su ejecución) se lleve a cabo de manera exitosa.

4. Conclusión

Comprender los módulos externos a un compilador permite tener una visión más amplia y realista del proceso de construcción y ejecución de programas. El preprocesador, el ligador y el soporte en ejecución no son meros complementos, sino eslabones fundamentales en la cadena que va del código fuente al programa operativo. REconocer su papel ayuda a apreciar que el trabajo del compilador no ocurre en aislamiento, sino dentro de un ecosistema de herramientas que, en conjunto, hacen posible la interacción entre las abstracciones de los lenguajes de programación y la maquinaria concreta de los sístemas computacionales.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Boston, MA: Pearson, 2007.

[2] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge, UK: Cambridge University Press, 1998.

[3] K. D. Cooper y L. Torczon, *Engineering a Compiler*, 2nd. San Francisco, CA: Morgan Kaufmann, 2011.