Compiladores
Unidad 1: Introducción
Variantes de traducción

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota vamos a revisar las distintas formas en que un programa puede ser llevado a la máquina. Primero revisaremos qué hace un compilador entendiendo que su arquitectura se divide en dos grandes etapas: análisis y síntesis. Después verémos qué hace un intérprete, qué comparte parte del análisis pero delega la ejecución a un motor que aplica reglas de semántica dinámica. Finalmente, hablaremos de un tercer enfoque que surge de combinar ambos mundos: las arquitecturas híbridas, que nos permiten comprender por qué lenguajes como Java pueden ser portables y eficientes al mismo tiempo.

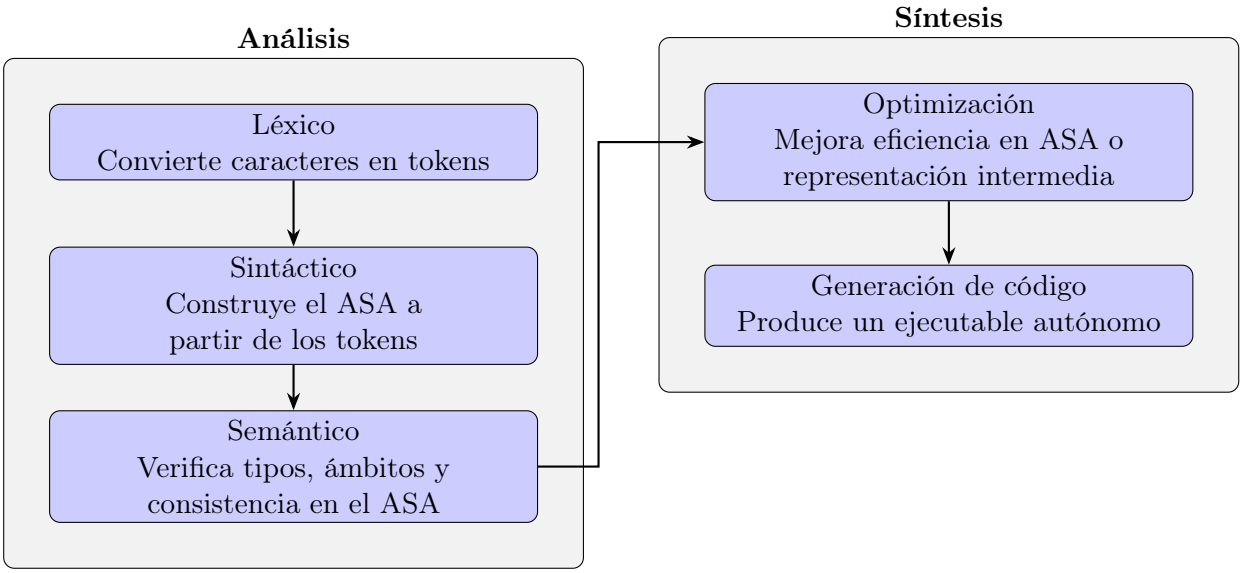
El estudio de los traductores de lenguajes de programación muestra que existen distintos caminos para transformar un programa fuente en comportamiento observable por la máquina. Mientras que los compiladores realizan un proceso completo de análisis y síntesis que culmina en un programa ejecutable autónomo, los intérpretes llevan a cabo únicamente parte del análisis y luego, mediante un motor de evaluación, aplicar directamente reglas de semántica dinámica operacional sobre el ASA. Por su parte, las arquitecturas híbridas combinan ambos enfoques, incorporando tanto traducción a un código intermedio como interpretación y compilación dinámica (*Just-In-Time*), lo que ilustra la evolución contemporánea de estas tecnologías.

1. Intérprete vs. Compilador

En los lenguajes de programación, compiladores e intérpretes son dos arquitecturas esenciales para llevar programas de alto nivel a la máquina. Ambos parten del mismo insumo (el código fuente), pero difieren en su alcance y en la manera en que utilizan la semántica estática y la semántica dinámica. La herramienta común entre ambos es el árbol de sintaxis abstracta (ASA), estructura intermedia que representa el programa de forma jerárquica.

El compilador: análisis y síntesis guiados por la semántica estática

Un compilador, como vimos anteriormente, se organiza en dos grandes etapas: análisis y síntesis.



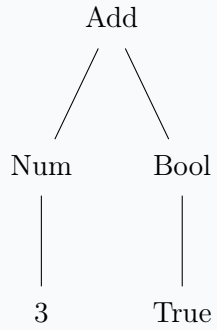
De este modo, el compilador no sólo comprende el programa (análisis), sino que también construye otro programa equivalente (síntesis).

Ejemplo 1

Código fuente:

```
3 + true
```

ASA:



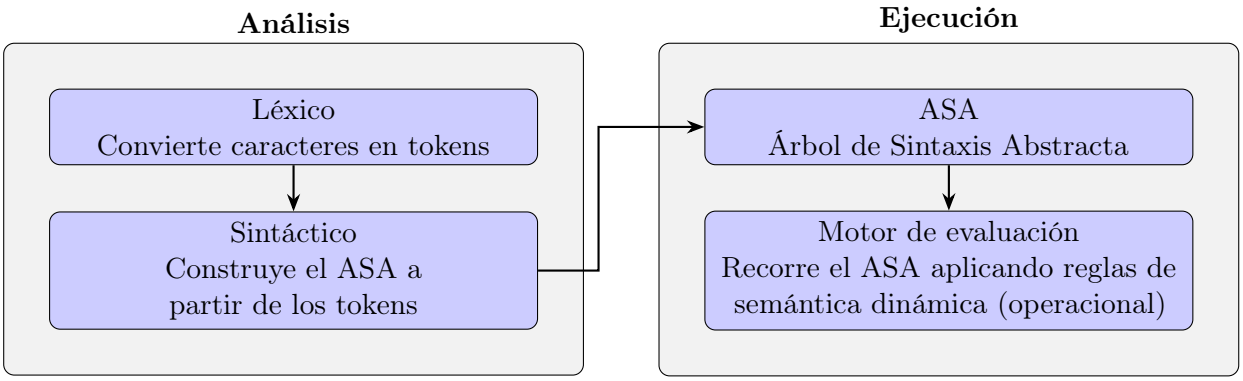
Regla de tipificado para la suma:

$$\frac{\Gamma \vdash i : \mathsf{Int} \quad \Gamma \vdash d : \mathsf{Int}}{\Gamma \vdash \mathit{Add}(i, d) : \mathsf{Int}}$$

El compilador intenta aplicar la regla sobre el ASA. La izquierda cumple $3 : \mathsf{Int}$, pero la derecha falla ($\mathit{true} : \mathsf{Bool}$). El programa se rechaza en la fase de análisis semántico, sin pasar a la síntesis.

El intérprete: análisis parcial y motor de evaluación con semántica dinámica

El intérprete comparte con el compilador las fases iniciales de análisis, pero se detiene antes de la síntesis:



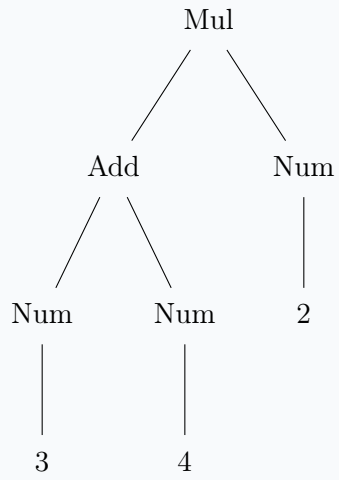
El intérprete no genera un programa autónomo; en cambio, va evaluando directamente el ASA.

Ejemplo 2

Código fuente:

```
(3 + 4) * 2
```

ASA:



Regla de evaluación usando semántica natural:

$$\frac{i \Rightarrow Num(n) \quad d \Rightarrow Num(n)}{Mul(i,d) \Rightarrow Num(n \times m)}$$

Reduciendo la expresión:

$$3 + 4 \Rightarrow Num(7) \quad 7 \times 2 \Rightarrow 14$$

$$(3 + 4) * 2 \Rightarrow Num(14)$$

Aquí el motor de evaluación aplica directamente las reglas dinámicas sobre el ASA, definiendo el comportamiento del programa en tiempo de ejecución.

La diferencia esencial radica en el alcance:

- ⋆ El compilador realiza tanto análisis como síntesis. El ASA es el insumo sobre el que se aplica la semántica estática, y a partir de ahí genera un programa ejecutable independiente.
- ⋆ El intérprete hace sólo la parte inicial del análisis (hasta construir el ASA), y luego su motor de evaluación aplica la semántica dinámica operacional, reduciendo el árbol hasta obtener un valor.

En ambos casos, el ASA es el puente entre el programa escrito y su significado formal: en compiladores garantiza corrección estática y traducción; mientras en intérpretes, guía la ejecución dinámica paso a paso o en evaluaciones completas.

2. Arquitecturas híbridas

En el desarrollo de lenguajes de programación modernos, la clásica distinción entre compiladores e intérpretes ha dado paso a un tercer enfoque: las arquitecturas híbridas. Estos sistemas combinan lo mejor de ambos mundos (la eficiencia de la compilación y la flexibilidad de la interpretación) para ofrecer un equilibrio adecuado entre rendimiento, portabilidad y facilidad de desarrollo. Quizá el ejemplo más conocido de este paradigma sea el ecosistema de Java cuyo modelo de jecución se ha convertido en un referente.

Compilación a un código intermedio

La primera etapa de una arquitectura híbrida sigue el camino de la compilación tradicional. El programa fuente, escrito en un lenguaje de alto nivel (como Java), se somete a análisis léxico, sintáctico y semántico. De este proceso resulta un código intermedio: en Java, los famosos bytecodes. Este código no es directamente ejecutable por el hardware, pero sí está diseñado para ser eficiente de analizar y portable entre diferentes plataformas.

El *bytecode* constituye una representación abstracta del programa, cercada a aun ASA aplanado y optimizado, que contiene la información necesaria para mantener portabilidad sin perder precisión semántica.

Interpretación sobre una máquina virtual

En la segunda etapa entra en juego la máquina virtual (como la Java Virtual Machine JVM). Esta actúa como un intérprete de *bytecodes*: lee las instrucciones intermedias y las ejecuta sobre la marcha. En este punto, el modelo recuerda a la semántica dinámica operacional, pues la máquina virtual aplica reglas de evaluación al código intermedio, asegurando que el programa tenga el mismo comportamiento en cualquier plataforma donde exista una JVM.

El motor de interpretación inicial permite arrancar rápidamente un programa, sin necesidad de esperar a que todo el código se compile a lenguaje máquina.

Compilación *Just-In-Time* (JIT)

La vertadera potencia de la arquitectura híbrida surge cuando se integra la compilación Just-In-Time (JIT). A medida que el programa corre, la máquina virtual detecta los fragmentos de código más utilizados (*hot spot*). En lugar de seguir interpretándolos, los compila dinámicamente a código nativo, específico del hardware en el que se ejecuta.

Este mecanismo ofrece lo mejor de dos mundos:

- ⋆ Portabilidad y flexibilidad gracias al *bytecode* y la interpretación inicial.
- ⋆ Eficiencia gracias a la traducción selectiva y adaptativa a código máquina.

Además, el compilador JIT puede aplicar optimizaciones imposibles en una compilación estática tradicional, como el *inlining* de métodos basados en estadísticas reales de ejecución.

Ventajas del modelo híbrido

Las arquitecturas híbridas permiten:

1. Un mismo programa puede ejecutarse en cualquier plataforma con una máquina virtual (portabilidad).
2. Gracias a la compilación JIT, el sistema ajusta dinámicamente la eficiencia según el uso del programa (rendimiento adaptativo).
3. La interpretación inicial facilita inspección, perfiles y recolección de información de ejecución (flexibilidad para la depuración y análisis).

Las arquitecturas híbridas como la de Java representan un puente entre la compilación y la interpretación. Al combinar la generación de un código intermedio protátil con la interpretación inicial y la compilación JIT, logran resolver la tensión histórica entre rendimiento y portabilidad. Para las personas estudiantes de teoría de compiladores, este modelo ejemplifica cómo la práctica contemporánea ha dejado atrás divisiones rígidas, adoptando soluciones que aprovechan simultáneamente la solidez de la compilación y la flexibilidad de la interpretación.

3. Conclusión

A lo largo de esta nota contrastamos las arquitecturas clásicas de compiladores e intérpretes, así como la forma en que las arquitecturas híbridas integran elementos de ambos enfoques. El compilador, con sus etapas de análisis y síntesis, ejemplifica el poder de la semántica estática para garantizar corrección y generar un programa autónomo. El intérprete, en cambio, muestra la importancia de la semántica dinámica operacional, al ejecutar directamente el ASA y revelar el comportamiento del programa paso a paso. Finalmente, las arquitecturas híbridas como la de Java nos recuerdan que la práctica moderna ha dejado atrás dicotomías rígidas, adoptando soluciones intermedias que privilegian la portabilidad, la eficiencia y la adaptabilidad en tiempo de ejecución.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd. Pearson/Addison Wesley, 2006.
[2] S. Krishnamurthi, *Programming Languages: Application and Interpretation*, 2nd. Brown University, 2007.
[3] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.