# Compiladores
# Unidad 1: Introducción
# Etapa de análisis

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota estudiaremos la fase de análisis de los compiladores, primera etapa del proceso que transforma un programa fuente en un programa objeto. Nos interesa destacar cómo esta fase articula la teoría de lenguajes formales con la práctica de la programación, sirviendo como puente entre la intención humana y la precisión computacional. A lo largo de la nota revisaremos sus subfases, ejemplos concretos y su importancia conceptual en la actualidad.

## 1. Introducción

Un compilador es un programa cuyo objetivo fundamental es traducir un programa fuente, escrito en un lenguaje de alto nivel, en un programa objeto, típicamente en lenguaje ensamblador o código máquina, que pueda ser ejecutado por una computadora. Esta tarea, lejos de ser un simple reemplazo de palabras, implica un proceso sistemático y estructurado en el que el compilador debe comprender el significado del programa original y producir una versión equivalente que sea ejecutable y eficiente.

Desde sus primeras concepciones, los compiladores se han diseñado con una organización modular, dividiéndose en dos grandes fases: análisis y síntesis. Esta división ha demostrado ser no sólo práctica para la implementación, sino también conceptual para la enseñanza y el diseño de nuevos lenguajes y arquitecturas.

### Análisis: Comprender el programa fuente

La primera gran fase del compilador es el análisis, cuyo propósito es entender qué significa el programa escrito por la persona usuaria. Para lograrlo el compilador debe leer el programa fuente y transformarlo en una representación estructurada que sea más fácil de manipular.

En términos generales, el análisis de encarga de procesar el código fuente para verificar su corrección y establecer su estructura lógica. Este proceso culmina con la construcción de un árbol de sintaxis abstracta (ASA), el cuál representa el programa de manera jerárquica y abstracta, sirviendo como puente entre el código fuente y las etapas posteriores.

Podría decirse que esta fase el compilador responde a la pregunta:

*¿Qué quiso decir la persona programadora con este código?*


Figura 1: Arquitectura de un compilador

### Síntesis: Construir el programa objeto

Una vez que el programa fuente ha sido comprendido y representado en el ASA, el compilador pasa a la segunda gran fase: la síntesis. El objetivo ahora es tomar la representación intermedia y generar un programa objeto que conserve el mismo significado, pero adaptado a la máquina destino.

De manera general, la síntesis produce primero un código intermedio, que sirve como representación independiente de la arquitectura. Posteriormente, este código se transforma en código objeto, más cercano al lenguaje máquina, y se optimiza para lograr un mejor desempeño en la computadora donde se ejecutará.

En términos conceptuales, en esta fase el compilador responde a la pregunta:

*¿Cómo transmito este significado a la computadora de la forma más eficiente posible?*

### ¿Por qué dividir en análisis y síntesis?

La separación en análisis y síntesis no es arbitraria, sino que responde a un principio de modularidad y claridad. Por un lado, el análisis se concentra en entender el programa fuente, sin importar la arquitectura en la que se ejecutará. Por otro lado, la síntesis se enfoca en generar el programa objeto, sin importar el lenguaje de programación en que fue escrito.

Esta división permite construir compiladores portables y reutilizables, donde un mismo *frontend* (análisis) puede combinarse con distintos backends (síntesis). Así un compilador puede procesar un lenguaje como C y generar código para arquitecturas diferentes (x86, ARM, RISC-V) sin necesidad de rediseñar todo el sistema.

Además, esta separación facilita el trabajo de investigación y docencia, ya que permite estudiar y mejorar cada fase de manera independiente. Las técnicas de análisis se relacionan con teorías de lenguajes formales y autómatas, mientras que las técnicas de síntesis involucran conceptos de arquitectura de computadoras y optimización.

La fase de análisis constituye la primera gran etapa en el proceso de compilación. Su propósito esencial es interpretar, verificar y estructurar el programa fuente para transformarlo en una representación intermedia que sirva de base a la posterior generación de código. En esta nota nos centraremos exclusivamente en el estudio de esta fase, la cual se descompone en las siguientes subfases principales:

- Análisis léxico
- Análisis sintáctico
- Análisis semántico

## 2. Análisis léxico

El análisis léxico es el primer paso y se encarga de transformar la cadena de caracteres que forma en programa en componentes léxicos (*tokens*), es decir, las unidades mínimas con significado reconocidas por el compilador. Estos componentes léxicos incluyen palabras reservadas, identificadores, operadores, números y símbolos de puntuación.

**Ejemplo 1**

La cadena:

```
while (x < 10)
{
  x = x + 1;
}
```

se segmenta en componentes léxicos:

```
[TPalabraReservada "while",
 TSeparador "(",
 TId "x",
 TOp "<",
 TNum 10,
 TSeparador ")"
 TSeparador "{",
 TId "x",
 TOp "=",
 TId "x",
 TOp "+",
 TNum 1,
 TPyC,
 TSeparador "}"]
```

Además de identificar estos elementos, el análisis léxico elimina aspectos irrelevantes para la traducción, como comentarios y espacios en blanco. También genera información valiosa para las fases posteriores, como la ubicación de los componentes léxicos en el archivo fuente, lo cual permite al compilador emitir mensajes de error más precisos.

El proceso de análisis léxico suele implementarse mediante autómatas finitos que reconocen patrones regulares. Herramientas como Lex o Alex en Haskell generar automáticamente analizadores léxicos a partir de expresiones regulares, lo que refleja la estrecha conexión entre teórica de autómatas y práctica de compilación.

## 3. Análisis sintáctico

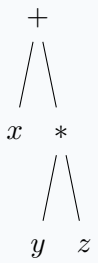
El análisis sintáctico recibe la secuencia de componentes léxicos producida por el análisis léxico y verifica que estos sigan las reglas gramaticales del lenguaje. Su producto es un árbol de sintaxis que refleja la estructura jerárquica del programa.

**Ejemplo 2**

En la expresión:

```
x + y * z
```

el analizador sintáctico no sólo valida la corrección, sino que construye un árbol donde `*` tiene mayor precedencia que `+`. El resultado es:




El árbol de sintaxis revela cómo debe interpretarse la expresión y resuelve ambigüedades propias del lenguaje.

Los analizadores sintácticos modernos se basan en algoritmos como LL (análisis descendente) y LR (análisis ascendente), que garantizan la construcción de árboles de derivación de manera eficiente. La elección de la técnica depende de la gramática del lenguaje y de las exigencias de rendimiento.

Además, el análisis sintáctico tiene una dimensión pedagógica: los errores de sintaxis detectados en esta etapa son los más comunes para las personas programadoras, y los compiladores invierten un esfuerzo considerable en recuperación de errores, de modo que un único fallo no interrumpa todo el proceso de compilación.

## 4. Análisis semántico

El análisis semántico se encarga de verificar reglas de significado estático, es decir, aquellas que pueden comprobarse en tiempo de compilación. En el contexto académico corresponde a la semántica estática

estudiada en los cursos de lenguajes de programación, y no debe confundirse con la semántica dinámica, que describe el comportamiento del programa en tiempo de ejecución.

Entre sus tareas más importantes se encuentran:

- ⋆ Verificar que las operaciones se realicen entre operandos compatibles. Por ejemplo, en la instrucción `x = true + 3;` la estructura sintáctica es válida, pero la operación carece de sentido semántico.
- ⋆ Garantizar que todas las variables y funciones hayan sido declaradas antes de ser utilizadas.
- ⋆ Validar que las apariciones de identificadores correspondan al alcance en el que están definidas.
- ⋆ Asegurar que las funciones sean llamadas con el número y tipo correcto de argumentos.

El análisis semántico produce un un tipo de árbol de sintaxis conocido como árbol de sintaxis abstracta (ASA) el cuál es enriquecido con información adicional, como tipos de datos o apuntadores a tablas de símbolos (hablaremos de esto más adelante en otra nota). Este árbol será utilizado en la fase de síntesis para la generación de código.

En términos conceptuales, mientras el análisis sintáctico responde a la pregunta "*¿está bien formado el programa?*", el análisis semántico responde a "*¿tiene sentido lo que el programa expresa?*".

## 5. El rol del análisis en la actualidad

Aunque cada subfase cumple un rol particular, es su interacción lo que otorga el poder al análisis como conjunto. El análisis léxico otorga precisión simbólica, el sintáctico organiza la estructura y el semántico asegura la coherencia en el significado. El resultado es un modelo formal del programa, preparado para ser traducido en la fase de síntesis.

Este proceso puede entenderse como un esfuerzo por reducir la brecha semántica: la distancia entre la forma en que los humanos expresamos nuestras intenciones en un lenguaje de programación y la manera en que la máquina puede procesar esas instrucciones. En la medida en que el análisis convierte texto ambiguo en estructuras formales y rigurosas, disminuye dicha breva y habilita la comunicación efectiva entre ambos mundos.

En la actualidad, esta noción cobra un valor renovado con el advenimiento de la Inteligencia Artificial Generativa. Modelos como ChatGPT, Copilot o Codex buscan interpretar directamente el lenguaje natural para producir código. En cierto sentido, estas herramientas trasladas la problemática clásica del análisis en un nuevo nivel: ya no se trata únicamente de interpretar un lenguaje formal, sino de reducir la brecha semántica entre el lenguaje natural humano y los lenguajes de programación.

Desde esta pespectiva, la fase de análisis en los compiladores puede verse como un precursor conceptual de los sistemas de IA Generativa, que también deben lidiar con la ambigüedad, la estructura y el significado para producir representaciones útiles. La diferencia más destacable es que los compiladores trabajan con reglas estrictas y bien definidas, mientras que los modelos generativos operan sobre distribuciones probabilísticas de lenguaje. Sin embargo, el objetivo de ambos sigue siendo el mismo acercar la intención humana a una representación computacional ejecutable.

## 6. Conclusión

En conclusión, la fase de análisis constituye un componente esencial en el proceso de compilación, al garantizar que el programa fuente sea interpretado, verificado y estructurado de manera rigurosa. Al dividirse en análisis léxico, sintáctico y semántico, nos permite establecer un modelo abstracto coherente del programa que servirá de insumo para las etapas posteriores. En las siguientes notas continuaremos con el estudio de la fase de síntesis y sus implicaciones.

## Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2.ª ed. Pearson, 2006.
[2] K. D. Cooper y L. Torczon, *Engineering a Compiler*, 2.ª ed. Morgan Kaufmann, 2011.
[3] M. L. Scott, *Programming Language Pragmatics*, 4.ª ed. Morgan Kaufmann, 2016.