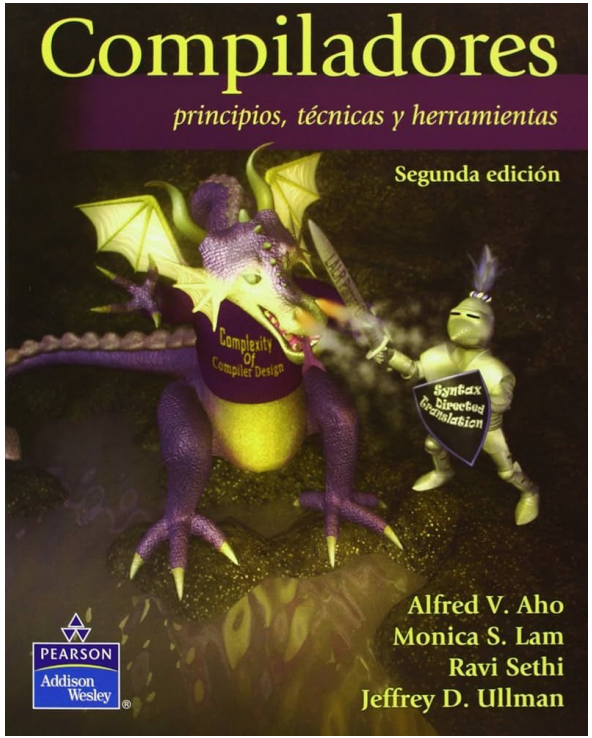
INTRODUCCIÓN A COMPILADORES

El reto es vencer al dragón

Ing. Max Cerna

“Un buen idioma es aquel que la gente usa”


Compiladores
Principios, técnicas y herramientas
MPLEXITY OF COMPILER DESIGN
POWER
ON OFF
BRIGHT
DATA FLOW ANALYSIS
LALR PARSER GENERATOR
SYNTAX DIRECTED TRANSLATION
PEARSON
Addison Wesley
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman


Compiladores
principios, técnicas y herramientas
Segunda edición
Complexity Of Compiler Design
Syntax Directed Translation
PEARSON
Addison Wesley
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Agenda

1. Introducción

2. Acerca de Compiladores

Introducción

Objetivos

- Entender que hace un compilador
- Entender cómo funciona un compilador
- Entender cómo se construye un compilador

Porque estudiar Lenguajes y Compiladores?

1. **Aumentar la capacidad de expresión**

Adquirir una mayor capacidad para expresar ideas y algoritmos de manera efectiva. Cada lenguaje tiene sus propias características y paradigmas que permiten abordar problemas desde diferentes ángulos.

2. **Aprender a construir un sistema grande y confiable**

Permite entender mejor cómo se ejecutan los programas, esto incluye el manejo de la memoria, la optimización del código y la gestión de recursos.

Porque estudiar Lenguajes y Compiladores?

3. **Mejorar la capacidad de aprender nuevos lenguajes**

Estudiar varios lenguajes y los principios de compilación te proporciona una base sólida que facilita el aprendizaje de nuevos lenguajes de programación.

4. **Mejorar la comprensión del comportamiento del programa**

Implica comprender cómo se diseñan y construyen sistemas grandes y complejos. Aprender sobre modularidad, gestión de dependencias, compilación eficiente y generación de código.

Porque estudiar Lenguajes y Compiladores?

5. **Ver muchos conceptos básicos de informática en funcionamiento**

Conceptos como estructuras de datos, algoritmos, teoría de autómatas, teoría de lenguajes formales, y más.

Lenguajes de Programación

Intérpretes que ejecutan los programas

Ejecución Inmediata

Portabilidad

Interactividad

Lenguajes de Programación

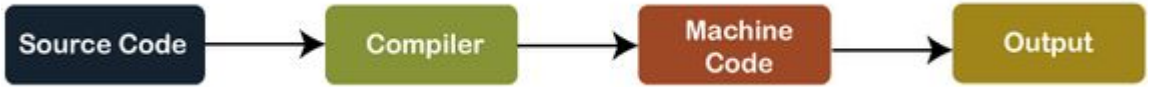
Compiladores que traducen los programas

Análisis Exhaustivo

Traducción a Código Máquina

Optimización de Código

Ejecución Eficiente



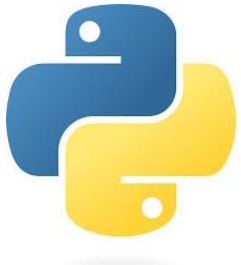
Implementaciones

Los compiladores dominan los lenguajes de bajo nivel



Implementaciones

Los intérpretes dominan los lenguajes de alto nivel



Implementaciones

Algunas implementaciones de lenguaje proporcionan ambos

Tendencia: Intérprete + compilador JIT

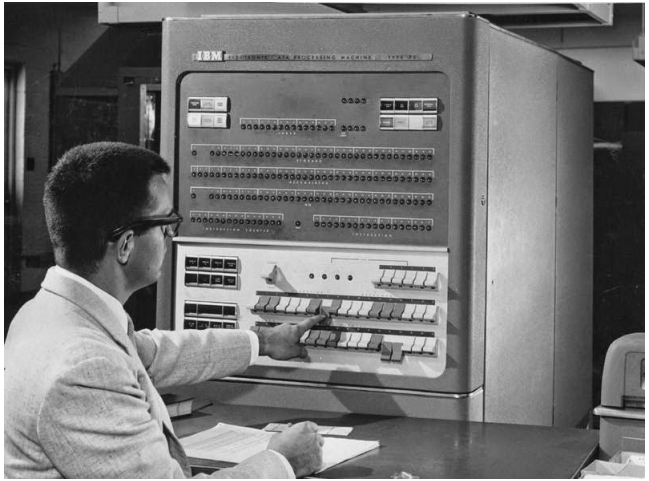
“Abstracción es ignorancia selectiva” - Andrew Koenig

Historia

1954: IBM desarrolla el 704

El costo del software excede al hardware

Toda la programación hecha en ensamblador



Historia

Speedcoding:

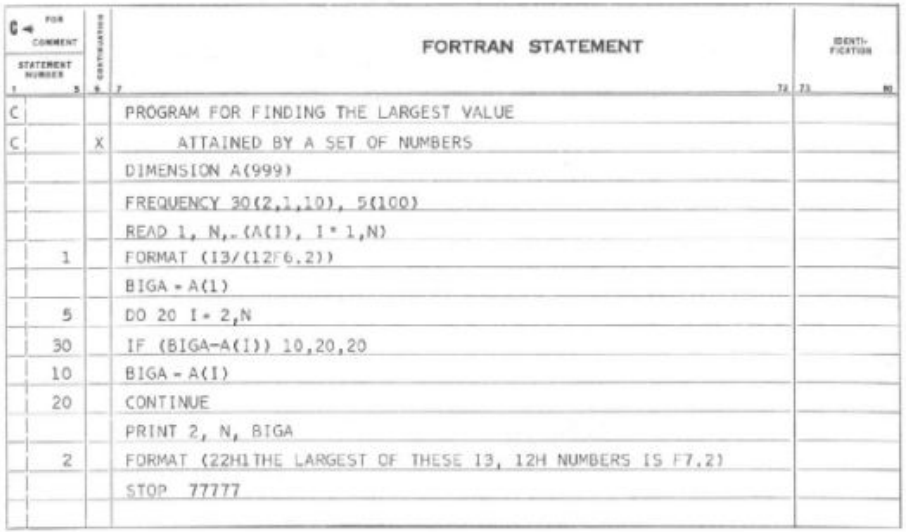
- Fue uno de los primeros lenguajes de alto nivel desarrollados para facilitar la programación en computadoras.
- Fue creado por John Backus en 1953 para el IBM 701
- Intérprete
- 10-20 veces más lento que el ensamblador escrito a mano

Fortran I (Formula Translation)

- Creado por John Backus
- Traduce código de alto nivel a ensamblador
- No fue el primer intento pero si el primero exitoso

Fortran I

- 1954-57 - El proyecto FORTRAN I
- 1958 - 50 % de software en el mundo escrito en Fortran
- El tiempo de desarrollo se redujo y el desempeño era equiparable a ensamblador



C = FOR COMMENT	STATEMENT NUMBER 1–5	CONTINUATION 6	FORTRAN STATEMENT 7–72	IDENTIFICATION 73–80
C			PROGRAM FOR FINDING THE LARGEST VALUE	
C		X	ATTAINED BY A SET OF NUMBERS	
			DIMENSION A(999)	
			FREQUENCY 30(2,1,10), 5(100)	
			READ 1, N,.(A(I), I = 1,N)	
	1		FORMAT (I3/(12F6.2))	
			BIGA = A(1)	
	5		DO 20 I = 2,N	
	30		IF (BIGA-A(I)) 10,20,20	
	10		BIGA = A(I)	
	20		CONTINUE	
			PRINT 2, N, BIGA	
	2		FORMAT (22H1THE LARGEST OF THESE I3, 12H NUMBERS IS F7.2)	
			STOP 77777	

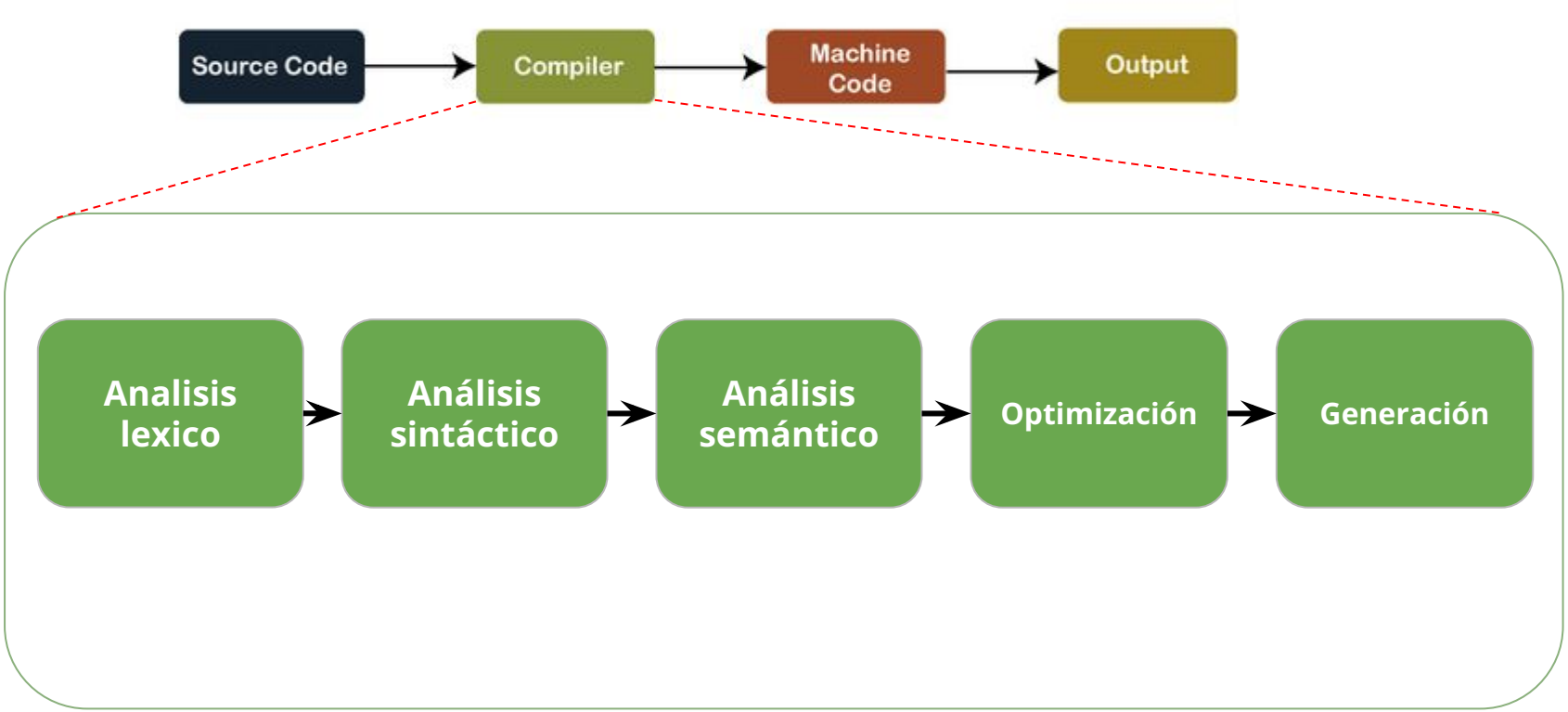
Fortran I

Ampliamente adoptado en la computación científica. Su desarrollo marcó el inicio de la programación de alto nivel y allanó el camino para otros lenguajes más modernos.

FORTRAN han influido en el diseño de muchos otros lenguajes y compiladores modernos. La eficiencia y la optimización para cálculos numéricos, características centrales de FORTRAN.

Compiladores

Estructura de un compilador



Estructura de un compilador

Analisis lexico

- Convierte el código fuente en una secuencia de palabras (tokens).
- Los tokens son la unidad más pequeña luego de las letras.
- Un token puede representar una palabra clave, un identificador, un operador, un delimitador, etc.
- Cuenta con un alfabeto.

Estructura de un compilador

Analisis lexico

Primer Paso:
Reconocer palabras

Ejemplo:

Esta es una oración.

Estructura de un compilador

Analisis lexico

Ejemplo:

taes se anun nociora

Estructura de un compilador

Analisis lexico

- Los analizadores léxicos dividen los programas en “tokens”:

if x == y then z = 1; else z =2;

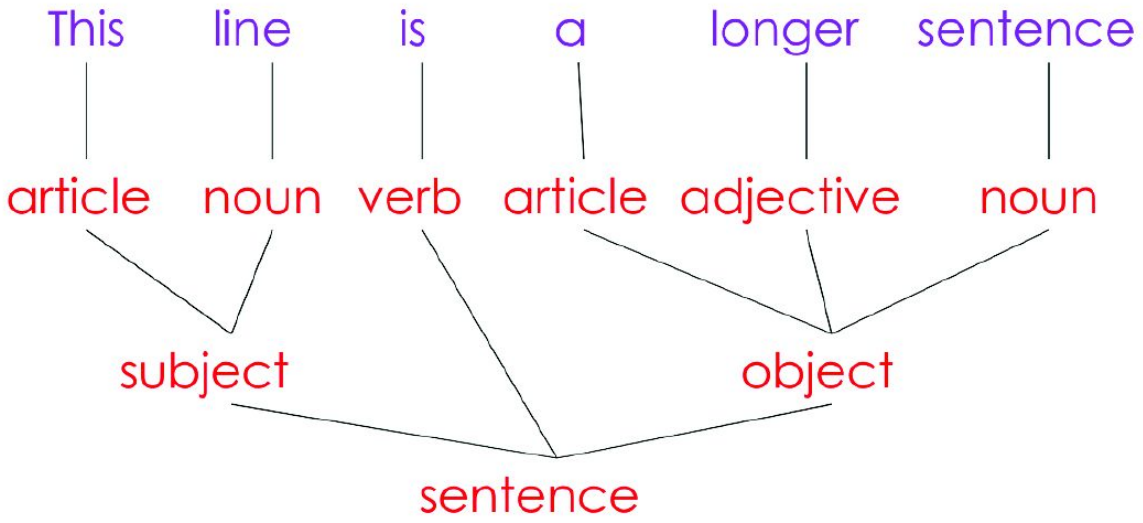
Estructura de un compilador

Análisis sintáctico

- Entender la estructura de la oración, o sea verifica que la secuencia de tokens siga las reglas de la gramática del lenguaje.
- Parsing = Brindar estructura a una oración (Generalmente un árbol)
- Detecta errores sintácticos y proporciona mensajes de error informativos para ayudar a los programadores a corregirlos.

Estructura de un compilador

Análisis sintáctico



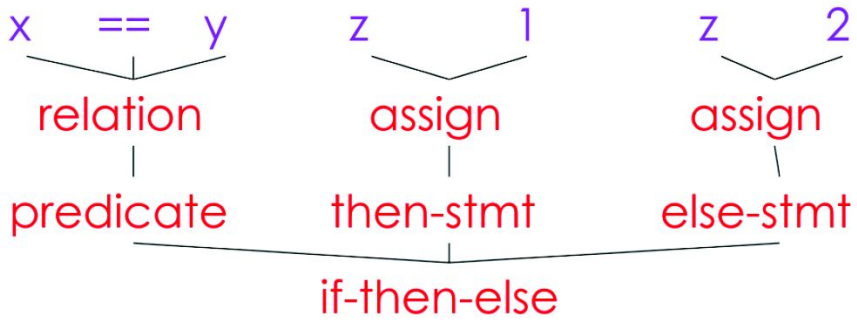
Estructura de un compilador

Análisis sintáctico

Consideremos

if x == y then z = 1; else z =2;

Diagrama:



Estructura de un compilador

Análisis semántico

- Una vez entendida la estructura de la oración necesitamos entender su significado.
- Detección de inconsistencias.
- Asegura que las variables y funciones se utilicen dentro del alcance en el que fueron definidas.
- Asegura que las operaciones se realicen entre operandos de tipos compatibles.

Estructura de un compilador

Análisis semántico

Ejemplo:

Juan dijo que José dejó su tarea en casa

¿Quien la dejó en casa?

Estructura de un compilador

Análisis semántico

Ejemplo:

Juan dijo que Juan dejó su tarea en casa

¿El multiverso de los Juan? ¿Cuantos Juan son?

Estructura de un compilador

Análisis semántico

Ejemplo:

```
{
    int juan = 3;
    {
        int juan = 4;
        cout << juan;
    }
}
```

Estructura de un compilador

Análisis semántico

Los compiladores ejecutan diversas verificaciones semánticas, por ejemplo verificación de tipos:

Pikachu es el mejor Digimon

“Type mismatch“ entre Pikachu y Digimon

Estructura de un compilador

Optimización

- Las optimizaciones pueden realizarse en varias etapas de la compilación, principalmente en el nivel intermedio (código intermedio) y el nivel del código máquina.

Estructura de un compilador

Optimización

Objetivos de Optimización

- Mejora del Rendimiento
  - Reducir el tiempo de ejecución del programa
  - Minimizar el número de instrucciones ejecutadas
- Reducción del Tamaño del Código
  - Disminuir la cantidad de memoria para almacenar el programa.
  - Compactar el código eliminando instrucciones redundantes.
- Eficiencia en el Uso de Recursos:
  - Optimizar el uso de registros y memoria.
  - Mejorar el rendimiento de la caché.

Estructura de un compilador

Optimización

Ejemplo simple:

```
X = Y *0
..
X = 0
```

Estructura de un compilador

Generación

- Al día de hoy muchos compiladores generan representaciones intermedias.
- En cada nivel se reduce la abstracción.
- El código intermedio es una representación abstracta y simplificada del programa fuente que se genera durante el proceso de compilación, sirve como una etapa intermedia que facilita la optimización y la generación de código final para diferentes arquitecturas de hardware.

Estructura de un compilador

Generación

Objetivos del código intermedio

- Independencia de la Máquina
- Facilitación de Optimización
- Simplificación de la Generación de Código (puente entre el código fuente de alto nivel y el código máquina de bajo nivel)