Compiladores
Facultad de Ciencias UNAM
Introducción

Lourdes Del Carmen González Huesca [*]

8 de febrero de 2024

Este curso está dirigido a estudiar los principios de compiladores, su diseño y entender la relación entre bajo y alto nivel. Además pretende complementar las bases del diseño de lenguajes de programación así como entender los detalles de generación de código. Se abordarán temas sobre componentes específicas de los compiladores y algunas optimizaciones. También se revisarán los aspectos prácticos al desarrollar paso a paso un compilador para un lenguaje imperativo. Estudiar los principios de compilación a través de técnicas y algoritmos ayudará a la comprensión de lenguajes de programación en general así como a mejorar el rendimiento en programas y desarrollos.

Introducción

En Ciencias de la Computación el estudio de los lenguajes de programación es importante por varios motivos, entre ellos que favorece la comprensión del diseño de los lenguajes y su implementación. También es útil cuando se desea hacer una buena selección del lenguaje a utilizar cuando se está por resolver una tarea.

Durante el curso de Lenguajes de Programación se abordan formalismos como sintaxis de lenguajes, diseño de sistema de tipos, manejo de variables, estilos de reducción, entre otras cosas. Se verá que el estudio de los compiladores incluye además el conocimiento de las herramientas que ayudan a la evaluación de un programa, es decir aquellas que permiten pasar del alto nivel al bajo nivel como lo son los *debuggers*, ligadores y ensambladores. También incluye comprender otros detalles como los análisis sintácticos, optimizaciones y representaciones de código.

La compilación o un proceso equivalente servirá para conectar la interacción humana y la ejecución final en un procesador.

Un compilador traduce un lenguaje de alto nivel (adaptado a los seres humanos) en un lenguaje de bajo nivel (diseñado para ser ejecutado "eficientemente" por una computadora). Las traducciones se realizan mediante fases bien definidas en donde cada una de ellas recaba o compila información útil para las fases siguientes. En caso de que alguna fase no acepte el código de entrada se señalarán los errores encontrados. Cada una de las transiciones debe preservar el significado del programa fuente, escrito en un lenguaje (imperativo), en un programa objeto, escrito en un lenguaje de bajo nivel, para finalmente ser ejecutado por un procesador designado.

El trabajo del compilador se divide en **analizar** (reconocer un programa y su significado y señalar los posibles errores) y **sintetizar** un código en lenguaje objeto al usar lenguajes intermedios. Como mencionamos, las traducciones deben respetar el código fuente [1] por lo tanto uno de los requerimientos

[*]Material revisado por el servicio social de Apoyo a la Docencia y Asesoría Académica en la Facultad de Ciencias de la UNAM con clave 2023-12/12-292, de Diana Laura Nicolás Pavia.

[1]El termino código fuente será utilizado frecuentemente durante el curso.

escenciales del compilador es la correctud entre fases. A continuación describimos tres requerimientos de un compilador que consideraremos en este curso:

- Correctud: verificar la correctud del código generado en cada etapa. El compilador debe rechazar programas que no están bien formados ya sea léxica, sintáctica o semánticamente. El compilador debe tener especial consideración en la semántica del lenguaje.
- Eficiencia: el código generado debe ser eficiente y así tambien lo debe ser el compilador en sí mismo. Para poder verificar la eficiencia del compilador es necesario mantenerlo modular y simple.
- Énfasis en el lenguaje objeto: en la práctica, los compiladores generan diferentes códigos objeto a partir de un mismo código fuente, es por esto que debe especificarse el lenguaje de bajo nivel mediante las características del procesador.

Fases de un compilador

Las fases de un compilador pueden tener ligeras diferencias dependiendo de la forma de estudio, uso, concepción o diseño. A continuación se presenta una estructura típica o común a los diferentes compiladores que servirá para estudiarlos en general. Obsérvese que se puede agrupar en tres grandes partes: front-end, middle-end y back-end; aunque a veces se omite la parte intermedia incluyendo las fases correspondientes en el análisis o síntesis según sea el caso.

El punto de partida para el compilador es tomar un código fuente, en la forma de un archivo, para ser analizado y transformado. Este archivo es escrito en un lenguaje fuente que es apto para el humano, es decir es expresivo, que permite la redundacia y es abstracto, lo que llamamos de alto nivel.

El punto final es la generación de código de bajo nivel que sea óptimo (sin redundacia, disminuyendo la ambigüedad) para el hardware en donde se ejecutará. También es posible que existan otras etapas u optimizaciones dependiendo del lenguaje objeto.

Front-end

- **Análisis Léxico**
  La primer fase incluye un escaneo del código fuente para detectar posibles errores de escritura, como errores de dedo, además de identificar las partes sintácticas mediante la clasificación de **tokens** generando una secuencia de símbolos significativos. También limpia los caracteres innecesarios como comentarios, espacios en blanco y saltos de línea. A esta secuencia de símbolos significativos se le conoce como *token stream*.
- **Análisis Sintáctico**
  También llamado *parsing*. Una vez que ha sido revisado y limpiado de caracteres innecesarios para su ejecución, usando el código es analizado bajo la estructura del lenguaje de alto nivel para obtener una representación de alto nivel, el resultado de esta fase es un *syntax tree* o árbol de sintaxis.
- **Análisis Semántico**
  La representación del código mediante un árbol facilita el análisis semántico para obtener el significado del código fuente. Es aquí que se hacen análisis dependientes del contexto como la verificación de tipos y esta información se incluye en el arb́ol de la fase anterior.

Las primeras dos fases sirven para reconocer la estructura del programa y podemos decir que el *parsing* es la etapa más importante del compilador generando una respresentación equivalente del código fuente (usualmente es un árbol de sintaxis).

Durante estas fases de **análisis** se guarda información en la **Tabla de Símbolos**, que es una estructura de datos para almacenar información sensible del código fuente. Esta tabla interactúa con las tres primeras etapas y será útil en la siguiente fase de síntesis junto con la representación intermedia del código.

El análisis del código fuente en el front-end se considera de alto nivel dado que las representaciones del código utilizadas en estas fases son *legibles*. Se puede decir que en esta parte de análisis se obtiene el "significado" y forma del código fuente.

Middle-end

- **Generación de código intermedio**
  En esta etapa se considera la tabla de símbolos para recuperar la información del código como los identificadores, su tipo, estructura y alcance, etc., para obtener una forma compacta que represente el código fuente respetando su significado.

- **Forma intermedia modificada** u optimización intermedia
  A la representación obtenida anteriormente se le hacen anotaciones (atributos en nodos) para mejorar el código independiente de la máquina, de esta forma genera una representación intermedia independiente. Ejemplos de representaciones intermedias son: árbol de sintaxis abstracta, gráficas de control de flujo, código de 3 direcciones.

Back-end

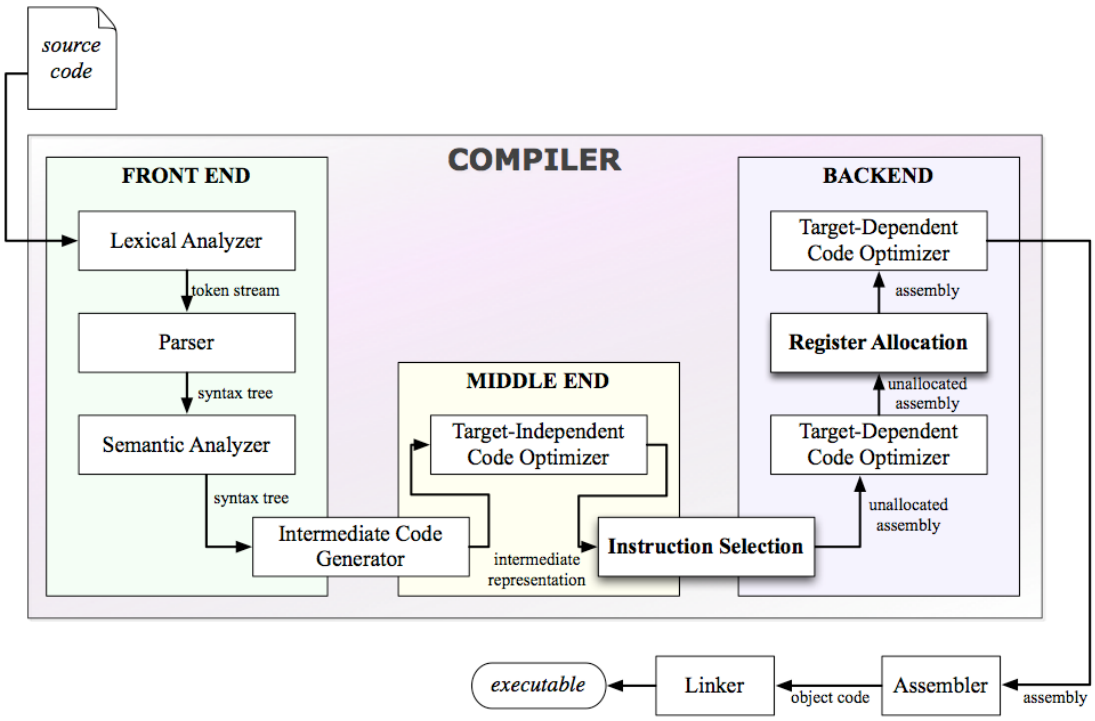
- **Selección de instrucciones**
  Esta fase se encarga de traducir un código intermedio en un código usando un lenguaje muy cercano a ensamblador.

- **Análisis de control del flujo**
  En esta fase se puede obtener la gráfica de control de flujo, es decir la fase en donde se identifican las estructuras de control y bloques para ordenar la ejecución del código,

- **Optimización intermedia dependiente y Análisis estático**
  Considerando una gráfica de interferencia a partir de un análisis de flujo de datos para mejorar el código dependiente de la máquina. Se analiza el tiempo de vida de las variables para optimizar el uso de registros.

- **Asignación de registros**
  Esta fase es la más cercana a la generación de código final o código ensamblador. Es aquí que se maneja la memoria en un nivel bajo lo cual puede o no depender de la máquina en donde se ejecutará el programa.

Existen otras clasificaciones que incluyen a las etapas de modificación del código intermedio y de selección de instrucciones dentro del back-end eliminando la parte media para definir la parte de análisis como las fases del front-end y las de síntesis con las fases del back-end. A continuación se presenta una figura con las fases y productos tomado de [4], este diagrama no es único dado que se pueden realizar optimizaciones en las diferentes etapas de traducción. También es importante agregar que en esa figura no aparece la Tabla de Símbolos, estructura importante para todo el proceso de compilación.



Clasificaciones de los Lenguajes de Programación

Los lenguajes de programación se clasifican de diferentes formas: por la expresividad, por su aparición histórica, por el poder de abstracción y propósito, características semánticas, moda y uso, etc. Estas características también los subclasifican en otras muchas ramas [2]. Una principal clasificación de los lenguajes de programación es la que distingue entre compilación o interpretación, pero existen otras formas de clasificación de programas.

Veamos una clasificación que enfatiza el tipo de lenguaje utilizado para programar:

Alto nivel

- propósito específico & multi-estilo (quinta generación)
- propósito general y manejo especializado (cuarta generación)
- imperativos, declarativos, etc. independientes de máquina (tercera generación)

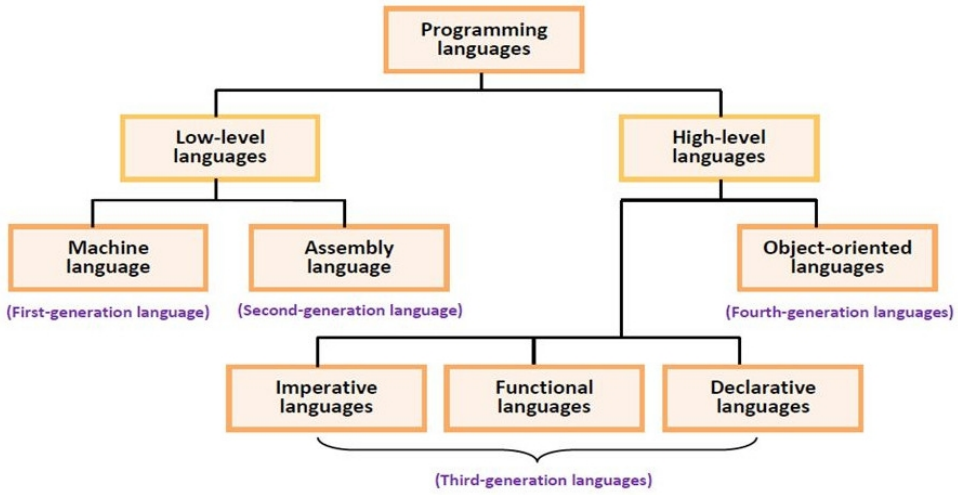
Bajo nivel

- ensamblador (segunda generación)
- lenguaje máquina (primera generación)

La clasificación anterior también es transversal a la clasificación generacional de los lenguajes y que se puede apreciar en la siguiente figura [3]:

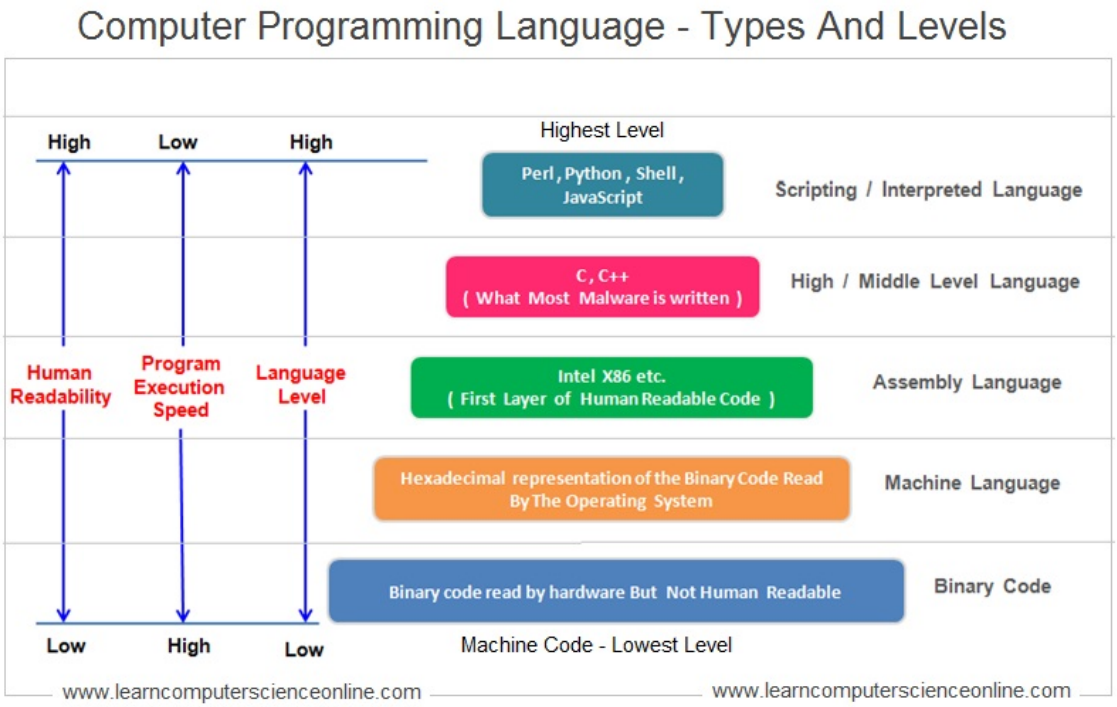
[2] `https://en.ppt-online.org/234073`

[3] Computer Science Course 2301, presentación disponible en SlidePlayer.



Cuando se habla de compilación se piensa que se obtendrá un código en lenguaje ensamblador para ser ejecutado, pero existen diferentes técnicas para obtener un código que será ejecutado en bajo nivel sin que el lenguaje objeto sea necesariamente ensamblador. Así también existen varias formas de interacción entre lenguajes de alto nivel y otros lenguajes. Veamos algunos casos y una imagen [4] que representa varias de las clasificaciones y los aspectos que se utilizan para ellas:

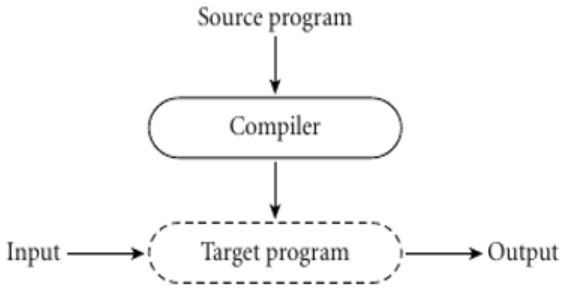
- lenguajes interpretados eg. Basic, COBOL, Ruby, Python, etc.
- lenguajes compilados a un lenguaje intermedio y luego interpretados eg. Java, OCaml, Scala, etc.
- lenguajes compilados a algún otro lenguaje de alto nivel
- lenguajes compilados al vuelo
- lenguajes compilados eg. C, Pascal, etc.



[4] https://www.learncomputerscienceonline.com/computer-programming/

Intérprete vs Compilador

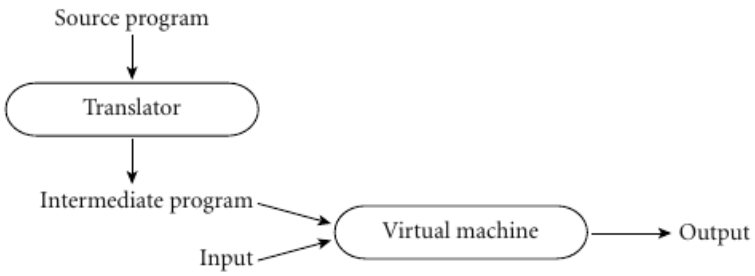
Un compilador traduce un programa de alto nivel a uno equivalente de bajo nivel. Este proceso es complejo, como se puede observar por las fases de transformación de código, pero se realiza una vez (cada vez que se compila) agilizando su uso múltiples veces para diferentes datos de entrada sin necesidad de procesarse nuevamente.



En la imagen anterior se pueden identificar dos momentos importantes para un código, el tiempo de compilación y el tiempo de ejecución. Aunque el tiempo de compilación puede ser muy largo el objetivo es tener una única traducción optimizada del código fuente. Lo anterior con el objetivo de que, en el tiempo de ejecución el código objecto interacúe según sean los datos de entrada y sólo con ellos.

La diferencia con un programa interpretado es que el proceso de interpretación es más simple pero debe de realizarse cada vez que se ejecuta el programa con diferentes entradas. En la interpretación usualmente no hay optimizaciones, pero suele ser más rápido para probar el código. El intérprete no genera código objeto ni traducciones intermedias y el código compilado es generalemente más eficiente.

Muchos lenguajes son implementados usando una combinación de las ventajas que ofrecen los intérpretes y los compiladores al usar una máquina virtual [5]. El lenguaje de esta máquina implementa el propio lenguaje de programación. El programa fuente traducido para servir de entrada a la máquina virtual será ejecutado línea por línea.



[5]Imágenes obtenidas de [5].

En algunos casos, se provee un módulo de preprocesado donde se "limpia" el código fuente al eliminar comentarios y espacios en blanco para generar una secuencia *tokens* equivalente al código fuente.

- si el traductor o preprocesador es sencillo se puede decir que es interpretado sino será compilado
- la máquina virtual es elaborada y es lo que puede indicar que el lenguaje es interpretado
- las fases de análisis y primeras transformaciones son las que caracterizan la compilación

Al finalizar la compilación se obtiene un código objeto que será usado para la ejecución del programa. Justo al terminar la compilación entra el ligador o *linker*, el cual es el encargado de incorporar al código las definiciones de los métodos o o definiciones de las bibliotecas (*libraries*) del lenguaje. El lenguaje ensamblador es cercano a las instrucciones de una máquina o procesador en particular. Un ensamblador traduce al language máquina, dicho lenguaje consta de una serie de identificadores de operaciones y operandos que serán usados. Luego finalmente hace la última traducción usando macros para abreviar secuencias de instrucciones.

Para entender más a detalle las diferencias entre compilador e intérprete se puede revisar la sección 1.1 de la referencia [1] o la sección 1.4 de la referencia [5].

Cabe mencionar que para la ejecución y desarrollo de un programa se ocupan no sólo compiladores e intérpretes si no también existen herramientas alternas que ayudan complementar este proceso, como lo son los *debuggers*, preprocesadores, ensambladores, editores o IDE.

Referencias

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Inc., Second edition, 2007.

[2] Torben Ægidius Mogensen. *Basics of Compiler Design*. Lulu Press, 2010.

[3] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[4] Frank Pfenning. Notas del curso (15-411) Compiler Design. `https://www.cs.cmu.edu/~fp/courses/15411-f14/`, 2014.

[5] Michael Lee Scott. *Programming Language Pragmatics*. Morgan-Kauffman Publishers, Third edition, 2009.

[6] Yunlin Su and Song Y. Yan. *Principles of Compilers, A New Approach to Compilers Including the Algebraic Method*. Springer-Verlag, Berlin Heidelberg, 2011.

[7] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[8] Steve Zdancewic. Notas del curso (CIS 341) - Compilers, Universidad de Pennsylvania, Estados Unidos. `https://www.cis.upenn.edu/~cis341/current/`, 2018.