Introducción
Compiladores 2020-1, Nota de clase 1

Lourdes Del Carmen González Huesca

14 de agosto de 2019
Facultad de Ciencias UNAM

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 pricipios 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

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 por lo tanto uno de los requerimientos 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 o concepción. 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.

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 además de identificar las partes sintácticas mediante la clasificación de **tokens** generando una secuencia de símbolos significativos.

- **Análisis Sintáctico**
  Una vez que ha sido revisado y limpiado de caracteres innecesarios para su ejecución, el código es analizado bajo la estructura del lenguaje de alto nivel para obtener una representación de alto nivel mediante un *parse tree*.

- **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.

- **Generación de código intermedio**
  En esta etapa se considera una tabla de símbolos para clasificar los identificadores, su tipo, estructura y alcance.

Las primeras dos fases sirven para reconocer la estructura del programa y podemos decir que el *parsing* es la etapa más importante del compilador.

Durante esta fase 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.

Middle-end

- **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.

Back-end

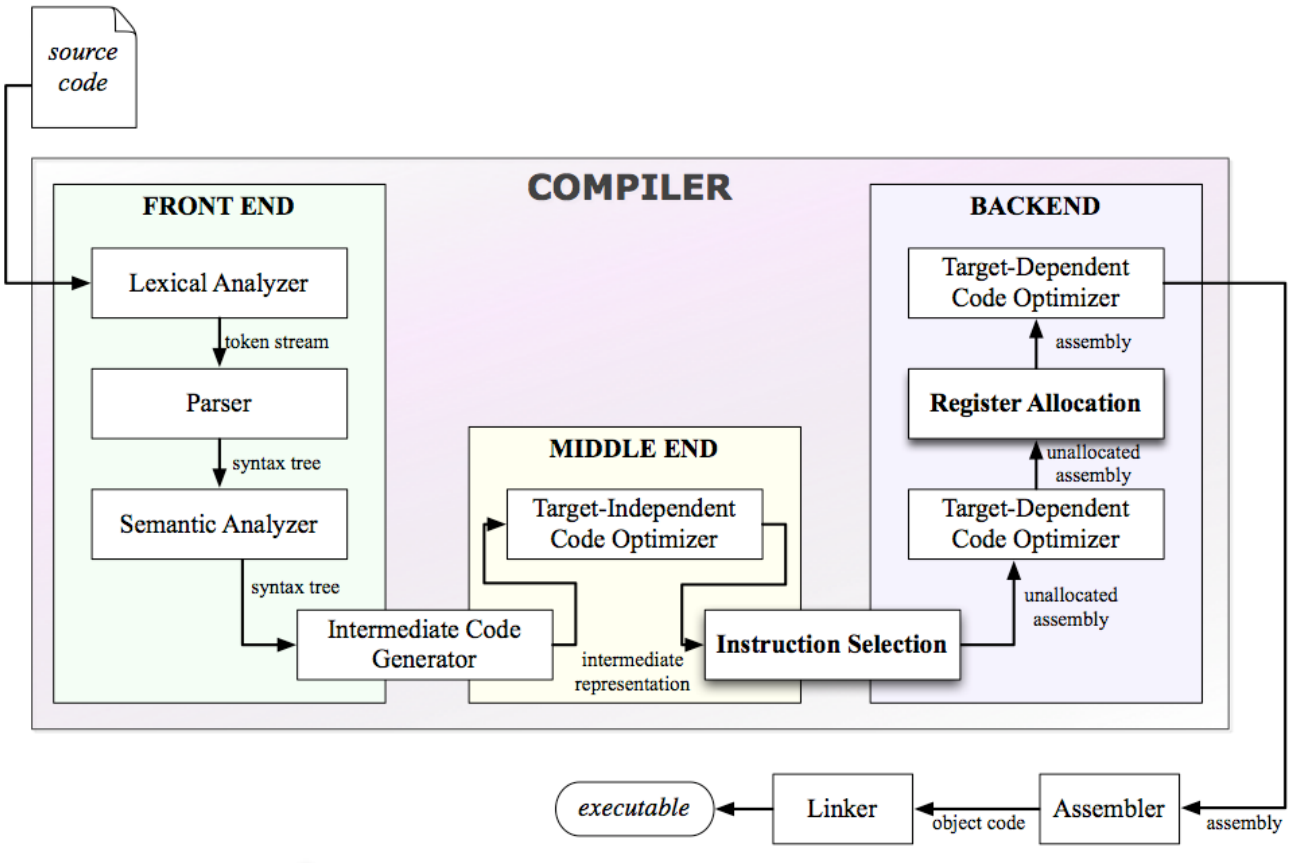
- **Selección de instrucciones**
  Esta fase se encarga de traducir un código intermedio en un código usando un lenguaje muy cercano a ensamblador.

- **Análisis de control del flujo**
  En esta fase se puede obtener la gráfica de control de flujo, es decir la fase en donde se idendifican 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 [3], este diagrama no es único, además se pueden realizar optimizaciones en las diferentes etapas de traducción.



Intérprete vs Compilador

Dos principales ramas de clasificación de los lenguajes de programación son la compilación o interpretación pero existen otras formas de ejecución de programas. 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:

- 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.

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. 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. El intérprete no genera código objeto ni traducciones intermedias y el código compilado es generalemente más eficiente.

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].

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] Jean-Christophe Filliâtre. Curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://www.enseignement.polytechnique.fr/informatique/INF564/`, 2018. Material en francés.

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

[4] François Pottier. Presentaciones del curso Compilation (inf564) école Polytechnique, Palaiseau, Francia. `http://gallium.inria.fr/~fpottier/X/INF564/`, 2016. Material en francés.

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

[6] Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.