Universidad Nacional de Córdoba

Facultad de Ciencias Exactas, Físicas y Naturales

Compiladores y ejecutables

Práctica Supervisada

Informe de Trabajo

Supervisor:

Prof. Maximiliano Eschoyez

Tutor:

Nicolás Papp

Autores:

José Cancinos y ***Julián González***

2021

Resumen

El presente informe tiene como objetivo presentar el trabajo realizado para el cumplimiento de la asignatura Práctica Supervisada. En primer lugar, se presentan los aspectos generales sobre compiladores y la producción de código intermedio. Luego, se pone el foco en el análisis estructural de las **suites** de compilación: GNU GCC y CLANG/LLVM. Posteriormente, es detallado el proceso de generación y código resultante de cada etapa de los compiladores. Luego, se procede a una breve explicación sobre los archivos ejecutables, resultantes del proceso de compilación. Más tarde, se hace foco en los archivos de tipo `ELF`: propiedades y estructura. Finalmente, se exponen los distintos métodos de inserción y posterior lectura de datos dentro de un archivo ejecutable de formato `ELF`.
Los programas de inserción y lectura se emplearán como base para el trabajo de Proyecto Integrador, siendo ambos un inicio para un mejor desarrollo en cuanto a profundidad, abstracción y uso en espacio de sistema (*kernel*).

Índice general

Índice de figuras

Índice de tablas

Listings

Capítulo 1

Compiladores

I Conceptos básicos sobre compiladores

A *Concepto de Compilación*

Compilación: Conjunto de procesos efectuados para obtener un archivo ejecutable a partir de código fuente. [1]

B *Tipos de Compiladores*

Los siguientes conceptos son fundamentales para introducir la clasificación.

- ***Host***: arquitectura de la máquina en la que va a correr el compilador.
- ***Build***: arquitectura que se usa para generar el compilador.
- ***Target***: arquitectura en la que deberá correr el ejecutable generado por el compilador.

Así, se presenta la clasificación de los distintos tipos de compiladores (Von Hagen, [1]) en la tabla 1.1.

C *Estructura de un Compilador*

Las fases que componen un compilador [2] pueden verse en la Fig. 1.1.

1) Analizador léxico

El analizador léxico, también llamado *scanner* o *lexer*, es aquel que lee la secuencia de caracteres del código fuente y los agrupa en otras secuencias llamadas **lexemas**. Por cada lexema, el *scanner* genera un *token* de la forma: La Fig. 1.2 muestra las entradas y salidas de la fase en la parte superior.

Tabla 1.1: Clasificación de compiladores.

Tipo de compilador	Descripción
Compilador nativo	Compilador que genera ejecutables para el mismo tipo de sistema en el cual está operando. (Ídem *build*, *host* y *target*)
Compilador "cruzado"	Llamado *cross-compiler* en inglés, compilador que corre en una arquitectura específica y genera código para otra distinta. (idem *build* y *host*, distinto *target*)
Crossback compiler	El sistema en donde corre y el de ejecución de binarios son iguales pero el sistema *host* es distinto. (idem *build* y *target*, distinto *host*). Se usan para construir un *cross-compiler* que corra en el sistema en el que el compilador corre)
Crossed native compiler	El sistema *target* y el *host* son los mismos pero el sistema en donde se construye el compilador es distinto. Usa un *cross-compiler* para construir un compilador nativo en un tercer sistema. (idem *target* y *host*, distinto *build*)
Compilador canadiense	El sistema en donde se construye, el *host* y el de destino son todos distintos. Un compilador que construye sobre una arquitectura, corre en otra arquitectura y crea código para una tercera arquitectura. (*build*, *host* y *target* son distintos)

$$< \textit{nombre-token}, \textit{valor-atributo} > \tag{1.1}$$

Siendo,

- **nombre-token**: símbolos abstractos usados durante el análisis sintáctico.
- **valor-atributo**: puntero a una entrada en la tablas de símbolos.

Tabla de símbolos El compilador guarda los nombres de variables o nombres de funciones (que aparecen en el código fuente) en la **tabla de símbolos**. También, almacena atributos varios de dicha variable, por ejemplo: tipo, *scope*, argumentos, tipo de pasaje de argumentos, tipo de retorno, etc. Luego, se realiza un mapeo de *tokens* con sus respectivos nombres de la tabla.

2) Analizador sintáctico

El analizador sintáctico, también llamado *parser*, utiliza el primer componente de cada *token* (*nombre-token* en ecuación 1.1) para crear una representación en forma de árbol que muestre la estructura de los *tokens*. Se genera un **árbol sintáctico**. La Fig. 1.2 refleja un esquema de esta fase en la parte superior.

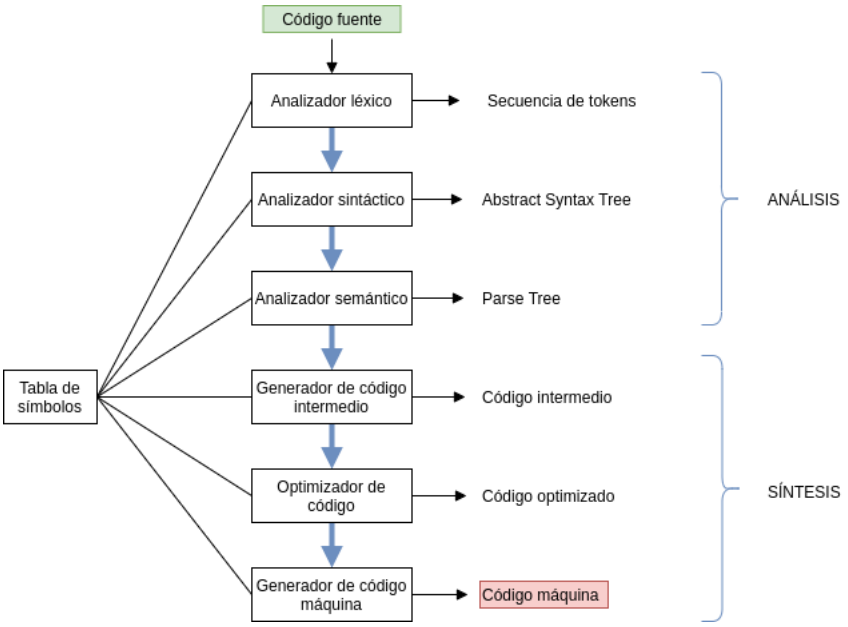

Figura 1.1: Fases de compilación.

3) Analizador semántico

El analizador semántico emplea el árbol sintáctico y la tabla de símbolos (creada por el *scanner*) para revisar la consistencia semántica del código fuente con respecto a la definición del lenguaje. También se realizan conversiones de un tipo de dato a otro (llamadas coerciones). Ver Fig. 1.2 en la fase correspondiente.

4) Generador de código intermedio

El generador intermedio tiene como meta producir código de tipo intermedio para obtener un lenguaje flexible y optimizable para la parte *backend* del compilador. Observar dentro de la Fig. 1.2 para una ilustración de la fase.

El código intermedio puede tener muchas formas distintas (dependiendo de cada compilador).

Propiedades de todo código intermedio Todo código intermedio debe cumplir las propiedades de: *a*) ser fácil de generar; *b*) ser fácil de traducir a lenguaje máquina

Código de tres direcciones Es una forma de código intermedio que consiste en una secuencia de instrucciones cuasi-*assembler* con tres operandos por cada instrucción (cómo máximo). Cada operando puede actuar como un registro.

Algunas características de este tipo de código son:

- Las instrucciones deben tener como máximo un **único operador** (órden de operaciones)

- El compilador debe generar un **nombre temporal** para guardar el valor retornado por la instrucción de tres direcciones (variables temporales tienen nombre).

5) Optimizador de código intermedio

El optimizador de código intermedio realiza mejoras sobre dicho código para obtener un “mejor código” como salida de cada sucesiva optimización. Se presenta también una ilustración en la parte inferior de la Fig. 1.2.

Mejor código = más rápido, más corto o de menor consumo de potencia.

6) Generador de código final

El generador de código final utiliza el código optimizado para generar el código final deseado. Si el lenguaje final es código máquina, se eligen los registros o lugares de memoria para cada variable usada en el programa. En este trabajo, el código final buscado es código máquina. Por ello, se empleará el término "generador de código máquina". En este caso, se traducen instrucciones inmediatas a secuencias de instrucciones máquina. La parte inferior de la Fig. 1.2 muestra una ilustración sobre esta fase.

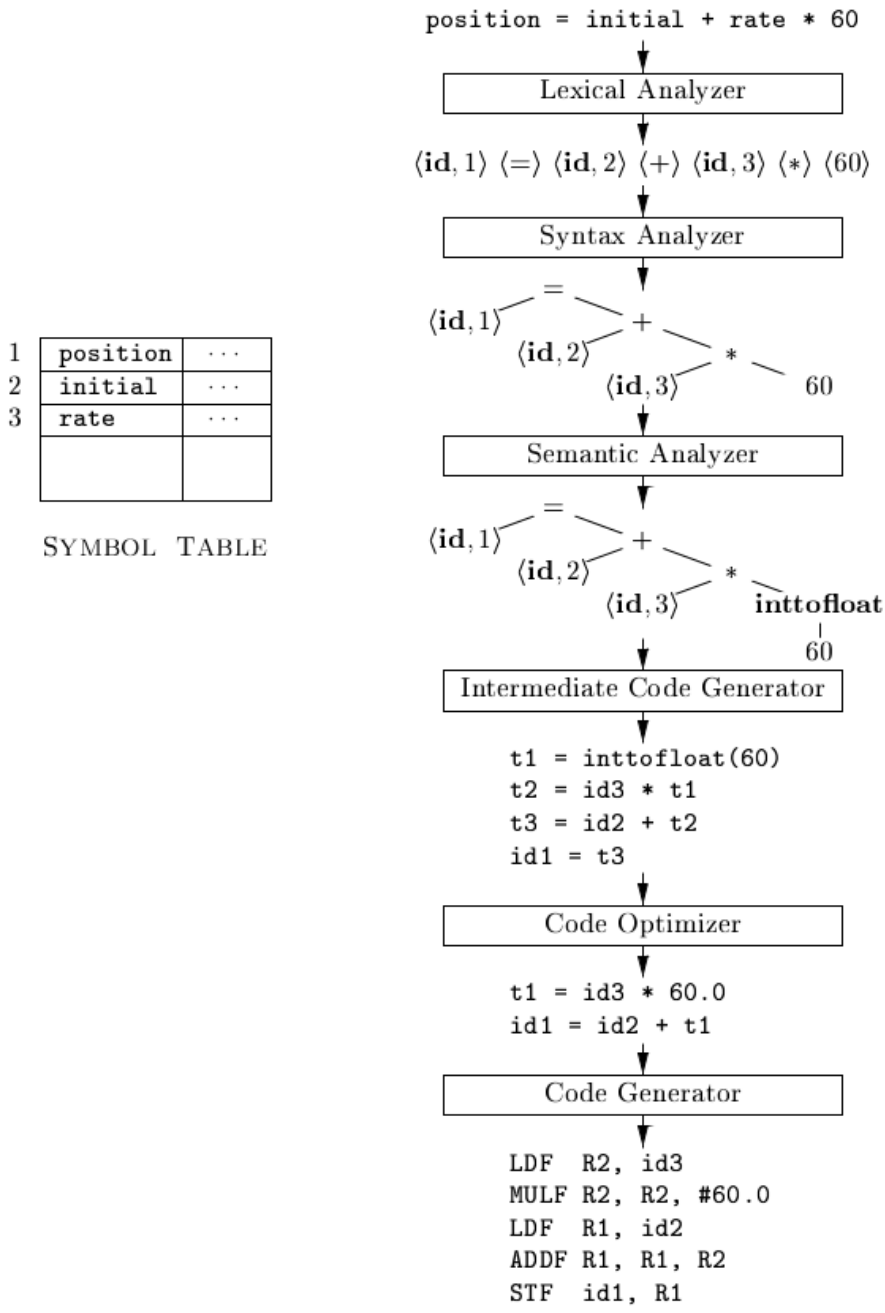

Figura 1.2: Ejemplo completo de fases de compilación.[2]

Bibliografía

[1] W. Von Hagen, *The definitive guide to GCC*, 2nd ed. Berkeley, CA: Apress, 2006, ISBN: 9781590595855.

[2] A. V. Aho y A. V. Aho, eds., *Compilers: principles, techniques, & tools*, 2nd ed. Boston: Pearson/Addison Wesley, 2007, ISBN: 9780321486813.