Diseño e Implementación de Compiladores

Diseño de un compilador para COMPI

Cardellino, Cristian · Leberle, Maico · Soldevila Raffa, Mallku

Diciembre 2015

1. Introducción

Se presenta a continuación el informe del proyecto final de la materia "Diseño e implementación de compiladores", dictada como materia optativa y de posgrado en la carrera de Licenciatura en Ciencias de la Computación de la Facultad de Matemática, Astronomía y Física, en la Universidad Nacional de Córdoba.

El objetivo de este proyecto es el diseño e implementación de un compilador para un lenguaje de programación imperativo simple, de estilo similar a C o Pascal, denominado COMPI.

En el presente informe se reportan las caraterísticas del proyecto, los pasos seguidos en el desarrollo del mismo, y las desiciones de diseño tomadas durante las etapas, entre otros temas de distinta índole respecto al panorama general del proyecto.

El siguiente documento se organiza de la siguiente manera: la sección 2 describe la estructura del directorio del proyecto y explica como compilarlo y correr la suite de tests del mismo. La sección 3 muestra un diagrama de la arquitectura del proyecto, al tiempo que la analiza y describe. La sección 4 hace un repaso general de las decisiones de diseño más importantes que se tomaron a lo largo del proyecto. Finalmente, la sección 6 cierra el informe con una visión general de los puntos fuertes del proyecto y nuestras visiones al respecto.

2. Estructura del Proyecto

El proyecto se encuentra disponible libremente en el repositorio `https://github.com/MaicoLeberle/COMPIcompiler`. La estructura de directorios es la siguiente:

```
./compi
├── bin
├── build
├── doc
│   ├── design_decisions
│   └── report
├── src
│   ├── parser
│   └── tests
└── test
```

En el directorio raíz (`compi`) se encuentra el `Makefile` para compilar el ejecutable de `compi` y crear un ejecutable para correr una suite de tests. Por defecto, correr `make`, crea compi y la suite de tests.

El directorio `bin` es el destino del ejecutable de COMPI una vez finalizado todo el proceso de compilación y enlace del mismo. Además, también es el destino de la suite de tests.

El directorio `build` es el destino de todos los archivos compilados intermedios (es decir,

sin enlace), además del destino de los archivos fuente generados automáticamente por Flex y Bison.

El directorio `doc` tiene dos subdirectorios: `design_decisions` contiene algunas notas sobre las decisiones de diseño que se tomaron a lo largo del proyecto, como el diseño del *Abstract Syntax Tree* (AST), decisiones sobre como trabajar con las llamadas *locations* (i.e. las referencias a un atributo de algún objeto) o decisiones respecto al *scope*; por otro lado, el directorio `report`, contiene el código fuente latex de este informe.

El directorio `src` contiene el código fuente de COMPI. En el subdirectorio `parser`, está el código fuente de Flex y Bison, que vienen a representar la tokenización y gramática del lenguaje compi, a su vez, dentro de este subdirectorio existen otros dos: `parser_asm`, que contiene la tokenización y gramática del código assembly; y `parser_ir` que contiene la tokenización y gramática del código intermedio.

Por otra parte, también dentro de `src`, pero en el subdirectorio `tests`, se encuentra el código fuente de la suite de tests para COMPI.

Finalmente, en el directorio `test`, se encuentra una serie de archivos en código COMPI que son utilizados para los tests.

Como se mencionó previamente, corriendo `make`, se genera el ejecutable de COMPI así como también la suite de tests. El ejecutable del compilador se encuentra en `./bin/compi`, la suite de tests se encuentra en `./bin/test`. Al correr la suite de tests a través del ejecutable correspondiente, corre todos los tests que se escribieron para la aplicación.

3. Arquitectura del Proyecto

La arquitectura puede observarse en la Figura 1. Se muestra aquí el flujo de operaciones desde el código fuente de COMPI hasta la obtención del código Assembly que lo representa.

A partir del código fuente en COMPI, el primer paso se da a través del **análisis léxico y sintáctico**. El lexer determina los tokens a procesar, identificando palabras reservadas, literales, etc. Esto se hace mediante la herramienta `flex`. Se sigue del parser, que toma los elementos devueltos por el lexer y, siguiendo la gramática definida para el lenguaje, crea los nodos del árbol sintáctico. La herramienta que se encarga de parsear es `bison`.

Una vez terminada la etapa de análisis léxico y sintáctico, se sigue la etapa de **análisis semántico**. La etapa arranca a partir del *abstract syntax tree* (AST o árbol sintáctico abstracto), que fue generado en la etapa anterior por el parser. Sobre el AST obtenido se aplica un *patrón visitor*, que crea y utiliza una tabla de símbolos localmente para comprobar distintos elementos de la semántica del lenguaje: tipos y alcance/visibilidad de los identificadores, asig-

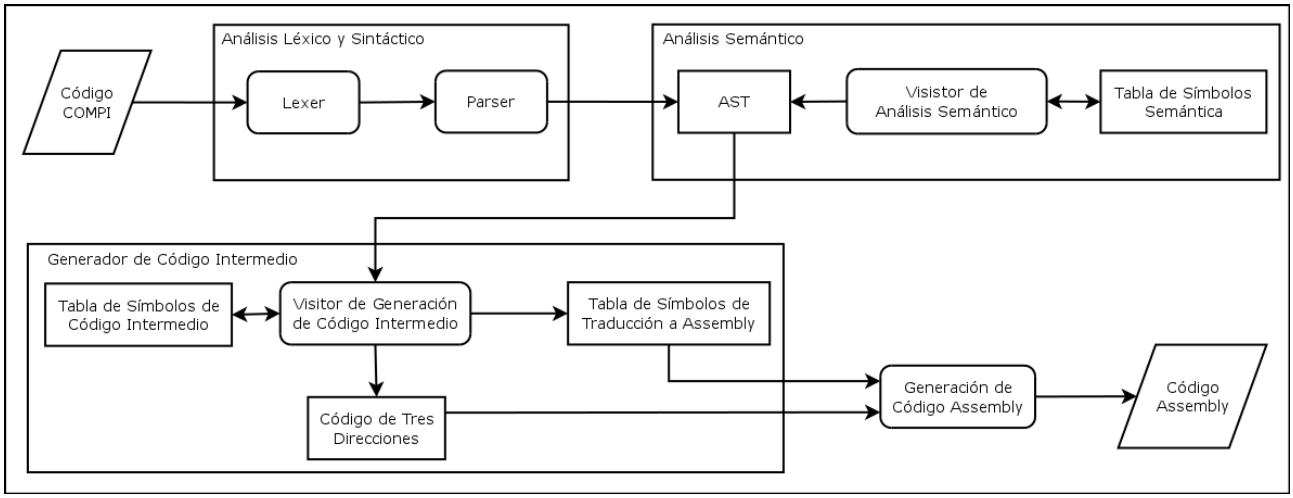

Figura 1: Arquitectura de COMPI

naciones, número de parámetros y valor de retorno en las funciones, etc. Además de revisar con detenimiento ciertas reglas semánticas, como la declaración de identificadores y su llamada, la existencia de un método *main*, la correcta declaración de arreglos, entre otras restricciones definidas en las descripciones del lenguaje.

Concluída la verificación semántica del AST pasamos a la etapa de **generación de código intermedio**. El AST es recorrido nuevamente por un *patrón visitor*, que en este punto busca generar el código intermedio, definido como un código de tres direcciones. A través de una tabla de símbolos interna al proceso, se guarda información del alcance de los identificadores, así como cualquier otra información necesaria para luego generar el código intermedio de tres direcciones (e.g., tipo de la variable, el offset dentro del cuerpo del método, las variable, etc.). Este proceso genera dos recursos: el código de tres direcciones, que viene a ser el código intermedio de la aplicación y la tabla de traducción de símbolos para la traducción a Assembly, que guarda directamente información sobre las variables a utilizar en el código assembly.

A partir de los recursos generados en la parte anterior se entra en la etapa de **generación de código assembly**, que producirá un archivo en código assembly que luego será compilado mediante el compilador correspondiente.

4. Decisiones de Diseño

4.1. Análisis sintáctico y AST

Para la creación del AST nos basamos en la estructura propuesta en la serie de posts de la página *gnuu*, llamado "Writing Your Own Toy Compiler Using Flex, Bison and LLVM"[1]. Este fue adaptado a los requerimientos del lenguaje COMPI. Todos los nodos heredan de una clase principal `node` que a grandes razgos se diferencia entre declaraciones (de métodos y variables), órdenes de flujo (*statements*, en inglés) y expresiones (matemáticas, comparativas, lógicas). Hay algunas enumeraciones para definir los tipos (int, bool, float, etc.), los operadores relacionales (suma, resta, igual, distinto, y, o, etc.) y finalmente operadores de asignación. Se definieron *smart pointers* sobre todas las clases para hacer más sencillo el manejo de memoria.

4.2. Patrón visitor

Antes de describir las decisiones de diseño tomadas en las etapas de análisis semántico y generación de código intermedio, es preciso aclarar cierto mecanismo elegido para implementar dichas etapas: el módulo `visitor` (declarado en `src/visitor.h`).

La clase visitor solo implementa 2 métodos: el modo de "aceptar" (visitar) una expresión y el modo de aceptar una sentencia, en ambos casos en función del tipo de cada una. El resto de los métodos de la clase son métodos virtuales, cuya implementación será establecida por las clases que hereden de visitor.

Nótese también que las distintas clases en `node.h` que representan cada uno de los constructos de un programa heredan (directamente o indirectamente) de la clase `node`, la cual tiene una clase virtual accept, que recibe un parámetro de tipo visitor. La idea es que cada constructo de un programa defina la manera de "aceptarlo"; es decir, de visitarlo mediante una entidad externa, en este caso de tipo visitor. Todos estas clases básicamente implementan accept como una llamada al método visit del visitor pasado como parámetro, con ellos mismos (`*this`) como parámetros.

De este modo, cada clase que herede de visitor define la manera de recorrer un programa, y computar lo que sea necesario. Teniendo esto en cuenta, véanse las decisiones tomadas en cada una de las 3 etapas siguientes del proceso de compilación; a saber, análisis semántico, generación de código intermedio y generación de código assembly.

[1] `http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/`

4.3. Análisis semántico

El análisis semántico, según se pidió que se realice en la sección 3.7 de la descripción del lenguaje COMPI, se implementa fundamentalmente en la clase `semantic_analysis`, declarada en `src/semantic_analysis.h`. Dicha clase hereda de visitor, e implementa el modo de recorrer un programa (una estrucutra de tipo `program_pointer`, definido como `std::shared_ptr<node_program>` en `node.h`) chequeando las características semánticas requeridas por el lenguaje. Para realización de su trabajo, `semantic_analysis` contiene un atributo de tipo `symtables_stack`.

La clase `symtables_stack` modela el scope actual dentro de un programa COMPI, mediante la creación de una pila de objetos de tipo symtable, cada uno de los cuales contiene un `std::map` que asocia un identificador a su información correspondiente, contenida en un objeto de tipo `symtable_element`.

Consideraciones a tener en cuenta en esta etapa:

- Por el modo en que se procesan los métodos y clases dentro de un programa COMPI (i.e., avanzando sintácticamente en el código fuente), es preciso actualizar la información asociada a objetos `symtable_element` de tipo `T_FUNCTION` o `T_CLASS` cada vez que se define un parámetro o atributo nuevo, respectivamente, a medida que se avanza en el análisis del programa. Con este fin fueron creados `symtable_element::put_func_param` y `symtable_element::put_class_field`.

- Como se ha tomado la decisión de no permitir tipos de datos recursivos (i.e., clases que contienen atributos que son objetos de la propia clase que se está definiendo), entonces es preciso, a la hora de chequear que la definición de un atributo en una clase, saber el nombre de la clase que se está definiendo. Esto se realiza mediante `symtable::is_recursive`.

- No se pueden definir dos identificadores iguales dentro de una sola symtable, y el método `symtable::id_exists` chequea esto.

- El análisis del scope creado por las funciones se implementa mediante `symtables_stack::put_fun`, `symtables_stack::put_func_param` y `symtables_stack::finish_func_analysis`. Análogamente, el análisis del scope de las clases se realia mediante `symtables_stack::put_class`, `symtables_stack::put_class_field` y `symtables_stack::finish_class_analysis`. De este modo, la resolución de scopes es transparente para el usuario de `symtables_stack`.

4.4. Generación de código intermedio

Ésta se realiza, fundamentalmente, en la clase `inter_code_gen_visitor` (la cual hereda de la clase visitor también) declarada en `src/inter_code_gen_visitor.h`. Esta clase implementa el modo de generar código intermedio para cierto constructo de un programa COMPI (constructo el cual se encuentra representado como objeto de alguna de las clases declaradas en `src/node.h`).

Se utilizan las constantes definidas en `src/constants.h` (valores iniciales de las variables, y espacio ocupado por cada tipo de variable).

Además, se ha implementado un módulo `three_address_code` (en `src/three_address_code.h`) donde se definen varios tipos de datos que se utilizarán en la generación de código intermedio. El principal es struct quad, que será la forma en que hemos optado representar a las instrucciones de código intermedio: consiste de un tipo de instrucción, un operador (correspondiente a ese tipo de instrucción) y 3 direcciones de memoria (varias instrucciones no usarán la totalidad de estas direcciones de memoria). Además, en este módulo se definen todas las funciones necesarias para construir o chequear propiedades de instrucciones de código intermedio (`std::shared_ptr<quad>`).

Para generar código intermedio es necesario volver a modelar la resolución de scopes como se ha hecho en la etapa de análisis semántico, y este trabajo es realizado por una nueva clase: `intermediate_symtable` (declarada en `src/intermediate_symtable.h`).

Pero `intermediate_symtable` no solo contiene un atributo de tipo `symtables_stack` para modelar la resolución de scopes, sino que también construye, a medida que se van generando las instrucciones de código intermedio (trabajo realizado por `inter_code_gen_visitor`), una tabla de símbolos (clase `ids_info`, también declarada en `intermediate_symtable.h`) que contiene información necesaria para posteriormente poder generar código assembly. Dicha información (almacenada en `ids_info`, con `ids_info::entry_info` como principal reunión de datos) incluye: tipo de identificador (método, variable, variable temporal, etc.), tipo de dato, `std::string` que representa el modo en que cierto identificador ha sido representado en la lista de instrucciones de código intermedio en cierto punto de scope, offset del identificador dentro del registro de activación al que pertenece (para posteriormente poder calcular la dirección exacta del identificador en memoria, en relación al valor de rbp en tiempo de ejecución), número de variables locales si se trata de un método (incluidas las variables temporales, que se almacenan en memoria también), clase a la que se pertenece si se trata de un método o un atributo, lista de atributos o de parámetros (si se trata de una clase o una función, respectivamente).

Además, nótese que `intermediate_symtable` tiene una interfaz similar a la de `symtables_stack`,

con el agregado de las funciones:

- `intermediate_symtable::new_label` (que devuelve una nueva etiqueta auxiliar que el usuario podrá incorporar en su código intermedio con la seguridad de ser única),
- `intermediate_symtable::set_number_vars` (que permite modificar el número de variables, temporales o no, utilizadas en la función actual, lo cual será útil posteriormente al generar el código assembly de registro de activación para dicha función),
- `intermediate_symtable::new_temp` (que registra una nueva variable temporal y le otorga una representación de tipo `std::string` única),
- `intermediate_symtable::get_ids_info` (que devolverá la tabla de símbolos con la información generada durante la generación de código intermedio, necesaria para la generación de código assembly).

5. Generación de código assembly

Teniendo ya una lista de instrucciones de código intermedio junto a la tabla de símbolos (construida en la generación de dicha lista, y que contiene información necesaria para esta etapa), se procede a la generación de código intermedio mediante la clase `asm_code_generator` (declarada en `src/asm_code_generator.h`).

Cabe aclarar aquí que se implementó el protocolo de stack frame (registro de activación) correspondiente al sistema `V AMD64 ABI`. Dicho sistema establece que los primeros 6 parámetros que se desean pasar al llamar a una función son pasados los registros `RDI`, `RSI`, `RDX`, `RCX`, `R8` y `R9`. Los restantes parámetros se pasan "pusheándolos" al stack frame. Técnicamente, el sistema `V AMD64 ABI` otorga los 128 bytes siguientes a la dirección a la que apunta el registro `RSP` para que los utilice la función en ejecución si le resulta necesario para almacenar variables temporales. Pero, por simplicidad, nuestro compilador guarda cada variable temporal en el stack frame, cargándolas en registros si es necesario computar algún valor sobre las variables temporales. Los resultados de tales cómputos también son guardados en el stack frame.

Por último, observar que en el módulo `asm_instruction` (`src/asm_instruction.h`) se definen los tipos de datos análogos a los definidos en el módulo `three_address_code`, pero para esta etapa. Contiene toda la información necesaria para poder generar la lista de instrucciones en assembly, junto a funciones (como `print_asm_instructions_list_intel_syntax`) que sirven para obtener, finalmente, el código assembly.

6. Conclusiones

En términos generales tenemos un balance positivo de este trabajo. El proyecto fue de gran utilidad para entender el proceso de un compilador. Desde el diseño del lenguaje hasta la generación del código objeto. El resultado final fue el mismo compilador de COMPI que se encuentra disponible web.

En el trabajo entregado buscamos seguir los lineamientos propuestos al principio de la materia en cuestión de diseño del lenguaje COMPI. Nos encontramos con algunos obstáculos, algunos de ellos los pudimos solucionar con facilidad y otros quedarían pendientes de disponer de mayor tiempo.

La materia en sí mostró ser de gran utilidad, teniendo en cuenta el transfondo fuertemente teórico que esta carrera posee, trabajando nuevos conceptos, más en el campo de lo práctico que lo teórico.