Reporte de entrega del proyecto final de la asignatura Complementos de Compilación

José Jorge Rdríguez Salgado
Christian Rodríguez Díaz
Hector Adrián Castellano Loaces
Alberto González Rosales
C-411

Curso: 2018-2019

1 Introducción

En el presente reporte se hace un recorrido por los principales aspectos de la implementación de un compilador del lenguaje COOL. Dicha implementación constituye la evaluación final del curso de Complementos de Compilación correspondiente al cuarto año de la carrera Ciencias de la Computación de la Universidad de La Habana.

En la siguiente sección se expone cómo obtener y utilizar este proyecto. La sección número 3 contiene los elementos destacables en la fase de parsing, en la número 4 se expone la fase de chequeo semántico, en la 5 la fase de generación de código y en la 6 se realizan las conclusiones.

2 Requisitos y uso del proyecto

La presente implementación de un compilador del lenguaje Cool fue echa en **Python3**. Para compilar el código Cool contenido en un archivo de entrada y obtener el correspondiente código **MIPS** en otro archivo de salida se debe ejecutar el script de python `cool.py` que se encuentra en el directorio `src` pasando como parámetros el path completo de los archivos de entrada y de salida respectivamente. A continuación un ejemplo:

```
python ./cool.py <input-path> <output-path>
```

En el ejemplo anterior se supone que el comando está escrito en una terminal abierta en el directorio `src`, que `python` se refiere al intérprete de Python3 y los valores entre angulares son el path de los archivos de entrada y salida respectivamente.

Para la fase de parsing fue utilizado el generador de parsers **Lark**, el cual puede ser instalado utilizando el gestor de paquetes **pip** o puede ser descargado de GitHub. A continuación el comando que debe ser escrito para instalar el módulo Lark utilizando el gestor de paquetes pip:

```
pip install lark-parser
```

No existe ningún otro requisito para utilizar el presente compilador que no sean los expuestos anteriormente.

El proyecto puede encontrarse en https://github.com/matcom-compilers-2019/cool-compiler-jj-christian-alberto-hector-c411

3 Parseo

En esta fase se procesa el código COOL y se crea el AST correspondiente. Lo primero que se hace es preprocesar el código de entrada para extraer los comentarios y destacar las palabras reservadas del lenguaje de forma tal que, más adelante, se puedan diferenciar de cualquier identificador.

Como se expuso en la sección anterior, para esta fase se utilizó el generador de parsers Lark.

Lark contiene una pequeña colección de expresiones regulares comunes como las reconocedoras de números, identificadores, etc. A partir de estas es fácil construir los tokens específicos necesarios para el lexer de COOL. Es posible además expecificar una gramática donde los terminales son escritos en mayúsculas y los no terminales en minúsculas. De esta forma Lark diferencia estos símbolos y construye un lexer automáticamente.

Lark da una alternativa al parser *LALR*: el parser *Early*, que es capaz de parsear gramáticas con un gran nivel de ambiguedad, además, puede mostrar en qué momento el código es ambiguo y muestra todas las alternativas posibles. Esto fue de gran ayuda para confeccionar rápidamente una gramática no ambigua que reconociera el lenguaje COOL.

Este módulo contiene una clase *Transformer* que posee las funcionalidades necesarias para procesar el árbol de derivación creado y convertirlo en el AST deseado, utilizando el patrón visitor.

De esta forma se obtiene un AST sobre el cual se realizará el chequeo semántico que se expone en la siguiente sección.

4 Chequeo semántico

En la presente sección se expone cómo se verifica el uso correcto de tipos que exige COOL. Para ello se hizo uso del patrón visitor de forma que, recorriendo los nodos del AST, se logra chequear el cumplimiento de las reglas de tipado presentes en COOL. Todo se realiza en una sola pasada por el AST.

Un programa en COOL se puede ver como una lista de clases donde desde cualquiera de ellas se puede referenciar a las demás. Es por eso, que en el

scope principal deben estar definidas todas las clases sin importar cuál se define primero y cuál después. De esta forma se consigue que desde una clase definida al inicio del programa se pueda hacer referencia a otra definida más abajo.

Para lograr esto, lo primero que se hace es definir en el scope principal todas las clases con sus atributos y métodos. Claro está que estos campos no están verificados aún, por tanto, lo que se pasa al scope es la referencia del nodo de AST correspondiente (nodo método o nodo atributo).

Un primer problema fue lidiar con el *SELF_TYPE* y la variable *self*. Debe destacarse que la primera instancia de un scope que se crea no cubre ninguna clase en particular, o sea, no es el scope interno de ninguna clase sino el scope que abarca todo el programa. En él están definidos los tipos básicos del lenguaje y el resto de las clases definidas por el programador. Pero cada scope que se cree a partir de ese momento va a representar un ámbito interno de una clase particular. Por este motivo se creó el campo *inside* dentro de cada scope, el cual contiene el nombre de la clase dentro de la que está definida esta instancia de scope (a menos que sea el scope inicial que tiene este valor anulado). Así es posible resolver de manera eficiente a qué tipo se refieren *SELF_TYPE* y *self* en cada momento.

En COOL es posible que una clase heredera redefina un método de una clase ancestro. Para ello, el nuevo método debe tener exactamente la misma signatura que el método que se quiere redefinir (mismo nombre, misma cantidad y tipo de parámetros y mismo tipo de retorno). En este aspecto surge la duda de qué hacer cuando se quiere redefinir un método que tenga tipo de retorno o de algún parámetro igual a *SELF_TYPE*. Porque pudiera el nuevo método sustituir este tipo específico por otro que herede del mismo en el momento que se definió el método original. En este aspecto se tomó a cabalidad las instrucciones presentes en el Manual de COOL donde se exige que los tipos deben ser exactamente los mismos, por tanto, debe mantenerse el *SELF_TYPE* para redefinir de forma válida el método deseado.

Para resolver el tipo de las expresiones *case of* e *if then else* se implementaron en la clase Scope las funcionalidades que permiten saber si una clase hereda de otra y el ancestro común más bajo a un par de clases respectivamente. No sólo se puede saber si una clase hereda de otra, sino que también se puede saber la distancia a la que se encuentran (algo que es necesario para resolver el tipo de las expresiones *case of*).

A continuación se expondrá el proceso de generación de código.

5 Generación de Código

Para la resolución de la generación de código se decidió generar un leguaje intermedio para facilitar la futura traducción al lenguaje ensamblador correspondiente(MIPS en este caso).

5.1 Código intermedio

Para la generación de código intermedio se definió todo un mecanismo capaz de, dado el *AST* del lenguaje *COOL*, fuese capaz de generar una especie de *AST* de este lenguaje intermedio. Se recalca el hecho de que es una especie de *AST* porque no es propiamente un árbol sino una lista de nodos de lenguaje intermedio dispuestos en el orden en que se va a generar su código correspondiente en *MIPS*.

El artífice principal de este mecanismo se encuentra en el módulo *transpilator.py*, el cual es el encargado de recolectar todos los *tipos*, *variables*, y *métodos* del programa así como de recorrer el *AST* de *COOL* y generar los nodos intermedios correspondientes. Todo nodo necesario se define en *ilnodes.py* y en el módulo *virtualtable.py* está presente todo lo referente a variables, métodos y atributos, y sus relaciones con los objetos.

Tremendamente útil fue definir para cada nodo intermedio un método *str* que permitiera obtener una representación como *string* de cada nodo. La representación se definió a conveniencia para ayudar en el proceso de encontrar errores en la lógica de la implementación. La idea era tener la capacidad de poder leer también el código intermedio y ser capaz de entenderlo casi tan claro como un lenguaje de alto nivel.

5.2 Código ensamblador

Una vez creada la lista de nodos de lenguaje intermedio se genera el código ensamblador correspondiente. Para lograr una mejor organización se reciben dos listas de nodos intermedios: una correspondiente a la sección *.data* y la otra correspondiente a la sección *.text*.

Lo primero que se genera es la sección *.data*, para lo cual se visitan en el orden en que aparecen los nodos de lenguaje intermedio correspondientes a esta sección y se devuelve para cada uno de ellos su pedazo de código ensamblador correspondiente. En particular los nodos a los que se hace referencia son los que tienen que ver con la declaración de strings, con las relaciones de herencia y las tablas de métodos de virtuales de cada clase. Las tablas de métodos virtuales son etiquetas en *MIPS* seguidas por una lista *.word*, el primero de ellos es una referencia a la dirección de memoria de la sección referente a la herencia de esta clase y los siguientes son referencias a las direcciones de memoria de los métodos de esta clase. Luego se genera el código de los métodos estáticos presentes en el lenguaje *COOL*, los cuales son: *inherit*, *Object.copy*, *Object.abort*, *IO.outstring*, *IO.instring*, *IO.outint*, *IO.inint*, *String.length*, *String.concat*, *String.substr*, *String.cmp* y *substrexception*.

Una vez generada la sección *.data* se procede a generar la sección *.text* de igual manera, o sea, recorriendo los nodos intermedios correspondientes a esta sección y generando una también el código ensamblador correspondiente. En esta sección los nodos que aparecen son los referentes a asignaciones, salida y entrada estándar, comentarios, llamados a funciones, herencia, condicionales, saltos, reserva de memoria, operaciones unarias y binarias, carga y declaración

de etiquetas, así como los imprescindibles nodos referentes a insertar y extraer información de la pila.