Informe sobre Compilador

Héctor Felipe Masson Rosquete C-412
Frankie Mujica Cal C-411
Frank Elier Rosales González C-412

Universidad de La Habana
Facultad de Matemática y Computación
2018-2019

Uso del Compilador

El compilador está implementado en Python, por lo que para usarlo debe ejecutarse el *script* llamado **coolc.py** pasándole como parámetros los ficheros de **COOL** (.cl) para ser compilados. El código COOL de dichos ficheros será unido para formar un solo programa, por lo que solamente uno de los ficheros .cl puede tener una clase **Main**. Tras dicha ejecución se generará un fichero con extensión .asm con el código **MIPS** equivalente al programa de entrada. El nombre del fichero de salida puede ser modificado con el comando **-o file_name.asm** . Dicho fichero deberá ser ejecutado en el **SPIM-Simulator** (simulador de **MIPS**) el cual permitirá la interacción con el programa a través de la consola.

Arquitectura del Compilador

El compilador se divide en cuatro módulos:

- *Lexicography*: Este módulo encierra la lógica para realizar el análisis léxicográfico de un programa de **COOL**. Contiene un fichero con la definición de la gramática de **COOL** (grammar_def.txt) en la cual nos basamos para las reglas de la gramática (grammar_rules.py) y las reglas léxico-gráficas (lexer_rules.py). Para el análisis léxicográfico se ha utilizado la librería **PLY** como generador de *Parser*, el cual recibe como entrada los dos archivos anteriores y devuelve la tabla de parsing del lenguaje (parsetab.py).

- *Semantic*: Este módulo encierra la lógica para el análisis semántico del lenguaje, por lo que contiene implementaciones de conceptos importantes en esta fase del compilador como *environment* (environment.py), *scope* (scope.py), *type* (type.py), así como la implementación del algoritmo LCA (Longest Common Ancestor) con la estrcutura de datos RMQ (fichero rmq_lca.py) y otros conceptos y estructuras de datos que se usan para resolver problemas de tipos y herencia que se abordarán más adelante.

- *Generation*: Este módulo está compuesto por los generadores de **CIL** y **MIPS**, por lo que contiene la lógica para generar un **AST** de CIL a partir de uno de COOL y luego generar código MIPS a partir del AST de CIL.

- *General*: Este módulo contiene las implementaciones de conceptos básicos del compilador como las jerarquías de **AST** para COOL y CIL, un fichero para el manejo de errores en tiempo de compilación y la implementación del patrón **visitor** el cuál es usado para el chequeo semántico y la generación de código. En fin este módulo contiene ficheros que son utilizados en el resto de los módulos del proyecto.

Detalles Técnicos de Implementación

Para comenzar la implementación del compilador fue necesario definir el AST del lenguaje en cuestión (módulo *General*) puesto que sería requerido en cada una de las fases posteriores del compilador.
Estos son algunos de los detalles más importantes de la implementación en cada uno de los módulos:

Análisis Léxicográfico

Para esta fase del compilador fue usada la librería de Python **PLY** como generador de **Lexer** y **Parser**.
Para ello fue necesario definir las reglas lexicográficas tales como las palabras reservadas del lenguaje, la definición de los *tokens* etc.
Además PLY, necesita como entrada, la definición de las reglas de la gramática del lenguaje, la cuál no es más que una clase que contiene un método para cada producción (regla) de la gramática. Dicha gramática presentaba conflicto *Shif-Reduce* pero el generador de *parser* erradicó satisfactoriamente esta contradicción construyendo un *parser* **LALR** para nuestra gramática.

Análisis Semántico

Para el análisis semántico, además de los conceptos mencionados en la sección anterior, fue necesario implementar ciertas estructuras de datos que nos permitieran encapsular toda la información semántica de un programa. Debido a que en COOL, es posible usar una clase que aún no ha sido definida; o sea, que la definición de dicha clase se encuentre después de la línea de código donde se usa; entonces es necesario comprobar primero que dicha clase y sus miembros realmente existen, para después conformar la jerarquía de tipos y por último hacer el chequeo semántico. Estas son las estructuras involucradas en este proceso:

- *Types Collector*: Se encarga de darle una pasada al código para coleccionar todos los tipos definidos en el programa.
- *Types Builder*: Una vez conocidos todos los tipos involucrados en el programa en cuestión, esta estructura se encarga de formar la jerarquía de tipos.
- *Types Cheker*: Una vez conocida la jerarquía de tipos definida en el programa, esta estructura se encarga de realizar el chequeo de tipos de cada una de las expresiones definidas en el programa.

Durante todo este proceso son utilizadas las estructuras mencionadas en la sección anterior como *environment*, *scope* y *rmq_lca*.

Generación de Código Itermedio

Una vez llegados a esta fase ya han sido chequeados todos los errores lexicográficos y semánticos y ya ha sido conformado el árbol de sintaxis abstracta que encierra la semántica del programa que se está compilando. Por lo que el próximo paso a dar es la construcción de un AST para **CIL**, un lenguaje intermedio que suavizará el salto desde COOL hasta MIPS y que resolverá problemas como los *Dispatch* estáticos y dinámicos (la tabla de métodos virtuales). Para ello se implementó una estructura llamada **InheritanceTree** que recoge, organiza y renombra toda la información acerca de los miembros heredados por cada una de las clases que están definidas en el AST de COOL, además también asigna a cada uno de los métodos sobreescritos, el nombre de la función que contiene su implementación, resolviendo así el problemas de los *Dispatch*. Con toda esta información recopilada, se genera entonces un AST de CIL usando el patrón **visitor** que contiene toda la lógica del programa; ahora en un lenguaje más sencillo y en un nivel más cercano a un lenguaje ensamblador.

Generación de Código Ensamblador

Nuestro compilador de COOL, compila para la arquitectura MIPS, por lo que esta fase se trata de traducir el código encerrado en un árbol de sintaxis abstracta del lenguaje intermedio hacia un fichero con el código MIPS equivalente a dicho árbol.
Para ello ha sido usado nuevamente el patrón **visitor**, por lo que, visitando cada nodo del AST de CIL se genera el código MIPS necesario para ejecutar la lógica encerrada en dicho nodo. En esta fase se resuelven e implementan además los métodos de los tipos básicos de COOL: Object, IO, Int, String y Bool.
Para facilitar la implementación en MIPS de todas las expresiones contenidas en un programa, fue necesario tratar a todas las variables como objetos por referencia, dándole por supuesto, semántica por valor a los tipos básicos. Para ello, la información de cada objeto está resumida en una tabla (prototipo) con las siguientes filas:

offset 0	class tag
offest 4	object size
offest 8	dispatch pointer
offset 12	attribute 1
...	...
offset $4(n-1)+12$	attribute n

Prototipo de un objecto en memoria

Class Tag: Un entero de 32 bits que representa un identificador único para cada tipo en el programa.
Object Size: Un entero de 32 bits que contiene el tamaño del prototipo en bytes.
Dispatch Pointer: Un entero de 32 bits que representa la dirección de memoria (referencia) de la tabla de métodos virtuales del tipo con tal Class Tag.
Attribute i: un entero de 32 bits que representa una referencia al prototipo del i-ésimo atributo del objeto.

Particularmente para **Object** *offset 12* no contiene información alguna, para **Int** *offset 12* contiene el entero de 32 bits que representa ese número y *object size* es 16 bytes, para **Bool** *offset 12* contiene cero o uno en dependencia del valor de la variable y finalmente para **String** *offset 12* contiene el String_length, o sea, la referencia a un objeto Int que representa la longitud de la cadena y en *offset* $16 + 4i$ contiene el código ASCII del i-ésimo caracter del String.

El uso de estos prototipos, si bien parece un gasto excesivo de memoria, permite resolver de manera muy sencilla problemas como los métodos *Copy*, *Abort* e *Equals* de Object, así como el problema del *Dynamic Dispatch* referenciando a un lugar en memoria donde se encuentra una tabla que en el *offset* $4i$ contiene la dirección de memoria donde se encuentra la implementación en concreto del i-ésimo método del tipo con dicho Class Tag.
En esta fase además se detectan errores en tiempo de ejecución tales como división por cero, *memory overflow* y *stack overflow*.