Cool Compiler

Alejandro Ojeda Fernández
Ernesto Estevanell Valladares
Liliette Chiu Rodríguez
C-412

1 Repositorio de GitHub

La implementación del compilador se encuentra en un repositorio de la plataforma **GitHub** que trabaja con el sistema de control de versiones **Git**.

Cool Compiler X

2 Uso del Compilador

El compilador está implementado en Python, por lo que no necesita ser compilado previamente a su uso con el makefile propuesto.

Se ofrecen dos opciones de ejecución del compilador:

- Opcion simple (compila un 'cool-code')

```
python CoolCompilerX.py <cool-code> [output-file-name]
```

  El compilador actúa de una manera esperada. Dado un fichero de código COOL se genera código MIPS que se guarda en el fichero de salida pasado como parámetro. Si no se especifica un fichero de salida entonces se guardará el resultado en el fichero **Output/output.mips** relativo a la ubicación del compilador.

- Opción para pruebas múltiples

```
python CoolCompilerX -t [testing-directory]
```

  Esta opción es de gran ayuda a la hora de probar un gran conjunto de ficheros de código COOL de manera independiente. Primeramente se escanea el directorio de pruebas dado (recursivamente) en búsqueda de ficheros de código COOL. Luego se compila cada uno de ellos de manera independiente y se almacenan los ficheros generados en el directorio **Output/** que se creará en el directorio de prueba.

3 Arquitectura del Compilador

El proyecto se divide en los siguientes módulos:

- Sintax: Este módulo encierra la lógica para realizar el análisis lexicográfico de un programa de COOL. Contiene un fichero con la definición de la gramática de COOL, así como las reglas de la gramática y las reglas lexicográficas. Para el análisis lexicográfico 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).
- Semantics: Este módulo encierra la lógica para el análisis semántico del lenguaje, por lo que contiene implementaciones de conceptos y estructuras de datos que se usan para resolver problemas de tipos y herencia, importantes en esta fase del compilador. Contiene los ficheros semantics (semantics.py) y scope (scope.py).

- GeneratingCIL y GeneratingMIPS: Estos módulos están compuestos 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.
- Utils: Este módulo contiene las implementaciones de conceptos básicos del compilador como las jerarquías de AST para COOL y CIL, contiene un fichero con la definición de la gramática de COOL y la implementación del patrón visitor el cuál es usado para el chequeo semántico y la generación de código.

4 Análisis Lexicográfico

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 la gramática.

5 Análisis Semántico

En la presente sección se expone como 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. 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 Checker: 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.

Para el análisis semántico fue necesario implementar ciertas estructuras de datos(class Scope y class Type) que permitieran encapsular toda la información semántica de un programa, estas se encuentran en scope.py. En la clase Type se implementó un algorítmo "conform" que permite, según la definición dada en el manual de Cool, saber si un tipo conforma a otro. También se implementó una función, para resolver el tipo de las expresiones case of e if then else, que permite saber el ancestro común más cercano a un par de clases. No se contempló el SELF_TYPE en el proyecto.

6 Generación de Código Intermedio

Puede ser muy engorroso y complicado generar código a MIPS directamente desde COOL por lo que se utiliza normalmente un punto intermedio entre ambos extremos: Se genera un lenguage intermedio tal que generarlo desde COOL sea más sencillo al igual que generar MIPS partiendo del mismo. Se decidió utilizar el Lenguaje intermedio de 3 direcciones propuesto en el curso (CIL).

6.1 Clases "built in"

COOL cuenta con las siguientes clases "built in": Object, String, Int, Bool, IO. Por cada una de estas clases se definió un tipo correspondiente en CIL. Se creó un tipo adicional Void que representa el vacío. Se decidió plantear como raíz del árbol de herencia de todo programa al tipo Void, sin embargo el único tipo que "hereda" de Void es Object, y, cumpliendo con el manual de COOl, todo tipo distinto a estos "hereda" transitiva o directamente de Object.

Se decidió tratar con todo objeto de COOL en CIL por referencia. Cada objeto Int en COOL será representado como una instancia del tipo Int en CIL, teniendo la instancia un atributo "value" que guardará el valor de dicho objeto de COOL. De manera homóloga se representan los objetos Bool y String con la diferencia de que una instancia de tipo String posee solamente un atributo "msg" que actúa como puntero a una constante String declarada en la sección "data".

6.2 Modelación de clases de COOL

Una clase de COOL está compuesta por atributos y funciones que forman los "features" de la clase. Cada tipo de CIL modela un tipo definido en el código COOL de entrada (o a los tipos base). Estos tipos cuentan con atributos y funciones definidas exactamente en el siguiente orden:

```
1  Type X {
2      [inherited features]
3      <type X attributes>
4      <type X functions>
5  }
```

Por cada tipo se creó una función especial, el constructor, que se encarga de inicializar cada atributo como tipo de CIL (crear las instancias de los tipos correspondientes) y aplicar las expresiones inicializadoras de los atributos en COOL a cada atributo correspondiente. Cada vez que se crea una nueva instancia de un tipo (se aloca en memoria), se realiza un llamado, pasando como parámetro la nueva instancia, al constructor del tipo. Podría preocupar el caso siguiente:

```
1  class X{
2      a:Int <− a + 1;
3  }
```

Pero, el constructor, al inicializar todos los atributos previamente a la generación del código de la expresión inicializadora, permite que antes de cualquier operación sobre cualquier atributo, este mismo, este inicializado con su valor por defecto. Los valores por defecto son: Int : 0 Bool : 0 String : "" Cualquier otro : Void

6.3 Chequeo de errores

En esta fase de compilación se resuelve el código necesario para los errores en runtime siguentes:

- La instrucción case (la expresión principal siendo de tipo Void).
- Los llamados de funciones desde una instancia de Void .

7 Generación de Código Ensamblador

El objetivo es compilar para la arquitectura MIPS, por lo que en esta fase se traduce el código que encierra el 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
offset 4	Object Size
offset 8	Dispatch Pointer
offset 12	Attribute 1
...	...
offset 4(n-1)+12	Attribute n

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.

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 la referencia a un objecto de tipo Int que contiene el valor del length del String y en offset 16 + i 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 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 substring is out of range.