Proyecto de Compilación: PyCOOL537

Marcos Alderete Flores
Roberto Balboa González
Luis Ernesto Martínez Padrón

25 de junio de 2019

1. Uso del Compilador

El compilador `PyCool537` está completamente elaborado en Python (específicamente con su versión 3.6). A pesar de que algunos de los archivos del proyecto fueron generados con el uso de ANTLR4, para el uso del compilador no es necesario tener la librería que utiliza Python para interactuar con esta herramienta. En caso de querer hacer cambios en la gramática que se toma como base de este compilador, es necesario tener JDK8 (Java Development Kit 8) o superior instalado. En todo caso es necesario tener instalada la librería de Python: `antlr4-runtime`.

Para la ejecución del compilador es necesario tener guardado en un archivo (`codigo_cool.cl`) el código COOL a compilar. Luego, en la carpeta del proyecto ejecutar la siguiente línea de comando:

```
python3 pycool537.py codigo_cool.cl
```

De esta forma la salida será dada en un archivo con el nombre `codigo_cool.asm`. De querer especificar el nombre del archivo de salida, puede utilizar la opción `-o` para esto. Por ejemplo:

```
python3 pycool537.py codigo_cool.cl -o archivo_de_salida.asm
```

Para ejecutar el código MIPS generado en el archivo de salida es necesario hacer uso de un compilador (simulador) de ensamblador MIPS. Entre los archivos del proyecto está contenido el compilador MARS, con el cual es posible abrir y ejecutar el archivo `.asm` de salida del compilador `PyCOOL537`.

2. Arquitectura del Compilador

`PyCool537` está dividido en n secciones principales:

- Archivo de inicialización del compilador: `pycool537.py`
- Gramática base(LL(k)): declarada en el archivo `cool.g4`

- Lexer-Parser: definidos en los archivos `cool.interp`, `cool.tokens`, `coolLexer.interp`, `coolLexer.py`, `coolLexer.tokens`, `coolListener.py`, `coolParser.py` y `coolVisitor.py` generados por ANTLR4
- Traducción del árbol de parsing al AST para chequeo semántico (`ast.py` y `ParsingToASTVisitor`)
- Chequeo semántico(`typeCheck.py`, `context.py` y `contextType.py`)
- Traducción del AST a árbol de código CIL(`cil_ast.py`, `pre_visitor.py` y `cool_to_cil_visitor.py`)
- Generación de código MIPS y generación del archivo de salida.

3. Problemas Técnicos

3.1. Representación de tipos en memoria

Los tipos fueron representados como direcciones de memoria en las cuales se guardan referencias a los atributos correspondientes y otras propiedades que se añadieron para guardar información referente a las instancias como las siguientes:

- Class Tag: un índice para identificar a las clases.
- Object Size: un número que representa el tamaño, contado en cantidad de palabras, de los objetos que se guardan en la instancia.
- Dispatch Table: una referencia a la tabla de métodos virtuales de la clase instanciada.

Luego de estas propiedades extra, se encuentran representados los atributos.

3.2. Ocultamiento de variables

Para realizar esta característica del lenguaje se utilizó un diccionario de Python en el cual se guardan como llaves los nombres de las variables locales que están declaradas y como valores la lista de los nombres de las variables locales en las cuales se ha representado el valor guardado en la función de variables con ese nombre. O sea, que en esas listas el último nombre da variable local es el valor que se toma dado que fue el último valor asignado a la variable pedida.