Universidad De La Habana

MATCOM

Complementos de Compilación

Compilador de COOL

Pin y Pon:
Olivia González Peña C411
Juan Carlos Casteleiro Wong C411

Introducción

Nuestro trabajo se centra en la construcción de un compilador para el lenguaje COOL. Se encargará de, dado un programa escrito en COOL, en caso de que existan, reportar los errores sintácticos y semánticos, y, en caso contrario, de generar el código MIPS correspondiente al mismo.

Nuestra implementación se encuentra dentro de la carpeta src y sigue la siguiente distribución:

- code_generator
  - cil_ast.py
  - BaseCOOLToCILVisitor.py
  - COOLToCILVisitor.py
  - BaseCILToMIPSVisitor.py
  - CILToMIPSVisitor.py
- cparser
  - parser.py
- lexer
  - lexer.py
- semantic
  - visitors
    - typeCollector.py
    - typeBuilder.py
    - varCollector.py
    - typeChecker.py
  - semantic.py
- utils
  - ast.py
  - errors.py
  - utils.py
  - visitor.py

Arquitectura del Compilador

Para llevar a cabo el proceso de compilación de un código de COOL, nuestro compilador atraviesa tres fases principales:

- Análisis sintáctico: En esta fase, se construye una gramática para el lenguaje y, en función de esta, se realiza un Árbol de Sintáxis Abstracta (AST). Se representa el código fuente del cual se parte en dicho árbol a través de los procesos de tokenización y parsing.
- Análisis semántico: Esta fase se encarga de revisar y validar las reglas y predicados inherentes al lenguaje. Para ello, utiliza la información almacenada en el AST construido en la fase anterior.
- Generación de código: Corresponde a la fase final, que genera el código MIPS equivalente al código fuente escrito en COOL. Para ello, primeramente, realiza un paso adicional que consiste en la traducción a un lenguaje intermedio y, es a partir de este último, que se genera el código MIPS con el que concluye el proceso de compilación.

Análisis sintáctico

Esta fase, a su vez, se divide en dos subfases. La primera de ellas es la tokenización, que consiste en el procesamiento de las sentencias del código recogiendo toda la información necesaria en forma de tokens. La segunda, que constituye el parsing, se utiliza para reconocer la sintaxis del lenguaje especificado en forma de gramática libre del contexto.

En estos dos procesos se hace uso de las herramientas lex y yacc del paquete ply de Python. Yacc utiliza la técnica de análisis sintáctico LR o shift-reduce, teniendo como salida un AST representativo del programa de entrada. Estas herramientas brindan además, facilidades para el reporte y manejo de errores lexicográficos y sintácticos.

Las soluciones a los procesos de tokenizar y parsear están incluidas en los archivos lexer.py y parser.py, respectivamente.

Análisis semántico

El objetivo de esta fase es validar que los predicados semánticos se cumplan. Para ello, se hacen varios recorridos con el patrón visitor por el AST que genera la fase anterior. Hay dos conceptos que resultan esenciales para manejar estos resultados, que son Context y Scope, cuyas implementaciones se encuentran en semantic.py. El Context se construye con la información de los objetos, mientras que el Scope permite verificar la visibilidad y accesibilidad de estos para su correcto uso.

El primer recorrido lo realiza el TypeCollector, que se encarga de registrar todos los tipos declarados en el código, además de incluir los tipos built-in de COOL. Estos tipos serán almacenados en el Context y, luego, el TypeBuilder se encargará de construirlos como tal, con sus funciones y atributos correspondientes.

El próximo recorrido lo realiza el VarCollector, que se encargará de inicializar un Scope para cada objeto, además de incorporarle a este las variables accesibles para él. Por último, el recorrido que realiza TypeChecker verifica la consistencia de los tipos de todos los objetos del programa. Para esto, fue necesario establecer la relación de conformidad entre los tipos, de forma que el TypeChecker permita donde se espera un objeto con cierto tipo X, un objeto de tipo Y, tal que Y se conforme de X.

A lo largo de todos los recorridos se van recopilando los errores semánticos obtenidos, especificándose la línea y columna del código fuente donde se encuentra cada error.

Generación de código

Como se indicó anteriormente en esta fase se realizan dos procesos fundamentales, la traducción de COOL a un lenguaje intermedio CIL, y posteriormente la generación de código MIPS a partir de este. Optamos por esta estrategia para aminorar la diferencia entre los niveles de abstracción de los mismos, y con ello la complejidad del salto.

COOL to CIL

Para esta etapa se parte de recorrer el AST de COOL, nuevamente utilizando el patrón visitor, generando para cada nodo de este la información necesaria para su representación en CIL. Todo programa de CIL contiene las secciones .TYPES, .DATA y .CODE, la información referida a estos se almacena en las variables dottypes, dotdata y dotcode, respectivamente. En dotcode se almacenan las instancias de los FunctionNode del AST de CIL a través de las llamadas al método register_function, análogamente register_type almacena las instancias de TypeNode del AST de CIL en dottypes. El FunctionNode que se esté creando es almacenado en la variable current_function, y es posible definirle parámetros, variables locales e instrucciones haciendo uso de los métodos register_param, register_local y register_instruction, respectivamente.

CIL to MIPS

De igual forma, se hace uso del patrón visitor para recorrer los nodos del AST de CIL y generar el código MIPS asociado a estos. Dentro de la clase Context se crea una estructura graph que representa la jerarquía de clases del programa. Al realizar un recorrido DFS sobre esta, se le asigna a cada tipo un

tag que indica el orden en que han sido recorridos los tipos, de forma tal que la clase Object tiene tag 0. El tag asociado a cada objeto se encuentra en el offset 0 a partir de su dirección en memoria. En el offset 4 se tiene un puntero a la dirección del segmento de datos donde se encuentra el nombre de la clase; le sigue en el offset 8, el tamaño de la clase, para definir hasta dónde hay datos asociados a la misma. Las funciones tanto inherentes a la clase como heredadas serán apuntadas desde el offset 12, y a partir del offset 16 se encontrarán los valores de los atributos tanto de la clase como heredados.

Representación de los objetos						
offset	0	4	8	12	16	...
	tag	*nombre	tamaño	*funciones	atributo	atributo

Tanto las funciones como los atributos de una clase estarán dispuestos en el orden en que son declarados, comenzando así por los heredados de sus ancestros. De esta forma, se garantiza que puedan ser localizados en el mismo offset que en sus ancestros. En el caso de las funciones, de ser redefinidas se sustituye en el offset correspondiente la dirección de la nueva definición. Esta distribución facilita el polimorfismo de los objetos.

Para gestionar el llamado a funciones se colocan en la pila los parámetros de esta y los valores temporales necesarios para realizar una operación, en el registro $a1 se almacena el valor de retorno. Desde la fase anterior se conoce el espacio que necesitaremos reservar para estos valores temporales así como el offset de los mismos para referenciarlos con el $sp.

Problemas Técnicos y Teóricos

Durante el proceso de análisis sintáctico se presentaron dificultades para la tokenización de comentarios y strings de múltiples líneas, que fueron resueltas utilizando los estados que provee el paquete ply.

En el recorrido del VarCollector durante el análisis semántico, presentó especial interés el trato de las expresiones de tipo Let y Case. En el caso del Case fue necesario declarar un Scope único para cada variable, de modo que la accesibilidad de la misma quedara restringida a este. En el caso del Let, todas las variables declaradas en el letBody, manejadas en el VarDeclarationNode del AST de COOL, serían solo accesibles desde el Scope que se declara para la expresión del inBody.

La expresión Case, en el proceso de generación de código, también requirió especial atención ya que para lograr el comportamiento esperado, que selecciona la opción cuyo tipo sea el más restrictivo tal que el tipo dinámico de la expresión Case se conforme a él, fue necesario recurrir a los tags previamente mencionados asocioados a la jerarquía de clases, priorizando aquel tipo con mayor tag que cumpla la relación de conformidad.

Uso del compilador

Para el uso del compilador es necesario tener instalado Python 3.7 o superior, se utilizan además los paquetes ply, pytest y pytest-ordering. Ply para la construcción del lexer y parser, mientras que pytest y pytest-ordering para la ejecución de los test automáticos. Estos paquetes pueden ser instalados ejecutando desde la raíz del proyecto:

```
pip install -r requirements.txt
```

El fichero pricipal del proyecto es main.py, el cual recibe como argumentos la dirección de otros dos ficheros: inPath y outPath. InPath es la dirección del fichero que contiene código COOL y outPath es la dirección del fichero donde se guardará el código MIPS generado. El archivo main.py puede ser encontrado en la carpeta src y se ejecuta de la siguiente manera:

```
python3 main.py <inPath> <outPath>
```

Observaciones

Para la implementación del proyecto se siguieron las pautas marcadas en las conferencias y clases prácticas, utilizando clases y métodos definidos en estas, añadiendo las modificaciones necesarias para lograr ciertos comportamientos.

Por ejemplo, en la fase de análisis semántico durante los distintos recorridos al AST, se utilizaron las clases Context, Scope, Type, Method y Attribute, entre otras.

De igual forma, en la fase de generación de código para la traducción a CIL, se utilizaron las implementaciones de BaseCOOLToCILVisitor y sus funciones para la definición de COOLToCILVisitor, y siguiendo este, realizamos el diseño de BaseCILToMIPSVisitor y CILToMIPSVisitor.