COOL Compiler

Miguel Alejandro Asin Barthelemy MIGUEL.ASIN@ESTUDIANTES.MATCOM.UH.CU
Grupo C411

Gabriel Fernando Martín Fernández GABRIEL.MARTIN@ESTUDIANTES.MATCOM.UH.CU
Grupo C411

Tutor(es):
Dr Alejandro Piad Morffis, *Universidad de la Habana*

Resumen

Se describe la implementación de un compilador para el lenguaje COOL (Classroom Object-Oriented Language). El funcionamiento de este se divide en varias etapas: tokenizar, generar el lexer, parsear, análisis semántico, generación de código intermedio CIL (Common Intermediate Language) y generación del código de ensamblador MIPS. Para las primeras tres etapas se emplearon las herramientas proporcionadas por la librería ply, específicamente ply.lex y ply.yacc para realizar el análizis lexicográfico y el parseo generando al final de este proceso un AST (Abstract Syntax Tree). Para el chequeo semántico se definió una clase Scope en forma de árbol que contiene en cada nodo las definiciones y tipos empleados en el nodo del AST equivalente. En esta fase se visitan todos los nodos y se realiza el chequeo de tipos y la comprobación de errores semánticos. Posteriormente con el scope y el AST se procede a generar otro árbol que en este caso contiene el código de CIL equivalente al AST de COOL original. Este proceso se realiza visitando cada nodo y generando los nodos del CIL equivalentes. Finalmente el arból con el código CIL se procedió a transformar sus nodos en código de ensamblador MIPS equivalente mediante un proceso similar al anterior.

Abstract

The implementation of a compiler for the COOL language (Classroom Object-Oriented Language) is described. Its operation is divided into several stages: tokenize, generate the lexer, parse, semantic analysis, CIL (Common Intermediate Language) intermediate code generation and MIPS assembly code generation. For the first three stages, the tools provided by the ply library were used, specifically ply.lex and ply.yacc to perform the lexicographical analysis and parsing, generating an AST (Abstract Syntax Tree) at the end of this process. For the semantic check, a Scope class was defined in the form of a tree that contains in each node the definitions and types used in the node of the equivalent AST. In this phase, all nodes are visited and type checking and semantic error checking are performed. Later, with the scope and the AST, another tree is generated, which in this case contains the CIL code equivalent to the original COOL AST. This process is done by visiting each node and generating the equivalent CIL nodes. Finally, the tree with the CIL code proceeded to transform its nodes into equivalent MIPS assembler code through a process similar to the previous one.

Palabras Clave: Compilador, tokens, lexer, parser, AST, semántica, CIL, lenguaje de ensamblador.
Tema: Compilador de COOL.

1. Introducción

La tarea fundamental de un compilador se puede resumir en traducir código escrito en un lenguaje de programación determinado a otro de más bajo nivel con el objetivo de crear un programa ejecutable. Debido a la gran cantidad de lenguages de programación que han surgido en los últimos tiempos la necesidad e importancia de los compiladores ha crecido. En este sentido el proyecto descrito en este informe es un ejemplo sencillo de de la estructura y componentes que presentan algunos de los compiladores modernos.

El lenguaje al cual se le confeccionó el compilador es COOL (Classroom Object-Oriented Language), el cual fue diseñado para su uso en la ensañanza de cursos de compilación para estudiantes universitarios. Este lenguaje es relativamente pequeño pero contiene muchas de las funcionalidades que presentan los lenguajes de programación modernos como son los objetos y un fuerte tipado estático.

Para la confección de dicho compilador fue necesario primero dividir el proceso en varias etapas más simples y procesar por pasos la entrada inicial obteniendo en el camino una serie de resultados intermedios hasta llegar a producir el código final objetivo, que en este caso es en el lenguaje de ensamblador de MIPS. En este informe se abordarán los detalles y particularidades de este proceso de compilación.

2. Detalles de la Implementación

2.1 Lexer

Lo primero a tener en cuenta en el proyecto fue la necesidad de tokenizar el código COOL de entrada. Para esto se decidió usar expresiones regulares y la clase token definida en la librería ply de python. Se deben definir primero apropiadamente cada uno de los posibles tokens del lenguaje y luego cuales son las palabras claves reservadas y los tipos por defecto. Esta librería permite manejar los errores al tokenizar de forma sencilla y solo fue necesario cambios menores para lograr la misma estructura de reporte de errores en todo el proyecto. La definición de los tokens puede verse en *Lexer/tokens.py*.

El lexer requiere definir las reglas que se tienen que tener en cuenta a la hora de tokenizar incluyendo los comentarios y el manejo de errores. Usando estas reglas y los tokens definidos anteriormente se pasa a construir el lexer con respecto a la entrada recibida. Para construir el lexer se emplea la sintáxis y las herramientas proporcionadas por ply.lex. Todo este proceso se encuentra implementado en *Lexer/Lexer.py* y genera el archivo *lextab.py* donde queda guardado el lexer. Los tipos de errores posibles en este proceso se definen en *Lexer/errors-types.py*.

2.2 Parser

Para realizar el parser primero necesitamos definir la gramática de COOL la cual se encuentra especificada de buena manera en los manuales del lenguaje provistos por la Universidad de Standford. Describimos las producciones de esta usando la sintaxis requerida por el parser de la librería ply. Debido a las características de COOL se decidió optar por un parser LALR(1), presente igualmente en ply. La implementación de las reglas empleadas por este parser se puede ver en el archivo *Parser/Parser.py* y el parser generado por ply se puede ver en *yacctab.py*.

Para que se pueda proceder a la siguiente fase de análisis semántico es necesario que el parser genere un AST con un formato que facilite las futuras tareas y para ellos se definió este AST en el archivo *ast.py*. Luego de parsear la entrada se produce precisamente un AST con dicha estructura el cual será empleando para las posteriores fases.

2.3 Análisis Semántico

Para la fase de análisis semántico primeramente se definieron una serie de clases y métodos para facilitaran su implementación. Debido a que una parte esencial de esta fase es el manejo de tipos fue necesario definir una clase que conteniera las características de los tipos predefinidos y de aquellos que se añadiesen posteriormente además de sus propiedades y métodos. Esto se encuentra en *Semantic/types.py*.

Otro factor importante a tener en cuenta es el scope de las variables pues así se puede tener una referencia de las variables ya declaradas cuando se visita un nodo determinado del AST y facilita grandemente el trabajo con los tipos y el funcionamiento básico como tal de la semántica. El scope usado se encuentra definido en *Semantic/scope.py*.

Para facilitar el recorrido de los nodos del AST durante el chequeo semántico se incluyó el archivo Semantic/visitor.py que brinda herramientas útiles que permiten definir un método visit para cada tipo de nodo donde el tipo de este es el que define cual de entre todos los métodos visit utilizar en cada caso.

Antes de proceder a confeccionar el analizador semántico como tal debemos aclarar que las reglas semánticas fueron tomadas del propio manual de COOL y como tal su funcionamiento exacto viene aclarado en ese documento.

Para realizar el análisis semántico nos basamos en el AST que se tiene ya confeccionado y en las herramientas antes citadas. Se comienza en el nodo Program que es la raíz del AST y se prosede a visitar a sus hijos para realizar el chequeo semántico. Las definiciones de cada método visit empleado según los nodos que se visiten y el funcionamiento general de este analizador podemos encontrarlo en *Semantic/Semantic.py*.

En este proceso descrito también se identifican los errores semánticos que puedan se identificados en la entrada. Los tipos de errores identificados en esta fase y su significado se definen explícitamente en *Semantic/error-types.py*.

El producto del análisis semantico es un "árbol de scope" con una distribución de nodos similar al AST generado previamente donde un determinado nodo de dicho árbol representa el scope que se emplea en el nodo equivalente del AST, donde se continene los tipos y clases definidos en dicho nodo y son accesibles los previamente definidos en un scope más general. Además se devuelve un nuevo AST similar al anterior al análisis semántico pero con los valores de los tipos actualizados.

2.4 CIL

Una vez realizado el chequeo de tipos, el siguiente paso es traducir el código de lenguaje COOL a un codigo intermedio (CIL) de bajo nivel que permita más adelante obtener el equivalente del mismo en lenguaje de máquina. El por qué de este paso se debe a la complejidad que conlleva traducir el codigo de un lenguaje de alto nivel a ensamblador. Características como:

- Lenguaje de instrucciones de 3 direcciones
- Todas las variables son enteros de 32 bits
- Tiene soporte para operaciones orientadas a objetos

convierten a CIL en un lenguaje intermedio hacia el cual es posible traducir desde COOL y es un buen punto de partida para la generación de código máquina.

Todas las expresiones en COOL se traducen en un conjunto de instrucciones a CIL, donde cada una de ellas es representada por un nodo determinado por el tipo de instrucción, así por ejemplo: una expresión en

COOL de tipo:

$$a <- if\ x < y\ then\ <expr1>\ else\ <else2>\ fi$$

se traduce a CIL como:

$t0 = < expr1 >$
$t1 = < expr2 >$
$BRANCHLESSTHAN\ x\ y\ Then\_label$
$GOTO\ Else\_label$
$Then\_label:$
$t2 = t0$
$GOTOEnd\_if\_expr$
$Else\_label:$
$t2 = t1$
$End_if\_expr:$
$a = t2$

Para ello se definen en *Codegen/Intermediate/cil.py* el conjunto de todos los nodos que representan las instrucciones posibles de CIL, de esta forma, cada expresión en cil es representada como una instancia de dichos nodos, a fin de facilitar posteriormente la conversión de dichas instrucciones a código máquina.

Para crear las instancias de los nodos de CIL, se realiza un "*visitor*" por cada uno de los nodos del AST obtenido del análisis semántico y, se crean las instrucciones CIL correspondientes. Una evz terminado la conversión, se obtiene un nuevo árbol de instrucciones que será procesado por el generador de código MIPS.

2.5 Lenguaje de ensamblador MIPS

Antes de proceder a transformar el CIL generado en instrucciones de ensamblador MIPS fue necesario una serie de pasos. Primeramente se definieron en *CodeGen/Assembler/mips-instructions.py* prácticamente todas las instrucciones de ensamblador salvo las empleadas para el trabajo con "floats" y "doubles" (pues estos no son tipos de COOL). Esto se hizo para permitir ver las instrucciones como clases dentro del código interno del compilador y que luego estas adopten su formato adecuado a la hora de generar el código de ensamblador MIPS.

Como parte del proceso de conversión de código intermedio a código máquina, se realiza nuevamente un "visitor" por los nodos del árbol devuelto por el generador del código intermedio. En cada nodo se realizará la conversión de la instrucción CIL analizadá al lenguaje MIPS, a partir de las instrucciones definidas en $mips - instructions.py$.

Debido a que en lenguaje ensamblador solo se trabajan con valores de 32 bits, fue necesario crear una definición de tipo para las variables. Para ello, a partir de cada declaración de una clase (junto con los tipos predefinidos), se crea una plantilla en la sección *.data* del fichero *.mips* de dicho tipo, con los valores: cantidad de atributos, métodos, nombre del tipo, tipo de asignación (valor o referencia), tipo de comparación, y clase de la que hereda.

Tras crear una instancia de un tipo, se libera un espacio en memoria igual a: $(cantidad\_de\_atributos + 1) * 4$. En la primera posición de dicho espacio se almacena la direccion de la plantilla del tipo al que pertenece la variable, y en el resto, el valor de sus atributos, luego en la variable se almacena solo la dirección de dicho espacio en memoria. Esto brinda la facilidad de conocer el tipo de cualquier variable accesible y su valor (atributos).

En la definición del lenguaje, se consideran ciertos tipos predefeinidos, varios de ellos con métodos incluidos (IO, Object, String), devido a la invariabilidad de dichos métodos en cualquier código de COOL, se consideró crear los mismos en un fichero *code.mips* (junto con otras macros de utilidad), las cuales se añadirán al código mips generado una vez realizada la compilación.

3. Conclusiones

El proyecto presentado ilustra algunos de los distintos procesos y pasos necesarios para la confección de un compilador de un lenguaje de programación moderno. Si bien COOL es relativamente sencillo ejemplifica claramente los retos que conlleva esta tarea y sirve como una experiencia introductoria a la confección de compiladores de lenguajes más complejos. Además, este proyecto todavía tiene un buen margen de mejora y puede ser empleado como base para implementar y ejemplificar varios métodos de optimización de código y otras técnicas empleadas en los compiladores.

4. Recomendaciones

Al compilador propuesto se le pueden agregar algunas funcionalidades para mejorarlo. Entre estas podemos citar un mejor manejo de errores en tiempo de ejecución y un garbage collector para liberar la memoria no utilizada. Además existen varias técnicas para optimizar el código generado que pudiesen mejorar la eficiencia en general del compilador tanto en su uso de memoria como en el tiempo de ejecución.

Referencias

[1] A. Piad, J. P. Consuegra, R. Cruz *Introducción a la Construcción de Compiladores*. Universidad de la Habana.

[2] Torben Ægidius Mogensen *Basics of Compiler Design*. University of Copenhagen, 2010. URL: http://hjemmesider.diku.dk/ torbenm/Basics/basics-lulu2.pdf.

[3] Alex Aiken *The Cool Reference Manual*. Stanford University, 1995. URL: https://theory.stanford.edu/ aiken/software/cool/cool-manual.pdf.

[4] David Beazley *PLY (Python Lex-Yacc)*. URL: https://ply.readthedocs.io/en/latest/index.html.

[5] James R. Larus *Assemblers, Linkers, and the SPIM Simulator*. University of Wisconsin–Madison, 1998.