Complementos de Compilación
COOL-Compiler

Dalianys Pérez Perera
Dayany Alfaro González
Gilberto González Rodríguez
C-411

Índice

1. Introducción

COOL es un pequeño lenguaje orientado a objetos, con tipado estático, herencia simple, polimorfismo, recolección automática de basura y un sistema de tipos unificado. Admite control de flujo condicional e iterativo, además de la coincidencia de patrones. Todo en COOL es una expresión.

El propósito de la asignatura Complementos de Compilación es desarrollar durante el curso un compilador para el lenguaje COOL que sea capaz de convertir programas escritos en Cool en programas escritos en MIPS.

2. Arquitectura del compilador

2.1. Análisis Lexicográfico y Sintáctico

En estas dos fases se emplearon las herramientas de construcción de compiladores lex y yacc a través del paquete de *Python* `ply`. El mismo incluye la compatibilidad con el análisis sintáctico LALR(1), así como la validación de entrada, informes de errores y resulta ser bastante exigente con la especificación de reglas gramaticales y de tokens.

PLY consta de dos módulos separados: `lex.py` y `yacc.py`. El primero se utiliza par dividir el texto de entrada en una colección de tokens especificados por una colección de reglas en forma de expresiones regulares(*tokenización*), mientras que el segundo se utiliza para reconocer la sintaxis del lenguaje que se ha especificado en forma de gramática libre del contexto. Las dos herramientas están diseñadas para trabajar juntas.

El módulo `yacc.py` implementa la componente de *parsing* de PLY teniendo como salida un árbol de sintaxis abstracta representativo del programa de entrada. Yacc utiliza la técnica de análisis sintáctico LR o *shift-reduce*. Es importante señalar que tiene como requisito la especificación de la sintaxis en términos de una gramática BNF(notación de Backus-Naur). Esta notación es utilizada para expresar gramáticas libres del contexto. En `grammar.md` se encuentra especificada la gramática utilizada en el proyecto.

2.2. Análisis Semántico

Durante esta fase se analiza el cumplimiento de todos los predicados semánticos del lenguaje, y por tanto, el uso correcto de los tipos declarados. A continuación centraremos la atención en el problema de la verificación de dichos tipos.

Primeramente tenemos que hacer un recorrido por todo el *AST* para encontrar las definiciones de tipos, las cuales serán almacenadas en el concepto `Context`. Esto lo haremos utilizando el patrón *visitor*, el mismo será utilizado en las siguientes pasadas al *AST*. Es importante conocer los nombres de las clases definidas de antemano , ya que podemos tener una declaración de un tipo A con un atributo de tipo B, donde la declaración del tipo B aparece luego de la de A. Con la clase `TypeCollector` se logra crear un contexto inicial que solo contendrá los nombres de los tipos, por eso es que solo visita los nodos del *AST* de tipo `Program` y `Class`.

```
class TypeCollector(object):
```

```
        def __init__(self, errors=[]):
        self.context = None
        self.errors = errors

        @visitor.on('node')
        def visit(self, node):

        @visitor.when(AST.Program)
        def visit(self, node):

        @visitor.when(AST.Class)
        def visit(self, node):
```

Importante mencionar que la información referente a los tipos se almacena en el contexto a través de la clase `Type`, y que la misma incluye la función `conforms_to` con el objetivo de establecer la relación de conformidad entre tipos y garantizar el principio de sustitución.

Posteriormente se pasa a construir los tipos como tal, con sus definiciones de atributos y métodos, por tal motivo se visitarán además los nodos que definen a los mismos en el *AST*. Durante esta pasada de la clase `TypeBuilder` también se chequea que la jerarquía de tipos conformada esté correcta y sea un árbol con raíz en el tipo *Object*. Esto último se resolvió comprobando si el grafo de tipos representa un orden topológico.

Por último tenemos un `TypeChecker` que verificará la consistencia de tipos en todos los nodos del *AST*. El mismo recibe el contexto construido anteriormente y procesa por completo el *AST*. A lo largo de este recorrido fue esencial el uso del concepto `Scope`, el cual permite gestionar las variables definidas en los distintos niveles de visibilidad, así como saber con qué tipo se definieron. Cada clase tiene su propio ámbito o *Scope* y a su vez cada método definido en esta. No obstante, la importancia de este concepto también se demuestra en el chequeo de las expresiones *Let* y *Case*, pues ambas poseen un ámbito interno para nuevas variables locales que podrían definir, por tanto, el *Scope* "hijo"que se le pasa a estos nodos permite desambiguar entre todas las variables declaradas.

Para manejar los errores de forma consistente, cada una de las clases anteriores posee como atributo una lista de errores de tipo `ErrorSemantic`. De modo que ante cualquier error de chequeo de tipos, simplemente se crea una instancia de esta clase y se añade a la lista. Aclarar que a cada error se le pasa la línea y columna del nodo del *AST* correspondiente.

2.3. Generación de Código

Esta fase comprende dos etapas esenciales. Primeramente es necesario traducir el código de COOL a un lenguaje que nos permita generar código de forma más sencilla. Este lenguaje se denomina *CIL* y todo programa en él tiene 3 secciones: .TYPES, .DATA y .CODE. Durante esta etapa se realiza un recorrido del AST de COOL y se obtiene un *AST* de *CIL*, representando toda la información y semántica necesaria del programa de COOL. Con ello logramos un mayor nivel de abstracción al disponer de instrucciones en 3-direcciones y de cualquier cantidad de registros.

Durante la segunda etapa se realiza un recorrido sobre el *AST* de *CIL* conformado para

generar el código de MIPS finalmente. En ambas, se emplea el patrón *visitor*.

2.3.1. CIL

El módulo correspondiente a la generación del código intermedio es `cool_to_cil.py`. Aquí se encuentra el recorrido realizado sobre el *AST* de COOL por medio de la clase `COOLToCILVisitor` la cual hereda de `BaseCOOLToCILVisitor`. Esta última contiene una serie de atributos claves y métodos auxiliares para facilitar la generación:

- Las variables de instancia `dottypes`, `dotdata` y `dotcode` almacenan los nodos correspondientes a las secciones .TYPES, .DATA y .CODE respectivamente de un programa en *CIL*.
- Las variables `current_type` y `current_method` almacenan instancias de `Type` y `Method` respectivamente.
- La variable `current_function` almacena el nodo cil.FunctionNode que está en proceso de construcción (estos nodos pertenecen a la sección .CODE).
- Para definir parámetros, variables locales e instrucciones dentro de `current_function` se usan las funciones auxiliares `register_param`, `register_local` y `register_instruction` respectivamente.
- Los métodos `register_function` y `register_type` almacenan instancias de cil.FunctionNode y cil.TypeNode en las variables `dotcode` y `dottypes` respectivamente.

2.3.2. MIPS

Puntualizar:
.Object Layout
.Polimorfismo
.LLamado a funciones

3. Problemas técnicos

Mencionar quizás:
. comentario múltiple
. herencia
. lowest commun ancestor

4. Uso del compilador

Cómo se ejecuta