Complementos de Compilación
COOL-Compiler

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

Índice

1. Arquitectura del compilador

De manera general la estructura de nuestro compilador se divide en varios módulos, pertenecientes a cada una de las fases por las que atravesó la implementación del mismo. A continuación se explicarán las fases mencionadas y, a su vez, los módulos correspondientes.

1.1. 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 para 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*. Tanto lex como yacc proveen formas de manejar los errores lexicográficos y sintácticos, que se determinan cuando fallan las reglas que les fueron definidas. 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.

El lexer y el parser del proyecto se encuentran implementados en los módulos `lexer.py` y `cparser.py` respectivamente, además `ast_nodes.py` ofrece la jerarquía del AST de COOL propuesta.

1.2. 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.

1.3. 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*.

1.3.1. 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.

1.3.2. MIPS

En el módulo `cil_to_mips.py` se encapsula el proceso de generar código MIPS. Se tiene la clase `CILToMIPSVisitor` que se encarga de recorrer el *AST* de *CIL* generado anteriormente.

- **Objetos**
  A la hora de generar código MIPS es necesario establecer ciertos convenios, y uno de los más importantes es la forma en que se van a organizar los objetos en memoria. Para mostrar el diseño utilizado se va a hacer uso de las clases declaradas en la Figura 1.

```
class A {
    a : Int <- 0;
    f(): Int { a };
};

class B inherits A {
    b : Int <- 1;
    f(): Int { b };
    g(): Int { a <- a + b };
};

class C inherits A {
    c : Int <- 2;
    h(): Int { a <- a - c };
};
```

Figura 1: Declaración de clases de ejemplo en Cool.

En la Figura 2 a la izquierda se muestra como quedarían dispuestos en memoria los objetos de tipo A, B y C definidos en la Figura 1. Para cada objeto se va a tener que, a partir de su dirección de memoria, en el *offset* 0 se va a encontrar un tag que va a ser un número único para cada clase. Este número se va a corresponder con el orden en que se visita cada clase en un recorrido de tipo *DFS* sobre el árbol que representa la jerarquía de clases comenzando por la clase `Object` que siempre tendría tag 0. A continuación en el *offset* 4 se tiene un puntero que apunta a una dirección del segmento de datos donde

se encuentra alamacenado el nombre de la clase. En el *offset* 8 va a estar contenido un número que representa el tamaño de la clase, lo cual se podría ver como cuántos cuadros ocupa. Le sigue en el *offset* 12 un puntero que apunta a una dirección donde van a estar dispuestas las funciones declaradas por la clase o heredadas, lo cual se puede apreciar en el cuadro que se muestra a la derecha y se explicará con más detalle luego. Por último a partir del *offset* 16 se van a encontrar el valor de los atributos declarados por la clase o heredados. Estos atributos van a aparecer en el orden en que fueron declarados, empezando por los pertenecientes al ancestro más lejano hasta llegar a los declarados por la propia clase.

Offset Class	0	4	8	12	16	20
A	Atag	“A”	5	*A	a	
B	Btag	“B”	6	*B	a	b
C	Ctag	“C”	6	*C	a	c

Offset Address	0	4
*A	fA	
*B	fB	g
*C	fA	h

Figura 2: Forma de colocar los objetos en memoria.

En el cuadro que se encuentra en la parte derecha de la Figura 2 se tiene cómo se van a organizar las funciones pertenecientes a una clase. Como se puede observar se va a seguir el mismo criterio de orden que en los atributos. La diferencia es que una clase puede redefinir las funciones heredadas y esto se va a reflejar como una sustitución en el *offset* que le corresponde a la función heredada,como se muestra en el ejemplo, que B redefine la función f y por tanto en ese *offset* va a apuntar a su propia definición, al contario que C que va apuntar a la definida por A.

Este diseño está pensado para lograr el poliformismo en los objetos pues, tanto para atributos como funciones, siempre los heredados van a estar en el mismo *offset* que en los ancestros, lo que beneficia que en los casos que un clase quiera sustituir a un ancestro sus atributos y funciones van a tener la misma forma de buscarse.

En el caso de las clases *built-in* también se va a hacer uso del diseño explicado y los valores de estas van a estar almacenados como un atributo. El valor `void` va a estar representado como una dirección estática en memoria.

- **Funciones**

  La forma en que se resuelven los llamados a funciones incluye otros de los convenios utilizados. Uno de los principales problemas a resolver fue dónde se iban a almacenar los parámetros que recibe una función. Para esto se decidió hacer uso de la pila, de forma que cuando se ejecuta la primera instrucción de una función los parámetros van a estar colocados en la pila en orden contrario al que fueron declarados y se podrá acceder a ellos usando como referencia el registro `$sp`.

  Otra situación a resolver es que para llevar a cabo cualquier operación es necesario almacenar valores temporales, que denominaremos *locals* de una función. Estos valores

también van a estar almacenados en la pila. Desde CIL se tiene conocimiento de cuántos *locals* se van a necesitar y lo primero que va a hacer una función es "salvar" espacio en la pila para guardarlos y se va a tener conocimiento del *offset* que va a corresponder a cada *local* que también van a ser referenciados usando `$sp`.

Al final de la ejecución de una función el valor que esta retorna se va a almacenar siempre en el registro `$a1`. Por último la función se encarga de sacar de la pila tanto los *locals* como los parámetros.

2. Problemas técnicos

2.1. Análisis Lexicográfico y Sintáctico

Durante la fase lexicográfica el mayor reto enfrentado fue la tokenización de *strings* y comentarios de múltiples líneas. Esto fue resuelto mediante el uso del concepto de estado que provee la herramienta `ply`. Cuando se detecta `"` o $\backslash*$ el lexer entra en un nuevo estado en el que solo se regiría por las reglas definidas para dicho estado. Este comportamiento se mantendría hasta que se detecte `"` o $*\backslash$ y se regresa al procesamiento normal. En el caso de los comentarios se mantiene un contador de los $\backslash*$ encontrados para manejar los comentarios anidados.

2.2. Análisis Semántico

Mencionar quizás:
. herencia
. lowest commun ancestor

2.3. Generación de Código

Uno de los problemas más interesantes que se presentó fue la implementación de la funcionalidad que provee la expresión `case`. Es necesario dado un tipo `T` determinar, en tiempo de ejecución, entre un conjunto de tipos $T_i, 1 < i \leq n$ cuál es el menor tipo con el que `T` se conforma. Para resolver esto se le asignó a cada tipo un tag, el cual se va a corresponder con el orden en que se visita cada clase en un recorrido de tipo *DFS* sobre el árbol que representa la jerarquía de clases comenzando por la clase `Object` que siempre tendría tag 0. Además cada tipo `T` va a saber cuál es el tipo con mayor tag $max\_tag(\mathrm{T})$ al que se puede llegar en un recorrido *DFS* desde él. Por la forma en que está definido el tag se puede decir que un tipo `T` se conforma con un tipo `A` si $\mathtt{tag_A} \leq \mathtt{tag_T} \leq max\_tag(\mathtt{A})$. Por tanto si se chequean los tipos T_i ordenados de mayor a menor según el tag el primero que cumpla dicha condición va a ser el tipo buscado.

3. Uso del compilador

Cómo se ejecuta