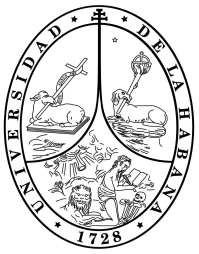

Complementos de Compilación
"COOL-Compiler"

Est. **Alain Cartaya Salabarría** A.CARTAYA@ESTUDIANTES.MATCOM.UH.CU

Est. **Andres Sainz Álvarez** A.SAINZ@ESTUDIANTES.MATCOM.UH.CU

Est. **David Cancio Milian** D.CANCIO@ESTUDIANTES.MATCOM.UH.CU

C-411

Índice

1 Introducción

El propósito de la asignatura Complementos de Compilación es llevar a cabo el desarrollo de un compilador para el lenguaje de programación orientado a objetos *COOL*, **Classroom Object Oriented Lenguage** , por sus siglas en inglés, haciendo uso de los conocimientos aprendidos en la asignatura Compilación cursada el pasado año.

2 Compilador

El objetivo primario de nuestro compilador, como su nombre indica, es lograr realizar el proceso de compilación para un archivo **.cl* a un archivo de mips **.s* pasando por un lenguaje intermedio.

3 Fases de Compilador

Atendiendo a lo estudiado en el curso de Compilación, las fases por las cuales se rige nuestro compilador son:

1. Parser.
2. Análisis Semántico.
3. Generación de Código.

Parser

Primeramente se llevó a cabo el diseño de una gramática que represente las reglas sintácticas de lo que es un código en *cool*. Para esto, utilizamos la herramienta *ANTLR* para la generación de un parser que resolviera esta gramática. Este generador de parser nos brinda un *Lexer*, un *Parser* y un conjunto de *interface IBaseVisitor* para hacer un recorrido sobre los nodos del árbol de derivación para obtener un *AST* (*Árbol de Sintaxis Abstracta*) de un código de *COOL*.

Árbol de Sintaxis Astracta

La jerarquía de nuestro *AST* viene determinada por el tipo raíz *AST – Node*, el cual representa el mayor nivel de abstracción de nuestro árbol, el cual está constituido de la siguiente manera:

```
public abstract class AST_Node
{
        public List<AST_Node> Children;
        public int row, col;
        ...
        public override T Visit<T>(IASTVisitor<T> visit) => visit.Visit(this);
}
```

row y col son el referente a la línea y la columna de un nodo del AST en el texto.

Sobre este *AST* se implementó el patrón *Visitor* para recorrer este árbol y realizar los chequeos semánticos asi como la generación de código en *CIL*.

Además hizo falta representar un nivel de abstracción para los nodos que constituyen una expresión en *COOL* para identificar estos nodos y conocer su tipo estático, atributo que sería muy útil en las restantes fases.

```
public abstract class AST_Expresion : AST_Node
{
        public SemanticType MyType { get; set; }
        ...
}
```

Luego la fase *Parser* recibe un código de *COOL*, primero se obtiene un árbol de derivación de la gramática y finalmente con un recorrido sobre este se obtiene un *AST* con el cuál sería la entrada de la fase *Chequeo Semántico* .

Un tipo semántico esta definido por su nombre, su padre, una lista de métodos y una lista de atributos que lo definen:

```
public class SemanticType
{
        public string Name { get; set; }
        public SemanticType Father { get; set; }
        public List<SemanticAttr> Attrs { get; set; }
        public List<SemanticMethod> Methods { get; set; }
            ...
    }
```

En las instancias de estas clases estarían almacenadas la semánticas de cada uno de los tipos de *COOL*, básicos y definidos por el usuario, con el cuál se resolverían varios problemas como el *SELFTYPE*, la utilización correcta de métodos y atributos, etc;

Análisis Semántico

Para realizar el análisis semántico fue esencial la creación del concepto *contexto* o *scope*, el cual representa el ámbito o la sección del código en la que nos encontramos. Esto es de gran utilidad para conocer si la variable ya fue declarada antes de su uso, obtener el tipo de una variable, etc.

```
        public interface IContext
        {
                bool IsDefine(string var);
                bool Define(string var);
                bool IsDefineInMe(string var);
                IContext CreateChild();
                HashSet<string> GetScope();
                IContext GetParent();
                SemanticType GetType(string field);
                bool SetType(string var, SemanticType type);
                bool ChangeType(string var, SemanticType type);
                SemanticType GetTypeInMe(string var);
                bool NullFather();
        }
```

Además de esto contamos con una clase *ErrorLogger* para archivar los errores semánticos existentes en el código.

Estos solo fueron conceptos necesarios para adentrarnos en el chequeo semántico en si. Para esto se implementaron cuatro *Visitors* sobre el *AST*: El primero para verificar que en la jerarquía de clases del programa no exista herencia cíclica, las redefiniciones de los métodos en clases herederas cumplan con la misma signatura de los ancestros, etc. Otro para verificar que las variables utilizadas en las instrucciones ya estén previamente declaradas, o sea que no se utilicen variables u objetos antes de su declaración, otro para hacer lo referente al chequeo de tipos y actualizar los tipos estáticos de las expresiones, y por último pero no menos importante verificar que los nombres de variables no sean ni tipos definidos ni palabras reservadas, y que los tipos asociados a las variables estén declarados.

En lo referente al chequeo de tipos, se apreciaron algunos casos interesantes, tales como en los argumentos de una expresion *let*, es necesario visitar primero su inicialización, porque pudiéramos chocar con algo como estp: *let x:Int <- x + 1* donde *x* pudiera estar previamente definida en el contexto actual. También se dieron casos interesantes tales como en el *if*, donde el tipo estático es el *lowest commun ancestor* del tipo estático del *then* y el del *else*.

Generación de Código

Como se mencionó con anterioridad la generación de código consta de 2 fases, primero a partir de un recorrido sobre el *AST* se genera un $AST - CIL$, un árbol de sintaxis abstracta del lenguaje *CIL* que representa el código de *COOL* que necesitamos compilar y luego otro recorrido a este $CIL - AST$ que devuelva el código en *MIPS*. Las ventajas de primero compilar para *CIL* son muchas, entre ellas encontramos que permiten un nuevo nivel de abstracción ya que en *CIL* todas las instrucciones son en $3 - direcciones$ y es muy parecido a los lenguajes de ensamblador, con la abstracción de tener infinitos registros para realizar las operaciones, elemento que no es posible en *MIPS* pero permite una mayor expresividad a la hora de recorrer el $AST - COOL$ para la generación de código.

CIL

La transpilación del $AST - COOL$ al $AST - MIPS$ fue hecha con el patrón *Visitor* antes mencionado, donde se recorría todo el *AST* y se iba recopilando toda la información necesaria para construir el $AST - CIL$. Primero se creaba el $Section - Type$, luego el $Section - Data$ y después con la ayuda de un contexto función, encargado de la creación automática de labels, variables locales, ocultamiento, etc, se iban creando las funciones en *CIL*.

MIPS

Una vez que tenemos el $AST - CIL$ que representa el código de *COOL* que se quiere compilar, la complejidad de la generación de código *MIPS* se reduce drásticamente provocando que gran parte de las operaciones se traduzcan de forma sencilla y legible. Sin embargo, hay tres funcionalidades que se llevan el peso de la generación de *MIPS*, ellas son : la abstracción del manejo de objetos , la creación de los contextos y llamadas a funciones y el manejo de tipos para resolver los problema de la herencia y el polimorfismo.

Tipos

La definición de un objeto(tipo) se encuentra en la sección de los datos donde se guardan todos los elementos necesarios para resolver cualquier situación en las que pueda estar un objeto de este tipo.

La definición de un tipo es una dirección de memoria tal que:

Sea dx la dirección de la definición del tipo T

$mem[dx] = sizetype$ donde $sizetype$ es el número de *word* que tiene esta definición en memoria.

$mem[dx + 4] = objectsize$ donde $objectsize$ es el número de *bytes* necesarios para crear un objeto de este tipo.

$mem[dx + 8] = father$ donde $father$ es el padre del tipo T, en caso de $Object$ es $void = 0$

$mem[dx + 12] = typename$ donde $typename$ es la dirección de memoria donde se encuentra su nombre.

a partir de $[dx + 16]$ empiezan las direcciones donde se resuelven sus funciones.

Un ejemplo del data de un código en *Mips* generado, donde solamente están los tipos básicos de *Cool*:

```
type_Main: .asciiz "Main"
type_Object: .asciiz "Object"
type_Int: .asciiz "Int"
type_IO: .asciiz "IO"
type_String: .asciiz "String"
type_Bool: .asciiz "Bool"
error_null: .asciiz "Null␣Reference␣Exception"
error_div0: .asciiz "Divition␣By␣Zero␣Exception"
error_indexout: .asciiz "Index␣Out␣Range␣Exception"
Object: .word 7, 8, 0, type_Object, Object.abort, Object.type_name,
               Object.copy
Int: .word 7, 8, Object, type_Int, Object.abort, Object.type_name,
                  Object.copy
IO: .word 11, 8, Object, type_IO, Object.abort, Object.type_name,
                  Object.copy, IO.out_string, IO.out_int, IO.in_string, IO.in_int
String: .word 10, 8, Object, type_String, Object.abort, Object.type_name,
                  Object.copy, String.length, String.concat, String.substr
```

```
Bool: .word 7, 8, Object, type_Bool, Object.abort, Object.type_name,
                  Object.copy
```

Las instancias del tipo *T* se representan en memoria de la siguiente forma, en la primera palabra el tamaño, en la segunda el tipo *T*, luego sus atributos.

Dado que cada objeto contiene una referencia a la definición de su tipo en memoria se pueden resolver el todas la situaciones en las que se pueda encontrar un objeto.

Funciones

Para el llamado a funciones se siguió el siguiente convenio, el registro fp marca el inicio de los argumentos de la función y *sp* marca el inicio de las variables locales a la función y en *ra* está guardada la dirección del retorno. Es necesario antes de hacer un llamado salvar fp y *ra* ya que estos registros cambian con cada registro de activación de la función. Es responsabilidad de la función reservar sus variables locales así como de limpiarlas. Y es responsabilidad del que llame a las funciones recuperar su fp y su *ra*.

4 Ejecución

Para correr el proyecto ejecutar en la carpeta **/src**:

```
coolc.bat pathfile
```

donde **pathfile** es la dirección del archivo *.cl* que se desea compilar.
solo es posible ejecutar el compilador en *Windows*.