Informe de Entrega CoolCompiler 2020

Jorge Daniel Valle Díaz JORGE.VALLE@ESTUDIANTES.MATCOM.UH.CU
Grupo C412

Leonel Alejandro García López L.GARCIA3@ESTUDIANTES.MATCOM.UH.CU
Grupo C412

Roberto Marti Cedeño R.MARTI@ESTUDIANTES.MATCOM.UH.CU
Grupo C412

Tutor(es):
Msc. Alejandro Piad Morffis, *Facultad de Matemática y Computación, Universidad de La Habana*

Tema: Compilación, Cool Language.

1. Introducción

El siguiente trabajo representa el informe sobre la confección del compilador. Para la confección del mismo se empleó el lenguaje python de programación. El proyecto se dividió en varias etapas: Análisis lexicográfico, sintáctico, semántico, código intermedio y generación de código de máquina. Es importante destacar que se reutilizaron en la medida de lo posible los archivos de clase práctica y los proyectos realizados en el curso previo de la asignatura.

2. Lexer

En la fase de análisis lexicográfico se empleó la biblioteca ply, en especial su módulo lex. Salvo en el caso de las cadenas de caracteres y los comentarios multilínea, el resto de los tokens fueron procesados mediante expresiones regulares. Para el caso de las cadenas de caracteres y los comentarios multi-líneas se emplearon estados especiales exclusivos.

3. Parser

La fase de análisis sintáctico ha sido una de las fases mas controversiales a la hora de la realización del proyecto. Se empleó en una primera fase el parser LR1 tomado de clase práctica y los proyectos previos de la asignatura. La gramática que se definió durante la primera entrega del proyecto contenía ambigüedades, las cuales fueron detectada comprobando las pruebas correspondientes a la parte de semántica.

Finalmente el equipo decisión emplear el módulo *Yacc* de *ply* para definir la gramática y evitar las ambigüedades que se desprendían de la implementación inicial.

La gramática resultante resultó trivial hasta el punto de definir la estructura de las expresiones, para estas últimas con la ayuda de *ply* se estableció una prioridad entre los operadores quedando de la siguiente forma:

Asociatividad	símbolo
derecha	$<-$
derecha	*not*
-	$<, <=, =$
izquierda	$+, -$
izquierda	$*, /$
derecha	*isvoid*
derecha	*compl*
izquierda	@
izquierda	.

Table 1: Operadores ordenados según la precedencia desde el menos prioritario.

Gramática de expresiones:

$$
\begin{aligned}
E &\rightarrow id <- E \mid OP \\
OP &\rightarrow OP < OP \mid OP <= OP + OP = OP \mid \\
&\rightarrow OP - OP \mid OP + OP \mid OP * OP \mid \\
&\rightarrow OP \ / \ OP \mid OP * OP \mid \\
&\rightarrow BS \\
BS &\rightarrow SA \ @ \ type \ . \ \textbf{\textit{FunctionCall}} \mid \\
&\rightarrow SA \\
SA &\rightarrow (\ E \ ) \mid \\
&\rightarrow SA \ . \ \textbf{\textit{FunctionCall}} \mid \textbf{\textit{FunctionCall}} \mid \\
&\rightarrow not \ OP \mid isvoid \ OP \mid compl \ OP \mid \\
&\rightarrow let \ \textbf{\textit{LetVariableDeclarations}} \ in \ E \mid \\
&\rightarrow case \ E \ of \ \textbf{\textit{CaseActions}} \ esac \mid \\
&\rightarrow if \ E \ then \ E \ else \ E \ fi \mid \\
&\rightarrow while \ E \ loop \ E \ pool \mid
\end{aligned}
$$

$\rightarrow A$

$A \rightarrow int \mid string \mid bool \mid id \mid$

$\rightarrow new\ type \mid \{\ \textbf{\textit{ExpressionsList}}\ \}$

4. Análisis Semántico

La fase de análisis semántico se compuso por 3 recorridos del árbol de sintaxis abstracta derivado de la fase de análisis sintáctico que siguen el patrón visitor.

Recolector de tipos: Primer recorrido del ast, en el cual se conforman los tipos nativos y los definidos en el archivo de código a procesar. En este mismo recorrido también se detectan los problemas relacionados con la herencia cíclica.

Constructor de tipos: Segundo recorrido del ast, en el cual se construyen los tipos, se definen sus métodos y atributos.

Verificador de tipos: Tercer y ultimo recorrido del ast, en el cual se verifica la estructura de cada uno de los nodos del ast, este recorrido es el que mas abarca las reglas semánticas del lenguaje Cool.

5. Código intermedio y código de máquina

El equipo decidió realizar una representación intermedia del lenguaje Cool antes de pasar a la generación de código de máquina. Para ello se definieron 2 nuevos recorridos a árboles de sintaxis abstracta, uno para llevar de Cool a CIL y otro para llevar CIL a MIPS.

5.1 Código intermedio

Del AST obtenido en las fases anteriores, en un cuarto recorrido se realizó una transformación hacia un AST de CIL que facilitara que el comportamiento obtenido en COOL sea más factible a un representación en MIPS.

Es también en esta fase donde se da por fin una implementación real a los tipos y funciones básicas definidas por el lenguaje. Estas hacen uso de nodos especiales del AST que solo son instanciadas para su uso desde COOL a través de las funciones básicas. Ejemplo son los métodos para la entrada y salida de datos brindada por la clase IO . Con estas estructuras definidas en el AST de CIL , durante la traducción a MIPS , ellas serán generadas automáticamente.

Aquí existió una vez más apoyo en los elementos de clase practica, al contar con un transpliador base para definir todas la convenciones tomadas, asi como contener los métodos necesarios para dicha conversión. En esta fase surgió la necesidad de resolver un conjunto de especificaciones como son:

- **Relacionado a la herencia** Mantenemos como tipos de CIL como el conjunto de atributos de los respectivos en Cool pero con los de sus antecesores en la jerarquía de tipos(los métodos y la tabla virtual queda para resolverse en MIPS)

- **De los tipos por valor** El problema de los tipos por valor Int y Bool, y de los Strings como casos especiales, al ser usados como un objeto por referencia. Para esto se definió una representación en memorias para los casos en que esto ocurriera. Todos estos tipos cuando fuera necesario tratarlos como objetos por referencias, para por ejemplo acceder a las funciones heredadas de la clase Object o las propiamente definidas por el tipo String , serían encapsulados en instancias similares a los otros tipos, la cual contaría con un atributo value en el cual se almacenaría el valor real de los mismos.

- **Inicialización de atributos** Una de las especificaciones que se tuvo en cuenta para la generación de CIL fue la inicialización de los atributos, tanto los heredados como los propios de la clase. Cuando se crea una instancia de una clase se deben inicializar todos sus atributos con su expresión inicial, si tienen alguna; en otro caso se usa su expresión por defecto. Con el objetivo de lograr esto se creó para cada tipo un constructor, cuyo cuerpo consiste en un bloque, dándole un valor a cada uno de sus atributos. Este es llamado cada vez que se instancia un tipo.

- **Para las estructuras *CaseOf*** de Cool realizamos un ordenamiento (dicho ordenamiento fue realizado de acuerdo a la profundidad de los tipos en la jerarquías, desde el más específico, hasta el más cercano a *Object*) de los *case actions* para realizar un *matching* no iterativo y a partir del primer tipo presente en la jerarquía del objeto resultante de la expresión del case. Luego esta lista que llamamos **CaseActionExpressions** se extiende de manera ordenada para añadir los tipos más específicos de los que se encontraban originalmente que a su vez como expresión a realizar sería la mismo que el ancestro del que extendió, quedando una lista con bloques de case donde antes solo una opción (conservando el orden anterior pero solo por bloques), de forma tal que en tiempo de ejecución el primer *matching* fuese el correcto a realizar.

5.2 Código de máquina

Último recorrido, en este caso del árbol de sintaxis de representación intermedia. Mediante el cual se genera el código a ejecutar en el microprocesador con arquitectura MIPS. Es en este recorrido donde se crean los datos y la representación en memoria que sustentan el sitema de tipos de Cool.

6. Representación y trabajo con la memoria

El equipo se decidió por una arquitectura en memoria que tiene por un lado la representación física del objeto con sus atributos y por otro una tabla de métodos virtuales global, la cual contiene todas las referencias a todas las implementaciones concretas de los métodos de las clases.

6.1 Representación en memoria

A continuación en la tabla 2 presentamos en esquema seguido en memoria de una clase de Cool bajo la solución propuesta.

El identificador de la clase se emplea además de su función básica para la comparación entre clases. El tamaño en memoria de la clase se emplea tanto para la copia como para el tamaño en bytes a reservar. La referencia al nombre de la clase representa la etiqueta correspondiente al nombre de la clase que representa la instancia en memoria y se emplea principalmente para los llamados al método typename. El desplazamiento representa la posición de la etiqueta correspondiente al primer método de la tabla virtual que corresponde a la clase que se tiene en cuestión. Finalizando, se encuentran, si existen, los atributos de la clase.

Identificador de la clase
Tamaño en memoria de la clase
Referencia al nombre de la clase
Desplazamiento en la tabla virtual
Atributo 1
Atributo 2
...
Atributo n

Table 2: Distribución en memoria de una clase de Cool.

6.2 Tabla de métodos virtuales

Para dar solución al problema del ligamiento dinámico que resulta de la herencia se ideó la construcción de una tabla de métodos virtuales global. La tabla contiene todos los métodos correspondientes a todas las clases que se generan a partir de un programa en Cool. Una posible tabla virtual de un programa en Cool tendría una estructura como la siguiente:

Método 1 de la clase 1
Método 2 de la clase 1
Método 1 de la clase 2
Método 1 de la clase 3
...
Método p de la clase n

Table 3: Distribución en memoria de la tabla de métodos virtuales.

La principal idea detrás de la tabla radica en la imposibilidad de tener toda la información de tipos en tiempo de compilación, el ligamiento de cada clase con su implementación de un método particular se deja para tiempo de ejecución.

Cada clase se construye siguiendo el mismo patrón, de forma tal de que en el recorrido de generación de código se conocen todos los métodos y todos los atributos de cada clase y lo más importante: Dadas dos clases A y B, si A desciende de B o B de A, los atributos y métodos de A y B comunes aparecen en el mismo orden tanto en la tabla como en su espacio en memoria de cada instancia.

Esta convención permitió que si una clase T tiene un tipo estático P en tiempo de compilación, pero realmente en ejecución tiene un tipo dinámico J, y J desciende de T, cualquier implementación de los métodos de T en J comparte el mismo orden en la tabla virtual. Por lo que conociendo el tipo estático y llamando al método correspondiente en la tabla virtual al tipo T, se esta llamando realmente al método de T que J sobrescribió.