Proyecto de Complementos de Compilación

Compilador para el lenguaje COOL

Alejandro Campos, Darian Dominguez

Facultad de Matemática y Computación
Universidad de la Habana
2022

Resumen

Este reporte contiene las principales ideas seguidas en la implementación de un compilador completamente funcional para el lenguaje COOL, el cual es un pequeño lenguaje que mantiene muchas de las características de los lenguajes de programación modernos, incluyendo orientación a objetos, tipado estático y manejo automático de memoria. A lo largo de este informe se explicarán, además, cada una de las etapas o fases seguidas en la implementación del compilador. Por último, daremos, a groso modo, una pequeña ojeada al código con el objetivo de que el lector entienda el funcionamiento de los métodos que consideramos fundamentales para el desarrollo del compilador.

I. Introducción

Un compilador es un programa que traduce código escrito en un lenguaje de programación, llamado fuente, a otro lenguaje. En este tipo de traductor, el lenguaje fuente es generalmente un lenguaje de alto nivel, y el otro, un lenguaje de bajo nivel. Un compilador es uno de los pilares de la programación y de cómo entender la comunicación entre un lenguaje de alto nivel y una máquina. Al poder conocer el funcionamiento de este paso intermedio nos permitirá desarrollar y programar de una forma más precisa en los lenguajes de alto nivel. El proceso de compilación es aquel por el cual se traducen las instrucciones escritas en un determinado lenguaje de programación a lenguaje de máquina, es decir, a algo que la computadora entiende.

En nuestro caso, el lenguaje fuente es COOL. Un programa en COOL son conjuntos de clases. Una clase encapsula las variables y procedimientos de un tipo de datos. Las instancias de una clase son objetos. En COOL se identifican clases y tipos; es decir, cada clase define un tipo. Las clases permiten a los programadores definir nuevos tipos y procedimientos asociados (o métodos) específicos para esos tipos. La herencia permite que los nuevos tipos amplíen el comportamiento de los tipos existentes. COOL es un lenguaje de expresión. La mayoría de las construcciones son expresiones, y cada expresión tiene un valor y un tipo. Este lenguaje es type safe, es decir, se

garantiza que los procedimientos se aplicarán a los datos del tipo correcto. Si bien el tipeo estático impone una fuerte disciplina en la programación, garantiza que no puedan surgir errores de tipo en la ejecución de los programas en COOL.

Una vez se tiene una idea acerca del funcionamiento de un compilador y del lenguaje COOL, podemos decir que el objetivo de este informe es presentar la solución al proceso de compilación del lenguaje COOL a lenguaje ensamblador, implementado en el lenguaje de pogramación python, cuyo código se encuentra en este repositorio de Github.

II. Gramática y AST de COOL

La gramática de este lenguaje responde al manual de COOL [1]. Se encuentra definida en `src/utils/COOL_Grammar.py` y su estructura se presenta a continuación:

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{class-list} \\
\textit{class-list} &\rightarrow \textit{def-class ;} \\
&\rightarrow \textit{def\_class ; class-list} \\
\textit{def\_class} &\rightarrow \textit{class type \{ feature-list \}} \\
&\rightarrow \textit{class type inherits type \{ feature-list \}} \\
\textit{feature-list} &\rightarrow \textit{def-attr ; feature-list} \\
&\rightarrow \textit{def-meth ; feature-list} \\
&\rightarrow \epsilon \\
\textit{def-attr} &\rightarrow \textit{id : type} \\
&\rightarrow \textit{id : type <- expr} \\
\textit{def-meth} &\rightarrow \textit{id (param-list) : type \{ expr \}} \\
\textit{param-list} &\rightarrow \textit{non-empty-param-list} \\
&\rightarrow \epsilon \\
\textit{non-empty-param-list} &\rightarrow \textit{param} \\
&\rightarrow \textit{param , non-empty-param-list} \\
\textit{param} &\rightarrow \textit{id : type} \\
\textit{expr} &\rightarrow \textit{id <- expr} \\
&\rightarrow \textit{while expr loop expr pool} \\
&\rightarrow \textit{\{ expr-list \}} \\
\textit{expr-list} &\rightarrow \textit{expr ;} \\
&\rightarrow \textit{expr ; expr-list} \\
\textit{expr} &\rightarrow \textit{let id-list in expr}
\end{aligned}
$$

$$
\begin{aligned}
\textit{id-list} &\to \textit{id : type} \\
&\to \textit{id : type <- expr} \\
&\to \textit{id : type, id-list} \\
&\to \textit{id : type <- expr , id-list} \\
\textit{expr} &\to \textit{case expr of case-list esac} \\
\textit{case-list} &\to \textit{id : type <- expr ;} \\
&\to \textit{id : type <- expr ; case-list} \\
\textit{expr} &\to \textit{not expr} \\
&\to \textit{comp = expr} \\
&\to \textit{comp} \\
\textit{comp} &\to \textit{comp} < \textit{artih} \\
&\to \textit{comp} \leq \textit{arith} \\
&\to \textit{arith} \\
\textit{arith} &\to \textit{arith + term} \\
&\to \textit{arith - term} \\
&\to \textit{term} \\
\textit{term} &\to \textit{term * factor} \\
&\to \textit{term / factor} \\
&\to \textit{factor} \\
\textit{factor} &\to \textit{isvoid factor} \\
&\to \sim \textit{factor} \\
&\to \textit{atom} \\
\textit{atom} &\to \textit{true} \\
&\to \textit{false} \\
&\to \textit{string} \\
&\to \textit{num} \\
&\to \textit{id} \\
&\to \textit{new type} \\
&\to \textit{if expr then expr else expr fi} \\
&\to \textit{(expr)} \\
&\to \textit{dispatch} \\
\textit{dispatch} &\to \textit{atom . id (arg-list)} \\
&\to \textit{id (arg-list)} \\
&\to \textit{atom @ type . id (arg-list)} \\
\textit{arg-list} &\to \textit{non-empty-arg-list} \\
&\to \epsilon \\
\textit{non-empty-arg-list} &\to \textit{expr} \\
&\to \textit{expr , non-empty-arg-list}
\end{aligned}
$$

El árbol de sintáxis abstracta de COOL se encuentra definido en `src/utils/ast/AST_Nodes.py`. Se definió teniendo en cuenta la gramática que mostramos anteriormente. Esta gramática es atributada y cada producción contiene como computar cada uno de los atributos que la conforman que, en la mayoría de los casos, son argumentos que se pasan como parámetros para la creación de los nodos del ast de nuestro lenguaje.

III. Análisis lexicográfico.

El análisis lexicográfico es la primera fase del proceso de compilación, es donde transformamos la entrada del usuario en tokens válidos del lenguaje. Esta fase se encuentra implementada en `src/utils/COOL_Lexer.py`.

El lexer en nuestro proyecto se apoya en el módulo **re** de python para expresiones regulares. La clase `COOL_Lexer` contiene las expresiones regulares que conforman el lenguaje COOL, esta clase hereda de `Lexer`, que implementa el método `tokenize`, el cual recibe un texto como entrada y devuelve los tokens del lenguaje. También se detectan los errores lexicográficos que pueden existir y, en caso de que hayan, se reportan y se detiene la ejecución del programa. Los errores que se detectan en esta etapa son token inválido, y los errores relacionados con los comentarios y la definición de cadenas de texto, es decir, variables de tipo String. Estos últimos se analizan de forma independiente al módulo **re** de pyhton para clasificar cada uno de los errores que pueden surgir.

IV. Análisis sintáctico

El proceso de parsing básicamente de lo que se encarga es en convertir cadenas de texto escritas en el lenguaje origen a una estructura arbórea que captura la semántica del programa. Nuestro parser consiste en analizar la secuencia de tokens calculada en el lexer, y produce un árbol de derivación.

La implementación del parser se enuentra en `src/utils/parser`.

Primeramente tenemos la clase `parser`, que contiene las características comunes a todos los parsers, en esta clase es donde se calculan los conjuntos firsts y follows, y contiene los métodos que deben implementar los parsers que heredan de esta. El parser utilizado es un parser ShiftReduce, la clase `ShiftReduceParser` devuelve el árbol de derivación de la cadena de texto analizada y las operaciones (shift o reduce) que, posteriormente, se utilizan para generar el ast correspondiente. Además esta clase es la encargada de devolver los errores sintácticos o de parser, en caso de que ocurra algún conflicto. La clase `LR1_Parser` implementa los métodos que calculan la tabla de parsing y su respectivo autómata. Finalmente, el parser utilizado fue el LALR 1, que es un parser LR 1, solo que reduce los estados del autómata antes mencionado.

En `COOL_Parser` se encuentran los valores precalculados de las tablas action y goto, que se guardan para acelerar la ejecución del programa, ya que de otra forma demora un poco en calcularlas desde 0. De esta forma `ShiftReduceParser` ejecuta el proceso de parsing con estas tablas ya calculadas. Si se detecta algún error se reporta y se detiene la ejecución del algoritmo.

V. Análisis Semántico

Llegados a esta fase, no existen errores lexicográficos ni de parsing, por lo que el texto está sintácticamente correcto y podemos pasar al chequeo de tipos sin problemas. Podemos encontrar la implementación de esta fase en `src/utils/semantic_check`.

Para verificar la consistencia de tipos se realizan tres recorridos por el ast. El primero de estos recorridos, `TypeCollector` añade al contexto todos los tipos declarados en el lenguaje, no sin antes añadir los tipos predeterminados de COOL con sus respectivos métodos. Verifica las redefiniciones de clases.

Luego, antes de pasar al cuerpo de los métodos y después de recolectar todos los tipos, se hace necesario otro recorrido que verifique declaración de atributos, parámetros, valores de retorno y otras referencias a tipos, y esto precisamnte es lo que hace `TypeBuilder`.

El tercer y último recorrido, `TypeChecker`, verifica que el programa haga un uso correcto de los tipos definidos, según las especificaciones del manual de COOL. En este recorrido también se chequea la herencia circular, la redefinición de métodos y que el programa contenga una clase Main con un método main sin parámetros.

Al principio de cada recorrido aparecen los errores e incosistencias que se detectan en cada uno. Los recorridos del ast fueron implementados con ayuda del patrón visitor.

VI. Generación de código

Para la generación de código intermedio de COOL a MIPS vamos a diseñar un lenguaje de máquina con capacidades orientadas a objetos. Este lenguaje nos va a permitir generar código de MIPS de forma más sencilla, ya que el salto directamente desde COOL a MIPS es demasiado complejo. Este lenguaje se denomina CIL, 3-address object-oriented.

En esta fase no hay incosistencia de tipos, es decir el código de COOL está lexicográfica, sintáctica y semánticamente correcto, listo para la ejecución. Esta fase se encuentra implementada en `src/utils/code_generation`

I. CIL

Como ya se dijo, primeramente, pasaremos de COOL para un lenguaje intermedio, CIL. El ast de este lenguaje se implementó según la definición brindada en [2] Capítulo 7, donde, además de las funciones que ahí se describen, se añaden otras para realizar la ejecución de las funciones predefinidas en COOL, y que son necesarias para el retorno de valores de métodos de este lenguaje.

Con ayuda del patrón visitor se recorre el ast de COOL y, primeramente, se añaden todas las funciones y tipos built-in, es decir, que están definidas por defecto en COOL. Posteriormente, por cada nodo del ast de COOL se crean las instrucciones correspondientes en CIL de acuerdo a su definición.

Contamos con una clase auxiliar, `BaseCOOLToCIL`, que contiene funciones útiles para el registro de parámetros, funciones, instrucciones, variables locales y otras declaraciones en CIL. Esta clase contiene, además, la definición de los métodos built-in de COOL.

II. MIPS

Después de tener el ast de CIL, el siguiente paso es, finalmente, generar el código ensamblador correspondiente a la entrada de COOL. Para ello, primeramente definimos el ast de MIPS, con ayuda de [3], en el cual cada nodo representa una instrucción válida en MIPS. Siguiendo la misma idea de generación de código anterior, se recorre cada nodo del ast de CIL y se va creando el correspondiente ast válido en MIPS. Una vez más, los recorridos de cada ast se realizan con ayuda del patrón visitor, mediante el cual también, se genera el código ensamblador en la clase `PrintMIPS`, que tranforma cada nodo del ast en el código de MIPS correspondiente.

Los objetos en memoria están ubicados de la siguiente forma: tipo, espacio que ocupa, dirección de tabla de dispatch, atributos y un valor que indica que hay un objeto en esta zona de memoria.

El valor que se encuentra en la sección que guarda el tipo se interpreta, como su nombre lo indica, como el tipo del objeto.

La segunda sección de este segmento se interpreta como el tamaño en palabras del objeto.

La tabla de dispatch es una dirección que indica el inicio del segmento de la memoria donde esta se encuentra. Las secciones de dicho segmento se interpretan como la dirección a cada uno de los métodos del objeto.

Cada una de las subsecciones de la sección de los atributos representa el valor de un atributo del objeto.

La última sección nos dice que esta zona de la memoria corresponde a un objeto.

Los tipos están representados en memoria mediante tres estructuras. Primeramente está definida una dirección a un segmento de memoria que representa el nombre del tipo. Luego se representa una estructura que es utilizada en la creación de los objetos, por lo tanto se representa como se dijo anteriormente. Este segmento de memoria es asignado al objeto una vez que se crea y contiene los valores por defecto de estos. Por último, también se encuentra una tabla de dispatch, y existe una tabla de nombres, donde se puede encontrar el nombre de un tipo específico.

VII. Ejecución

Este proyecto no usa ninguna librería de python fuera de la librería estándar. Por lo que para ejecutarlo solo debe tener instalado python 3 o superior. Desde el directorio del proyecto ejecutar `python3 main.py file` donde file debe ser un archivo ubicado en el mismo directorio del proyecto. El compilador generará como salida un archivo con extensión `.mips` que puede ejecutar con cualquier simulador de spim. Sugerimos el uso de QtSpim.

References

[1] Cool-manual (abrir)

[2] Compilers (abrir)

[3] Spim Manual (abrir)