Reporte de Compilación

Oscar Luis Hernandez Solano
Harold Rosales Hernandez
Carlos Rafael Ortega Lezcano

Grupo - C411

1. Uso del Compilador

Para instalar todas las dependencias del compilador se debe emplear el archivo `requirements.txt`, y la instalación debe ser mediante pip haciendo: `pip install -r requirements.txt`. Para ejecutar el compilador se debe estar en la carpeta `./src` y ejecutar el `cool.sh` recibiendo como entrada la dirección del archivo `.cl` a compilar. En caso que no tenga forma de correr el archivo `.sh`, entonces puede ejecutarse usando el archivo principal del compilador `main.py`, es requerido `Python 3` para correr el compilador: `python3 main.py <path>` donde `<path>` es la dirección del archivo a compilar

2. Estructura del Compilador

El compilador se divide en tres partes fundamentales, análisis lexicográfico y sintáctivo, análisis semántico y generación de código. En este apartado estaremos hablando de la estructura de cada uno y finalmente como se combinan para formar el proceso de compilación. Todas las estructuras que veremos, tanto como el Lexer, Parser y los distintos Visitors implementados heredan de `State`, esta clase define un estado del pipeline que controla la ejecución de cada parte del compilador, se encarga de registrar los errores ocurridos y detener la ejecución para reportarlos, su definición se encuentra en `src/pipeline/`, cada estado define una función `run` que maneja la entrada que puede venir desde otro estado o ser la entrada del pipeline, realiza las operaciones correspondientes y pasa los resultados al próximo estado.

2.1. Análisis Lexicográfico y Sintáctico

Esta parte del compilador se compone por dos procesos, primero se tokeniza la entrada y luego se construye el AST que representa el programa de entrada. Para dividir la entrada en tokens se empleo `ply`, empleando las funcionalidades que brinda este se definio nuestro `Lexer` para los programas escritos en Cool este es `CoolLexer` (algunas característica de su implementación serán abordadas más adelante), este recibe recibe la información desde el `Reader`, el cual lee el archivo especificado en la entrada, generando así una lista de tokens (los tokens definidos para el Lexer se encuentran en `parsing.md`), estos serán la entrada para el proceso de parsing.

Para describir el conjunto de programas que pueden ser escritos en Cool se definió una gramática LR basada en la que se encuentra en [1] (la gramática puede consultarse en `parsing.md`), esta es ambigua, lo cual podemos notar fácilmente si vemos las producciones asociadas a las expresiones de Cool, pero mediante el uso de las características de `ply.yacc` podemos resolver los problemas de ambigüedad y obtener una grámatica más compacta y simple de leer. El proceso de parsing resulta en el AST que describe el programa de Cool que recibimos como entrada.

El AST tiene como nodo base `ASTNode`, a partir de este empezamos a establecer la jerarquía, encontramos los nodos de declaración, el nodo de programa y un extenso conjunto de nodos que corresponden a las expresiones, las cuales son mayoria en Cool (los nodos del AST se definen dentro `./src/cl_ast`, se puede ver la jerarquía del AST en `parsing.md`)

2.2. Análisis Semántico

Luego que tenemos un AST correcto, no hay errores léxicos y sintácticos, podemos pasar a comprobar la si es correcta la semántica del programa, para ello emplearemos el *patrón visitor*, este permite separar los algoritmos de comprobación de semántica de la estructura del AST, ya sea recolectar los tipos presentes o definir las variables en el scope correspondiente, de esta forma podemos separar el proceso semántico en diversas fases más simples y enfocadas algunas en nodos en particulas como es el caso de la recolección de tipos. La defínción base de un visitor en Python se encuentra en `./src/visitors/visitor.py`. Como herramientas auxiliares para conocer la información de los tipos y variables que intervien se definen dos estructuras `Context` y `Scope`, en este último se compone por `ClassScope` y `InnerScope`. Los tipos definidos se representan por `Type`, para cada tipo puede definirse sus atributos y métodos mediante `define_attribute` y `define_method` (la definición de tipo y los tipos por defecto de Cool se encuentran en `./src/semantic/types.py`)

Context: El context maneja todas los tipos que intervienen en el programa, además de los que ya están definidos por defecto para un programa de Cool. Permite que se defina un nuevo **Type** además que podemos consultar los tipos definidos.

Scope: Para la creación de un scope es necesario tener definido ya un context. El scope contiene aquellos atributos y variables que aparecen en el programa. Esta compuesto por:

ClassScope: El scope del programa tendrá un ClassScope para cada clase definida, este contiene para la clase que representa, el valor de `self`, los atributos y métodos de esta, para cada atributo o método se define un nuevo scope anidado que es `InnerScope`

InnerScope: Cada ClassScope contiene tantos InnerScope como definición de métodos o atributos contenga, este contiene las variables definidas y referenciadas en la definición de estos, existen expresiones de como *let* las cuales definen variables por lo tanto un InnerScope tendrá anidados más scopes.

Para guardar información en las estructuras anteriores es necesario realizar diversos recorridos al AST, a continuación se explican por orden de aplicación:

TypeCollector: Este visitor se encarga de recolectar los tipos que se definen en el programa, detecta la existencia de tipos definidos con igual nombre, este pasa al siguiente visitor un context con todos los tipos.

TypeBuilder: Este visitor construye los tipos recolectados y establece las relaciones de herencia entre estos, se compone de dos partes, el **Builder** que se encarga de añadir las definiciones de atributos y métodos a los tipos recolectados y detectar errores asociados, luego el **InheritBuilder** se encarga de establecer los padres de forma adecuada para cada tipo, comprobar la redefinicion de atributos y métodos y detectar herencia cíclica.

VarCollector: Este visitor construye el scope para las expresiones de los atributos y para el cuerpo de las funciones, emplea el scope para definir variables asociada con su tipo, detecta declaraciones repetidas de variables, referencias a variables no declaradas, entre otros.

TypeChecker: Este constituye la fase final del análisis, se encarga de comprobar la correctitud de las expresiones y los tipos que las componen, detecta errores de incompatibilidad de tipos a la ahora de asignar expresiones, errores semánticos en diversas expresiones como `if`, `let`, entre otras, además anota en el context los tipos dinámicos asociados a las distintas expresiones que aparecen en el programa

2.3. Generación de Código

Esta fase del compilador la vamos a analizar en dos etapas distintas: generación de código intermedio y generación de código en código de máquina (MIPS), en la primera procesar el AST, el scope y el context obtenidos de las fases anteriores, para obtener una secuencia de pasos para un IL equivalente al código de un programa de COOL, la otra etapa sería luego de obtener un codigo en IL generar la secuencia de instrucciones a un lenguaje de bajo nivel (MIPS).

Cool $\rightarrow$ **IL**: El modulo correspondiente a este proceso se encuentra `transpilator.py` en la carpeta visitors. Se definió un modelo de IL personalizado que dista un poco de los IL convencionales, pero hacerlo de esta manera nos permitio eliminar algunas estructuras complejas de llevar a bajo nivel como por ejemplo: let o case. Se implementó una jerarquia de nodos de tipo IL similar a la de los anteriormente definida para los nodos del AST, cabe destacar que no se genera un AST de IL como tal sino una secuencia bien defnida del orden de sus operaciones, dicha implementacion esta en el modulo `nodes_il` en la carpeta de `code_generation`. Se definieron tambien modulos auxiliares como `virtual_table.py` y `variable.py`, en los que se procesan los resultado del proceso del chequeo semántico y se acomoda a nuestro IL para hacer mas sencillo el proceso de transpilacion, para lograr dicho proceso se implemento un patron visitor donde se recorren los nodos del AST y se obtiene como resultado la secuencia de nodos de nuestro IL para pasar a la siguiente etapa.

IL $\rightarrow$ **MIPS**: El módulo correspondiente a este proceso se encuentra en `to_mips.py` de la carpeta visitors. Una vez creada la lista de nodos de lenguaje intermedio se genera el código ensamblador correspondiente. Para lograr una mejor organización se reciben dos listas de nodos intermedios: Una correspondiente a la sección `.data` y la otra correspondiente a la sección `.text`. Lo primero que se genera es la sección `.data`, para lo cual se visitan en el orden en que aparecen los nodos de lenguaje intermedio correspondientes a esta sección y se devuelve para cada uno de ellos su pedazo de código ensamblador correspondiente. En particular los nodos a los que se hace referencia son los que tienen que ver con la declaración de strings, con las relaciones de herencia y las tablas de métodos de virtuales de cada clase. Las tablas de métodos virtuales son etiquetas en MIPS seguidas por una lista `.word`, el primero de ellos es una referencia a la dirección de memoria de la sección referente a la herencia de esta clase y los siguientes son referencias a las direcciones de memoria de los métodos de esta clase. Luego se genera el código de los métodos estáticos presentes en el lenguaje COOL, los cuales son: `inherit`, `Object.copy`, `Object.abort`, `IO.out_string`, `IO.in_string`, `IO.out_int`, `IO.in_int`, `String.length`, `String.concat`, `String.substr`, `String.cmp` y `substrexception`. Una vez generada la sección .data se procede a generar la sección .text de igual manera, o sea, recorriendo los nodos intermedios correspondientes a esta sección y generando una también el código ensamblador correspondiente. En esta sección los nodos que aparecen son los referentes a asignaciones, salida y entrada estándar, comentarios, llamados a funciones, herencia, condicionales, saltos, reserva de memoria, operaciones unarias y binarias, carga y declaración de etiquetas, asi como los imprescindibles nodos referentes a insertar y extraer información de la pila.

3. Problemas Técnicos

En esta sección expondremos los aspectos del compilador cuya implementación fue interesante debido a que requirieron un mayor esfuerzo de trabajo para ser resueltos.

3.1. Análisis Lexicográfico y Sintáctico

En esta fase la mayor dificultad fue encontrar una forma de ignorar los comentarios de varias líneas debido que una expresión regular no era capaz de resolverlo, para ello se empleó una caracterísitica presente en `ply`. Un objeto `ply.lexer` permite la definición de estados para el proceso léxico, mediante determinada regla podemos indicar que se desea iniciar este estado, ejecutando entonces otro conjunto de acciones distinto para este, por ello nuestro lexer cuenta además de su estado `MAIN` con otros dos estados: `'comments'` y `'string'`, en el caso del 2do estado se emplea para la captura más cómoda de un string por parte del lexer. Para dar inicio al estado de coments se busca una coincidencia con la expresion "(*", de esta forma se inicia el estado mientras que "*)" lo termina bajo determinadas condiciones. Como es posible tener comentarios anidados y deseamos saber si no falta algún "*)", asociado a este estado calculamos el balance de apertura y cierre de comentarios, de esta forma cuando se detecte un "*)", si dicho balance es mayor a 0, no se abandona el estado, de esta forma podemos detectar un error de este tipo ya que el EOF no debería aparecer en un comentario de múltiples líneas.

3.2. Análisis Semántico

Tipos por defecto de Cool: Para representar los tipos definidos en un programa de Cool se definió **Type**, asi cuando creamos un nuevo tipo en el proceso de análisis semántico este queda registrado en el contex de esta forma para realizar consultas a este desde diversos puntos del código siempre recibiremos la misma instancia proveniente del context. En el caso de los tipos por defecto estos se definen mediante herencia de **Type** por lo tanto es posible en todo momento crear una nueva instancia de estos, por ejemplo para el chequeo semántico realizado en el último visitor en ocasiones resulta más claro y limpio devolver un tipo **IntType** que realizar una petición al context para que nos de la instancia que este tiene, debido a esto a la hora de definir los métodos para las clases **Object**, **String** e **IO** contabamos con instancias que contenían estas definiciones y otras que no. Para resolver esto empleamos el patrón *singleton*, así cada vez que instanciabamos un tipo por defecto nos referiamos a la misma instancia. De entre las diversas implementaciones del patrón *singleton* se seleccionó el uso de metaclases por ser simple de añadir a la estructura que teniamos.

Comprobación de Herencia: Como parte del proceso semántico es necesario comprobar la correctitud de las relaciones de herencia en el programa, de esto se encarga el **TypeBuilder**, en un programa de Cool no es obligatorio definir una clase antes de emplearla en una definición de herencia por tanto un sólo recorrido por el AST no permite establecer del todo las relaciones. Para resolver esta situación se pensaron dos opciones, la primera realizar un orden sobre el árbol de herencia parecido a el orden topológico en un DAG, de esta forma tendríamos organizadas las clases y para resolver la herencia múltiple la búsqueda de ciclos en el grafo. La segunda consiste en realizar dos pasadas al AST, la primera para conformar los tipos, con sus atributos y métodos y luego una segunda pasada para establecer la herencia. Se decidió emplear la segunda opción ya que permitía la detección de una mayor cantidad de errores en esta fase, que de otra forma deberiamos realizarlo en el **TypeChecker**.

3.3. Generación de Código

< Explicación Harold >

References

[1] The Cool Reference Manual, Alex Aiken