Proyecto de Compilación
Informe Final

Autores:

C412_Gabriela Mijenes Carrera

C412_Yunior Tejeda

Introducción

En la computación, el compilador se reconoce como un programa que funciona como traductor de un lenguaje de programación a otro. Para cumlir esta función, el compilador debe analizar un código del lenguaje inicial y sintetizar código en el lenguaje al que traduce. Evidentemente, este proceso es bastante complejo.

En este proyecto, presentamos nuestro propio compilador. Este se encarga de analizar un código dado del lenguaje COOL, y convertirlo en código mips, ejecutable por la máquina. Para cumplir esta finalidad pasamos por varias fases: Análisis Lexicográfico, Análisis Sintáctico, Análisis Semántico y Generación de Código. A lo largo del presente informe estaremos exponiendo detalladamente las ideas seguidas para la concepción de cada una de estas fases.

Cabe resaltar que una herramienta principal para la realización de nuestro proyecto fue ply, librería de python que brinda una implementación completa de un constructor de analizador lexicográfico y sintáctico empleada por nosotros para generar la porción del proyecto encargada de estas funciones.

Análisis Lexicográfico

El análisis lexicográfico en un compilador es el fragmento inicial de análisis del código, donde se reconocen los tokens del lenguaje que componen la cadena a analizar, y se sientan las bases para el posterior análisis sintáctico.

Para esta fase empleamos la clase lex del módulo ply antes mencionado, creando un analizador lexicográfico para el lenguaje COOL que nos permitió reconocer y separar los tokens válidos para nuestro lenguaje, que serán posteriormente analizados en la etapa de análisis semántico.

Para ello primeramente debemos declarar quiénes serán los tokens en nuestro lenguaje, y quiénes las palabras reservadas. Ambos conceptos se diferencian en la manera en la que aparecen en el texto; las palabras reservadas se muestran en nuestro código exactamente igual para cada uso, como *class*, *if*, *fi*, etcétera, mientras que los tokens pueden aparecer con distintos lexemas, por ejemplo un token numérico puede tener en una aparición el lexema 665 y en otro 110.

Es necesario también definir las expresiones regulares que permiten tomar los elementos del texto obtenido como entrada de nuestro compilador y convertirlos en nuestros tokens. Esto último se hace solamente para los tokens definidos anteriormente, pues las palabras reservadas se obtienen tal y como aparecen en el texto.

Luego se definen las expresiones regulares para cada tipo de *token* que está en la lista *tokens*, por ejemplo, un comentario de una línea es r'--.*', o sea '--.' y todo lo que venga delante hasta llegar a un salto de línea (estas expresiones regulares que se definen son las que permiten tomar los elementos del texto de entrada y crear los tipos de *tokens* correspondientes). Así mismo se definen las demás funciones con expresiones regulares de los demás símbolos.

Es importante recalcar que por la forma en que esta implementado *ply*, si dos símbolos o *tokens* tienen expresiones regulares tales que una es prefijo de otra, entonces la función de la que tiene mayor tamaño de expresión regular debe ser definida primero, ya que el texto de entrada se evalúa con las funciones que declaras en el módulo de arriba hacia abajo.

Por la dificultad que conlleva construir las expresiones regulares correspondientes, para reconocer strings y comentarios hemos hecho uso también de las máquinas de estado que facilita *ply*. De esta forma una vez que nuestro analizador reconoce que está dentro de una cadena de caracteres o una porción de texto comentado deja de evaluar el resto de las expresiones regulares para los demás tokens y se centra en reconocer completamente el string o comentario actual.

Además, hemos hecho uso de las capturas y restornos de errores que brinda este módulo para hacer saber al programador de COOL que el texto proporcionado presenta errores lexicográficos.

La salida final de esta fase es la lista de tokens lista, con sus respectivas lineas y columnas para ser analizadas en la siguiente fase.

Análisis Sintáctico

El análisis sintáctico es una parte crucial del compilador. En este momento tomamos los tokens y analizamos que la cadena que se nos fue proporcionada pertenece al lenguaje que estamos analizando o no. El descarte de errores en esta fase permite crear un árbol que representa la lógica ordenada del

programa inicial donde solo faltarán analizar posibles errores ligados al contexto.

En esta fase, hemos hecho uso también de las facilidades de *PLY*, brindadas en *yacc* para analizar la lista de tokens válidos que obtuvimos de la fase anterior y aplicar las producciones correspondientes para verificar que tenemos una cadena perteneciente al lenguaje.

Se hace necesario entonces plantear una gramática que genere el lenguaje para presentarsela a nuestro analizador. A continuación adjuntamos dicha gramática:

```
program -> classList

classList -> classDefinition classList

classList -> classDefinition

classDefinition -> CLASS CLASSID LBRACE classFeatureList RBRACE SEMICOLON

classDefinition -> CLASS CLASSID INHERITS CLASSID LBRACE classFeatureList RBRACE SEMICOLON

empty -> <empty>

classFeatureList -> feature classFeatureList

classFeatureList -> empty

feature -> attributeFeature

feature -> functionFeature

attributeFeature -> ATTRIBUTEID COLON CLASSID SEMICOLON

attributeFeature -> ATTRIBUTEID COLON CLASSID ASSIGNATION expression SEMICOLON

functionFeature -> ATTRIBUTEID LPAREN parameterList RPAREN COLON CLASSID LBRACE expression RBRACE SEMICOLON

functionFeature -> ATTRIBUTEID LPAREN RPAREN COLON CLASSID LBRACE expression RBRACE SEMICOLON

parameterList -> parameter COMMA parameterList

parameterList -> parameter

parameter -> ATTRIBUTEID COLON CLASSID'

expression -> notForm
```

```
expression -> mixedExpression
notForm -> NOT mixedExpression
mixedExpression -> mixedExpression LESSEQUAL arithmeticExpression
mixedExpression -> mixedExpression LESS arithmeticExpression
mixedExpression -> mixedExpression EQUAL expression
mixedExpression -> arithmeticExpression
arithmeticExpression -> arithmeticExpression PLUS term
arithmeticExpression -> arithmeticExpression MINUS term
arithmeticExpression -> term
term -> term TIMES isvoidForm
term -> term DIVIDE isvoidForm
term -> isvoidForm
isvoidForm -> ISVOID expression
isvoidForm -> complementForm
complementForm -> COMPLEMENT expression
complementForm -> programAtom
programAtom -> TRUE
programAtom -> FALSE
programAtom -> STRING
programAtom -> NUMBER
programAtom -> ATTRIBUTEID
programAtom -> LPAREN expression RPAREN
programAtom -> NEW CLASSID
programAtom -> memberCall
memberCall -> ATTRIBUTEID LPAREN RPAREN
memberCall -> ATTRIBUTEID LPAREN argumentList RPAREN
argumentList -> expression
programAtom -> programAtom functionCall
functionCall -> DOT ATTRIBUTEID LPAREN argumentList RPAREN
```

```
functionCall -> DOT ATTRIBUTEID LPAREN RPAREN

functionCall -> DISPATCH CLASSID DOT ATTRIBUTEID LPAREN
argumentList RPAREN

functionCall -> DISPATCH CLASSID DOT ATTRIBUTEID LPAREN RPAREN

programAtom -> ATTRIBUTEID ASSIGNATION expression

programAtom -> CASE expression OF caseBody ESAC

caseBody -> ATTRIBUTEID COLON CLASSID ARROW expression SEMICOLON
caseBody

caseBody -> ATTRIBUTEID COLON CLASSID ARROW expression SEMICOLON

programAtom -> LET letBody IN expression

letBody -> ATTRIBUTEID COLON CLASSID

letBody -> ATTRIBUTEID COLON CLASSID ASSIGNATION expression

letBody -> ATTRIBUTEID COLON CLASSID COMMA letBody

letBody -> ATTRIBUTEID COLON CLASSID ASSIGNATION expression COMMA
letBody

programAtom -> LBRACE expressionList RBRACE

expressionList -> expression SEMICOLON expressionList

expressionList -> expression SEMICOLON

programAtom -> WHILE expression LOOP expression POOL

programAtom -> IF expression THEN expression ELSE expression FI
```

Primeramente, es evidente que nuestros símbolos terminales deben corresponderse con los tokens teclarados en la fase anterior.

Luego, para definir la gramática, creamos funciones que contienen cada una de estas producciones como encabezado, y según estas producciones vamos formando el árbol de sintaxis abstracta, con los nodos definidos en *ast_hierarchy*.

También definimos un capturador y lanzador de errores para esta etapa.

Al finalizar esta fase de nuestro análisis tendremos un ast que será evaluado en el chequeo semántico.

Análisis Sintactico

Una vez construído el AST, tenemos la seguridad de que no se nos han escapado errores en la escritura de los tokens, así como que la cadena analizada es generada por la gramática del lenguaje después de un número finito de derivaciones. Es el momento entonces de realizar el análisis semántico.

En este momento del compilador nos imponemos la misión de reconocer errores que van más allá del poder del parser, pues están intrínsecamente ligados al contexto del programa en cuestión.

Es evidente que para que un programa tenga sentido se deben cumplir las siguientes reglas semánticas:

* Una variable solo puede ser definida una vez en todo el programa.

* Los nombres de variables y funciones no comparten el mismo ámbito (pueden existir una variable y una función llamadas igual).

* No se pueden redefinir las funciones predefinidas.

* Una función puede tener distintas definiciones siempre que tengan distinta cantidad de argumentos (es decir, funciones del mismo nombre pero con cantidad de argumentos distintos se consideran funciones distintas).

* Toda variable y función tiene que haber sido definida antes de ser usada en una expresión (salvo las funciones pre-definidas).

* Todos los argumentos definidos en una misma función tienen que ser diferentes entre sí, aunque pueden ser iguales a variables definidas globalmente o a argumentos definidos en otras funciones.

* En el cuerpo de una función, los nombres de los argumentos ocultan los nombres de variables iguales.

Primeramente, necesitamos guardar toda esa información que define un contexto. Para esto hemos creado context.py donde tenemos la definición de 4 clases que serán las encargadas de almacenar toda la información de un programa.

➔ *class Type()*

Esta clase guarda una lista de atributos y otra de métodos definidos dentro de ella, así como una referencia a su padre dentro de la jerarquía de clases del

programa analizado. En ella además se definen las funciones que regulan la localización y creación de dichas características.

➔ *class Variable()*

La clase variable es bien sencilla, en ella solo se guardan los nombres y tipos de las variables del programa en cuestión.

➔ *class Method()*

Similar a la clase variable, la clase Method guarda la información necesaria para definir

un método, esto es nombre, parámetros y tipo de retorno.

➔ *class Context()*

La clase *Context()* evidentemente es la más compleja de las cuatro, pues ella no solo guarda la información relacionada con un tipo, sino también regula las modificaciones del sistema que representa un programa cualquiera del AST. Desde una instancia de Context nuestro compilador genera los escenarios de vida de los objetos, así como los espacios de acción de las funciones, pues se usa además como una especie de "scope" donde se almacenan las variables que han sido definidas y pueden usarse desde el objeto al que está ligado nuestro contexto.

Dicho de esta manera puede no resultar claro al lector. Ilustraré entonces con un ejemplo.

Una de las funciones que definimos dentro de la clase Context es la de crear un contexto hijo.

Dicho contexto hijo, no solo guardará las información que hasta el momento le fue proporcionada al padre, sino que también almacenará la información que está vinculada solo con él y es invisible a su contexto padre.

Veamos entonces el ejemplo:

```
class A {
        a: Int <- 5
        f( x: String) : String {
                x;
        };
```

Si creamos el contexto de la clase A tenemos como atributos del contexto la lista [a], mientras que si creamos ahora el contexto correspondiente al método f la lsita de atributos guardará también a x.

Pero este contexto no se construye solo, más bien es el resultado de un exhaustivo análisis sobre el AST, donde analizamos la existencia de errores ligados a la semántica del lenguaje y al tipado del mismo, logrando que si al finalizar este análisis tengamos la certeza de que nuestro código es una cadena bien definida del lenguaje y poder pasar entonces a la generación del CIL.

En este proceso analítico-constructivo juega un rol principal el patrón visitor. Este nos permite abstraer el concepto de procesamiento sobre un nodo. Implementamos definiciones particulares de visit() en dependencia del nodo que será analizado, teniendo en cuenta sus particularidades, logrando así un recorrido uniformesobre nuestro AST.

Tenemos entonces tres clases que impelentan esta política, estas son TypeCollectorVisitor, TypeBuilderVisitor y TypeCheckerVisitor que, como su nombre indica, se encargan de recolectar, construir y checkear los tipos definidos en un programa Cool.

El primero implementa su recorrido sobre el nodo Programa, desde donde hace un llamado a cada uno de los nodos Clase, recogiendo los nombres de los tipos definidos, a su vez revisa que estos sean válidos.

Una vez validados los tipos definidos por el programador, pasamos a crearlos con ayuda del builder. Este recorre el AST desde el nodo raíz Programa y va clase por clase definiendo los atributos y métodos de cada una después de validarlos.

Ya creados los tipos, debemos entonces analizar su consistencia en el programa. Esta implementación de visitor es la más abarcadora, pues parte desde la raíz y llega a todos los nodos hojas recogiendo su tipo.

Aquí cabe mencionar los dos atributos que permiten lograr este análisis: Tipo Computado y Tipo Definido.

El tipo definido es el tipo que le asigna el programador a un atributo cuando crea su código, mientras el computado es el que realmente tiene y es el resultado de analizar todas las expresiones y sus tipos de retorno. Para nosotros el Tipo Definido entonces es un String con un nombre que, para ser válido, debe corresponderse a uno de los tipos definidos y el Tipo Computado

es una lista de todas las instancias de la clase tipo que son posibles tipos de la característica en cuestión. Veamos un ejemplo:

i: Int <- 5;

s: String <- "cinco";

a: Object <- if x then i else s;

En el ejemplo anterior para el atributo 'a' su Tipo Definido será "Object" mientras que su Tipo Computado será [<Tipo Int>, <Tipo String>]. Esto quiere decir que mientras hemos declarado la variable 'a' como un objeto los tipos inferidos de ella son Int y String.

Se hace evidente entonces que una asignación será válida si cada uno de los tipos en el Tipo Computados son hijos del Tipo Definido. Este mismo análisis se emplea para los pases de parámetros en los llamados de función.

Generación de Código

Esta es la etapa final del proceso de compilación. Llegamos a esta posición con un código COOL perfectamente válido y debemos entonces generar el correspondiente código MIPS.

En esta etapa seguimos la idea vista en clases en el año anterior, donde se plantea la opción de pasar a CIL y posteriormente a MIPS. Es por esto que dividiremos este fragmento del informe en dos minietapas, una, la generación de CIL y la otra, la generación de MIPS a partir de CIL.

Generación de Código Intermedio

En este momento del compilador traducimos el ast obtenido de yacc y modificado en la etapa de chequeo semántico a un ast de CIL.

Para esto realizamos dos pasadas con el patrón visitor, la primera para reconocer los tipos que han sido declarados y la segunda para finalmente construir el ast de CIL.

En esta etapa nos auxiliamos de las clases definidas VariableInfo, MethodInfo y ClassInfo que guardan la información asociada a variables, métodos y clases, para hacer usada posteriormente en la generación de mips.

Es en esta fase donde se definen los tipos built-in Int, String, Bool e IO y sus métodos asociados. También se resuelve el problema de inicialización de variables mediante la incorporación de un constructor a cada clase.

Generación de Código MIPS

Llegados a este momento de compilación solo quedaría generar el código MIPS. Aquí tomamos nuestro AST de CIL y creamos código MIPS perfectamente ejecutable.

En la resolución de esta etapa cabe resaltar dos aspectos fundamentals, el primero la representación de los tipos, para los cuales se va reservando el espacio de memoria correspondiente, calculado ya en la etapa anterior y la referencia a las etiquetas de sus métodos, logrando la conocida tabla de métodos virtuales. Al crear una instancia de la clase A se tiene en memoria los atributos que le corresponden y una referencia a la tabla del tipo A.

Otro aspecto crucial a tener en cuenta en este momento fue el estado de la pila. Para la ejecución de un método se tomó el convenio de dejar la pila en el mismo estado antes de la ejecución. La función que realiza el llamado es la encargada de salvar los registros y pasar los parámetros por la pila, y al finalizar la llamada, dejar registros en su estado inicial.

Observaciones:

Después de cosntruir nuestro chequeador semántico se hacen necesarias las siguientes observaciones que a pesar de que no representan un error en nuestro código, no concuerdan con la respuesta esperada en los test.

La primera observación que debemos hacer es en la herencia. Para los casos de herencia cíclica nuestro compilador reconoce perfectamente los escenarios presentados pero imprime el error exactamente igual, pues lo sitúa en otra de las clases involucradas en la herencia cíclica. Al correr el código del cuarto caso de herencia proporcionado podemos apreciar que aunque no logramos PASSED, imprimimos el siguiente error: “`SemanticError: Class A, or an ancestor of A, is involved in an inheritance cycle`.”

Además, tenemos que comentar el caso de los *while*, quienes devolverán como tipo computado el tipo computado de la última expresión dentro de su cuerpo, así sucede también con los *blocks.*

Otra particularidad que debemos resaltar se encuentra en el caso de los números de fila y columna del error mostrado en pantalla. En el análisis semántico, cuando encontramos un error, por ejemplo, en el llamado a un método, quizás porque en el pase de parámetros la expresión proporcionada no concuerda con el tipo esperado, el error debería mostrarse en la columna del parámetro en cuestión, mientras que nosotros lo imprimimos en la columna donde comienza el llamado a la función.

En la creación de la gramática que utilizamos en el análisis sintáctico en nuestro compilador cometimos errores, que fueron arrastrados en las siguientes etapas, dificultando grandemente el trabajo en las mismas. A continuación exponemos esta idea con mayor claridad.

Como se conoce un AST de COOL, tiene como raíz una instancia del nodo Programa, del que parten instancias de clases, las que recogen dentro de sí todo el código. Estas clases a su vez tienen definidos dentro atributos y métodos. Estos serán instancias de los nodos *AttributeFeatureNode* y *FunctionFeatureNode*, respectivamente. Además tenemos lo que en nuestro código de recoge dentro de Statements que son los nodos creados a partir de palabras reservadas como *let*(que genera un *LetStatementNode*) o *while* (con el correspondiente *LoopStatementNode*).

Para lograr aterrizar la idea de una manera más comprensible nos centraremos en el ejemplo de *let*.

Dentro de un statement let tenemos las llamadas variables, que se iniciaizan en la primera etapa del *let* y que serán usadas en el in en más complejas expresiones. Estas variables, fueron creadas como nodos del *AST* del tipo *AttributeFeatureNode*, lo que posteriormente creó para nosotros grandes confusiones a la hora de representarlas en tanto en el context como en *CIL* y *MIPS*, y que dieron pasos a estrategias poco correctas como la empleada a la hora de visitar un nodo atributo dentro de nuestro árbol.

Conclusiones

El desarrollo de este proyecto final de la asignatura Complementos de Compilación representó un fuerte reto para nuestro equipo, pues puso a prueba no solo nuestros conocimientos de compilación adquiridos en el pasado curso, sino también los conocimientos prácticos de desarrollo en python y MIPS. Además, nos hizo enfrentarnos a un proyecto a gran escala cin los riesgos que esto conlleva, como la necesidad de lograr sentar unas bases teoricas irrefutables que no nos arrastren a cometer mayors errores en el furuto. Ha sido, indudablemente, una experiencia enriquecedora, a pesar de no haber logrado el objetivo principal que era presentar un compilador totalmente funcional del lenguaje COOL.