Informe del Proyecto de Compilación

Autores: C – 412 Lázaro Jesús Suárez Núñez

C – 412 Marcos Maceo Reyes

Introducción

Este informe presenta la idea general que hay detrás de nuestra implementación del compilador de COOL, la cual está en la carpeta **src** que se encuentra donde esta este documento. Se divide este informe en cuatro secciones que corresponden a las fases de Análisis Lexicográfico, Análisis Sintáctico, Análisis Semántico y Generación de Código. Este documento solo da una visión general de las ideas y procedimientos que tomamos en la construcción del compilador, para un mayor entendimiento se sugiere ir a la vez mirando el código que se encuentra en la carpeta **src**.

Análisis Lexicográfico

Se encuentra en lexer.py.

Para la fase del análisis lexicográfico se utilizó el utilísimo módulo *ply* que contiene la clase *lex* que permite construir de forma muy sencilla un analizador del texto de entrada(código) para separar los diferentes componentes en *tokens* que después serán analizados en la fase de análisis sintáctico.

Los tipos de *tokens* se declaran mediante las estructuras *reserved* y *tokens*. El diccionario *reserved* contiene la lista de las palabras clave del lenguaje (como aparecen en el texto de entrada) con su respectivo tipo de *token* (Ejemplos: 'class': 'CLASS', 'while': 'WHILE', etc.…). La lista *tokens* es una simple lista que contiene los nombres de los restantes tipos de *tokens* como 'NUMBER' o 'DOT'. La diferencia entre los dos es que los tipos de *token* que están en *tokens* pueden tener distintas formas dentro del código, por ejemplo, un *token* de contante numérica puede aparecer en el código como el numero 8749 o puede aparecer como 213; sin embargo, los *tokens* que corresponden a palabras claves siempre se encuentran en el código de la misma

forma, o sea la palabra clave *class* siempre aparecerá así en el código, luego usar el diccionario *reserved* nos permite simplificar el proceso de crear el *token*, pues la forma es la misma, además para reconocer que se ha encontrado una palabra clave solo basta con revisar si la cadena de entrada se encuentra en las llaves de *reserved*.

En esta fase se ignoran caracteres extraños de entrada como '\r' o '\v'.

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.

En este módulo se implementó también, y gracias a facilidades que nos brinda *ply*, unas simples máquinas de estado para clasificar los *strings* ("algo entre comillas") y los comentarios mistilínea, pues las expresiones regulares correspondientes eran un poco complicadas por el tema de los caracteres extraños o por el tema de los comentarios mixtilínea que podían redefinirse dentro de ellos mismos (y había que contar hasta que los símbolos de inicio y fin de este tipo de comentario quedaran balanceados). Estas máquinas de estado permiten que una vez que se inicia alguna se dejan de evaluar las demás funciones de los demás tipos de *tokens*, por ejemplo, cuando se lee un “ se entra en el estado inicial de "leer string" y se leen entonces (y se guardan) todos los caracteres hasta que aparece otro “ que no está antecedido de un \, entonces todo lo que se lee dentro no se evalúa en las funciones de los tipos de *tokens* que no son STRING.

También *ply* nos permite dejar guardado en los *tokens* de salida la información referente a la línea y la columna donde se encuentran en el texto de entrada, lo que es muy necesario después cuando se necesita devolver un error y decir en qué línea y columna esta.

Por ultimo en este módulo se capturan y retornan los errores, que estos ocurren cuando algo de la entrada no se puede machear con las expresiones regulares que definimos previamente, o cuando detectamos símbolos como \0 que constituyen errores.

El único método que se importa en *compiler.py*, que es el modulo que recibe el código y manda a ejecutar todas las fases, es *make_lexer*, en el cual le pasamos el texto de entrada y aquí devolvemos todos los tokens sacados y los errores encontrados.

Análisis Sintáctico

Se encuentra en *parser.py* y *ast.py*.

En la fase de análisis sintáctico se utilizó también el módulo *ply* que contiene también importantes mecanismos para esta parte y además la construcción del Árbol de Sintaxis Abstracta.

Este módulo consiste en una serie de funciones que leen los *tokens* enviados por el *lexer* y realizan las correspondientes producciones dependiendo de la gramática definida.

Primeramente, hay que tener ya conformada la gramática de nuestro lenguaje COOL para definir después las producciones a llevar a cabo. Nuestra gramática quedo de la siguiente forma:

program -> class_list

class_list -> class_definition class_list

| class_definition

class_definition -> CLASS CLASSID LBRACE class_feature_list RBRACE SEMICOLON

| CLASS CLASSID INHERITS CLASSID LBRACE class_feature_list RBRACE SEMICOLON

class_feature_list -> feature class_feature_list

| empty

Feature -> attribute_feature

| function_feature

attribute_feature -> ATTRIBUTEID COLON CLASSID SEMICOLON

| ATTRIBUTEID COLON CLASSID ASSIGNATION expression SEMICOLON

function_feature -> ATTRIBUTEID LPAREN parameters_list RPAREN COLON CLASSID
LBRACE expression RBRACE SEMICOLON

| ATTRIBUTEID LPAREN RPAREN COLON CLASSID LBRACE
expression RBRACE SEMICOLON

parameters_list -> parameter COMMA parameters_list

| parameter

Parameter -> ATTRIBUTEID COLON CLASSID

Expression -> not_form

| mixed_expression

not_form -> NOT mixed_expression

mixed_expression -> mixed_expression LESSEQUAL arithmetic_expression

| mixed_expression LESS arithmetic_expression

| mixed_expression EQUAL expression

| arithmetic_expression

arithmetic_expression -> arithmetic_expression PLUS term

| arithmetic_expression MINUS term

| term

```
term -> term TIMES isvoid_form
        | term DIVIDE isvoid_form
        | isvoid_form

isvoid_form -> ISVOID expression
               | complement_form

complement_form -> COMPLEMENT expression
                   | program_atom

program_atom -> boolean | string | int | id | parentesis | new | member
                | function | assign | case | let | block | while | if

boolean -> TRUE
           | FALSE

string -> STRING

int -> NUMBER
```

id -> ATTRIBUTEID

parentesis -> LPAREN expression RPAREN

new -> NEW CLASSID

assign -> ATTRIBUTEID ASSIGNATION expression

case -> CASE expression OF case_body ESAC

case_body -> ATTRIBUTEID COLON CLASSID ARROW expression SEMICOLON case_body

| ATTRIBUTEID COLON CLASSID ARROW expression SEMICOLON

let -> LET let_body IN expression

let_body -> ATTRIBUTEID COLON CLASSID

| ATTRIBUTEID COLON CLASSID ASSIGNATION expression

| ATTRIBUTEID COLON CLASSID COMMA let_body

| ATTRIBUTEID COLON CLASSID ASSIGNATION expression COMMA let_body

```
block -> LBRACE expression_list RBRACE

expression_list -> expression SEMICOLON expression_list
                 | expression SEMICOLON

while -> WHILE expression LOOP expression POOL

if -> IF expression THEN expression ELSE expression FI

function -> program_atom function_call

function_call -> DOT ATTRIBUTEID LPAREN argument_list RPAREN
               | DOT ATTRIBUTEID LPAREN RPAREN
               | DISPATCH CLASSID DOT ATTRIBUTEID LPAREN argument_list RPAREN
               | DISPATCH CLASSID DOT ATTRIBUTEID LPAREN RPAREN

argument_list -> expression
               | expression COMMA argument_list
```

member -> member_call

member_call -> ATTRIBUTEID LPAREN RPAREN

| ATTRIBUTEID LPAREN argument_list RPAREN

Ver que los símbolos terminales son los tipos de *token* que declaramos en *lexer.py*.

Luego sería usar la clase *yacc* de *ply* para definir esta gramática mediante funciones que contienen estas producciones como encabezado, y según las producciones que se van tomando ir formando el árbol de sintaxis abstracta.

La estructura del árbol de sintaxis abstracta se encuentra en *ast.py*, y consiste en la colección de distintos tipos de nodo. A cada tipo de nodo se le pasa por el constructor la información referente al nodo, o sea, si es un nodo de definición de clase se le deberían pasar el nombre de la clase, el padre si esta hereda, y la lista de atributos y métodos (sus características).

Para seguir con este ejemplo si vamos a la función de las producciones de este no terminal:

class_definition: CLASS CLASSID LBRACE class_feature_list RBRACE SEMICOLON

| CLASS CLASSID INHERITS CLASSID LBRACE class_feature_list RBRACE SEMICOLON

Tenemos que cual delas dos producciones se tomó:

```
if len(p) == 7:
        p[0] = ClassNode(p[2], p[4], None, [GetPosition(p, 2)])
    else:
        p[0] = ClassNode(p[2], p[6], p[4], [GetPosition(p, 4)])
```

Si p (que es la lista de elementos que contiene la producción) tiene tamaño 7 entonces quiere decir que se tomó la producción donde la clase no hereda de nadie (la primera), luego retornar en esta producción un nodo de tipo ClassNode pasándole los elementos correspondientes. Notar que si se le manda un elemento que en la producción es un símbolo no terminal entonces este será el nodo resultado de dicho símbolo no terminal cuando se evalúe, de esta forma se va creando el árbol.

Al final se define también una función para evaluar y retornar los errores que se van encontrando, así como la línea y la columna correspondiente. Ver también que a cada nodo se le pasa la información de la fila y la columna donde se encuentra en el texto para ser utilizado posteriormente en el chequeo semántico.

Análisis Semántico

Se encuentra en *semantic.py*, *graph.py* y *type_defined.py*.

La fase de análisis semántico se realizó sobre el AST que se devuelve en la fase del análisis sintáctico. Aquí se llevaron a cabo los siguientes chequeos (en este orden):

- Chequeo de tipos declarados (método *check_type_declaration*)

- Chequeo de herencia de tipos (método *check_type_inheritance*)

- Chequeo de grafo de herencia (método *check_cyclic_inheritance*)

- Chequeo de atributos y métodos de los tipos (método *check_features*)

- Chequeo de expresiones (método *check_expressions*)

- Chequeo de la existencia de la clase *Main* y método *main* (abajo en *check_semantic*)

En el chequeo de tipos declarados se comprueba que no haya dos tipos con igual nombre y se guardan en un diccionario todos los tipos. También se comprueba que no se haya declarado un tipo con el nombre de un tipo básico (ejemplo: *Int*).

En el chequeo de herencia de tipos se busca que los tipos de los que se heredan sean tipos que están definidos, además de que no se puedan heredar de los tipos básicos *Int*, *Bool* y *String*. También se chequea el caso de que el tipo este heredando de sí mismo.

En el chequeo de grafo de herencia se analiza el grafo formado por los tipos y la herencia de estos en búsqueda de ciclos. Como no existe la herencia múltiple y todos los tipos heredan de la clase *Object* entonces basta con realizar una búsqueda a lo largo (DFS) sobre el grafo empezando en *Object* y comprobando que se lleguen a todos los demás tipos. Aquí se utiliza el modulo *graph.py* para representar el grafo.

En el chequeo de atributos y métodos de los tipos se comprueba que los atributos y los métodos dentro de los tipos estén correctos, o sea, que devuelvan un tipo que esté definido, que no se llamen *self*, y que si se redefine un método de una clase padre este tenga el mismo tipo de retorno y la misma cantidad de parámetros (esta es una restricción de COOL).

En el chequeo de expresiones se analizan todas las expresiones de los atributos y métodos de los tipos, en búsqueda de que no haya conflictos de tipo en las operaciones, valores de retorno, parámetros, entre otras. También se comprueban que se estén llamando a atributos, argumentos o métodos que estén definidos dentro de la clase o en alguna clase padre.

Se tiene bien en cuenta que los tipos de retorno de las expresiones no tiene que ser el mismo tipo que se define en su inicialización o definición, sino que este puede ser de un tipo que hereda del esperado.

En el chequeo de la existencia de la clase *Main* y el método *main* queda explicito lo que se hace.

El código correspondiente al chequeo se encuentra en *semantic.py* y en *graph.py* se encuentra una pequeña clase para la estructura del grafo a la hora de buscar herencia cíclica. En *type_defined.py* se encuentran unas clases que se utilizaron para convertir el AST en algo más compacto y declarar unas funciones para mayor comodidad en la búsqueda de tipos y sus métodos y atributos, estas clases son *CoolType*, *CoolAttribute* y *CoolMethod*, además de que definieron los tipos básicos y sus métodos.

Generación de Código

En la fase de generación de código optamos por generar un código de lenguaje intermedio (CIL) para que, como disminuye un poco el nivel de abstracción de los datos, nos organizara el código en una forma más similar a como se encuentra en MIPS, además de que se pudiera utilizar para generar otro tipo de lenguaje ensamblador.

La generación de CIL se encuentra en *cil_generator.py*. Este módulo utiliza a su vez a los módulos *cil_ast.py* y *string_data_visitor.py*. En *cil_ast* se encuentra la estructura de un AST que

represente el código en CIL, y en *string_data_visitor* se encuentra una implementación de un patrón visitor que se utilizó para buscar cadenas estáticas dentro del AST devuelto por el *parser*.

Primeramente, se genera una lista (TYPES) que contiene nodos de tipo TypeNode, que nos guardara todos los tipos y los nombres de sus métodos y atributos. Esta lista es utilizada en la generación de MIPS para chequear los tipos de los atributos cuando se van a crear instancias de los tipos saber la cantidad de memoria que van a ocupar, entre otras cosas.

Después se genera una lista (DATA) que contendrá los datos del código, que serán puestos en la parte .data del código MIPS. Aquí es donde se utiliza el patrón visitor para recoger las cadenas de caracteres que hay dentro del código y ponerlas en un nodo tipo DataNode. También aquí se agregan después los nombres de las variables locales que se declararían dentro del código que serían usadas como punteros a direcciones en memoria en el código MIPS.

Finalmente se genera una lista (CODE) donde como tal iría el AST con los nodos de instrucciones en CIL. A esta lista se le van agregando nodos del tipo FunctionNode a medida que se van viendo las distintas funciones que se declaran en el código. Lo primero que se hace en esta última parte de generación de CIL es agregar al código las funciones de los tipos básicos, y luego la de los procedimientos de las inicializaciones de atributos y funciones de los tipos, también se agrega al final una función extra *main* que sería la función de entrada en el código MIPS.

Los nodos del AST construido para la generación de CIL representan instrucciones que se ponen de forma similar a MIPS, aunque alguna de esta podría traducirse en más de 10 instrucciones en código MIPS. La introducción de este código intermedio nos permite convertir las expresiones en COOL en algo menos abstracto que sería más fácil hacer que convertir directamente a MIPS, y que a la vez nos haría el trabajo más fácil a la hora de como tal generar el MIPS.

Se utiliza la clase Node_Result para en su atributo *node* ir llevando la serie de instrucciones hasta el momento y en *result* la variable local donde se encuentra el resultado hasta el momento. Por ejemplo, tomemos la expresión que representa una condicional en COOL:

if <expr> then <expr> else <expr> fi

para convertirla al CIL hacemos lo siguiente:

```
def convert_conditional(expression):
    predicate = convert_expression(expression.evalExpr)
    if_expr = convert_expression(expression.ifExpr)
    else_expr = convert_expression(expression.elseExpr)
    label_if = get_label()
    label_else = get_label()
    result = get_local()
    node = predicate.node + [
            IfGotoNode(predicate.result.id, label_if)] + else_expr.node + [
            MovNode(result.id, else_expr.result.id),
            GotoNode(label_else),
            LabelNode(label_if)] + if_expr.node + [
```

```
        MovNode(result.id, if_expr.result.id), LabelNode(label_else)]


    return Node_Result(node, result)
```

En la primera línea del método mandamos a convertir la expresión a evaluar en la instrucción condicional en CIL y el resultado queda en *predicate*, *predicate* sería entonces una variable de tipo Node_Result donde en *node* tiene la lista de las instrucciones que se realizan para calcular el valor de la expresión y en *result* contiene la variable local donde quedaría el resultado (la variable local es de tipo LocalNode y consiste en un identificador que es el que se utilizaría como tal en el código MIPS). Luego se procede igual para las expresiones dentro del cuerpo del *then* y del *else*, y se guardan en *if_expr* y *else_expr* respectivamente. Después se definen, o crean, dos variables donde estarán las etiquetas y la variable local que se van a utilizar para hacer los saltos y devolver el resultado de la instrucción.

Finalmente se meten en una lista los nodos que corresponden a las instrucciones que se ejecutan en total en esta instrucción. Aquí primero vendrían las instrucciones para calcular la expresión a evaluar, luego se vería un nodo IfGotoNode que se le pasan el nombre de la variable local donde se encuentra el resultado de la evaluación y el nombre de la etiqueta a donde se saltaría si el resultado fuera *True* (o 1 que es lo mismo). Luego se ponen las instrucciones del cuerpo del *else* pues si no se salta en la instrucción anterior quiere decir que la expresión a evaluar era *False*. Después se agrega un nodo MovNode que lo que haría fuera meter el resultado de la expresión del cuerpo del *else* en la variable local de retorno de la instrucción condicional. Luego un salto como es lógico al final ya que a continuación se encuentra la etiqueta y las instrucciones correspondientes a si el predicado era True, estas instrucciones igual que el *else*, agregan las

instrucciones para llegar al resultado del cuerpo del *then* y las pondrían en el resultado en la variable local de retorno de la instrucción condicional.

Finalmente se pone la etiqueta de fin para si se había ido por el *else*. Se retorna un Node_Result donde la lista de instrucciones es la que se construyó anteriormente y el resultado que estará en la variable *result* de retorno de la instrucción condicional.

La generación de CIL entonces nos manda para el módulo de generación de MIPS la lista de los tipos, los datos, el código, entre otros datos como la cantidad de parámetros y variables locales máximos en una función.

El generador de MIPS a partir del CIL se encuentra en *mips_generator.py*. Aquí se reciben las anteriores listas desde el CIL y se genera el código MIPS en correspondencia. Es más sencillo aquí pasar de nodo CIL a MIPS que si lo hiciéramos directamente del AST que se devuelve en el análisis sintáctico.

La función *generate_mips* de este módulo devuelve un texto que corresponde al código MIPS que devuelve la evaluación de los datos enviados por el CIL.

En el código MIPS se ponen las funciones de los tipos con el nombre (nombre_del_tipo)_(nombre_de_función) para que no se repitan en el código ningún nombre de función por la redefinición de estas por algún tipo hijo. Los parámetros se pasan por la pila, almacenando en esta los valores de variables locales correspondientes. Se guarda en memoria la lista de los tipos que hay con sus nombres para poder hacer instanciado dinámico de tipos con las direcciones de memoria correspondientes, pues a veces una expresión devuelve un tipo hijo del tipo esperado, y aquí entonces se pide en memoria en dependencia del tipo devuelto, que esto a veces solo es conocido en ejecución.

Las variables locales son datos que se declaran en la sección .data, que consisten en dos *word* seguidos, en el cual el primero contiene la dirección en memoria del tipo de dato correspondiente

(en esta dirección en memoria comienza una cadena con el nombre del tipo) y en el segundo se guarda el valor de la variable. Si la variable es de tipo *Int* o un *Bool* su valor es el entero correspondiente (0 o 1 en caso de *Bool*). Si la variable es de tipo *String* entonces el valor será una dirección en memoria, en la cual primero hay un *word* que dice la longitud de la cadena y a continuación esta la cadena (finalizada por el carácter 0). Finalmente, si la variable es de algún tipo declarado, su valor es una dirección en memoria en la cual va a haber primero un *word* con el tamaño en bytes del objeto, y seguido estarán los valores de los atributos correspondientes, que estos estarán puestos de la misma forma que una variable local, o sea, primero un *word* con la dirección del tipo y después un *word* con su valor.

También se trató de minimizar la cantidad de variables locales definidas en la sección .data y de parámetros, pues cuando se generaba el CIL se contó la cantidad máxima de parámetros y variables locales en una función, por lo que al cambiar de función en ejecución se reutilizan los mismos datos, pero los anteriores valores se guardan en la pila por si son necesarios reutilizarlos.

La forma en que se procedió para convertir un nodo de CIL en código MIPS fue creando por casi cada nodo de CIL una función en MIPS, que lo único que se le pasaba eran las direcciones de variables locales o datos involucrados con la instrucción CIL correspondiente. Como cada instrucción en CIL solo tiene a lo sumo 4 elementos entonces las direcciones de variables locales o datos respectivas fueron pasadas por los registros de salvado (los *$sx*) para minimizar el uso de la pila.

Aparte de las funciones correspondientes de los nodos de CIL, también se crearon algunas funciones para procesos o acciones que se realizaban frecuentemente, como por ejemplo calcular la longitud de una cadena o copiar *n* bytes de una dirección a otra.