Complementos de Compilación

Cool-Compiler

Universidad de La Habana

Facultad de Matemática y Computación

Laura Brito Guerrero C-412

Sheyla Cruz Castro C-412

Ariel A. Huerta Martín C-412

Índice

1. Uso del Compilador:

Para utilizar el compilador es necesario instalar todas las dependencias utilizadas por este como son los paquetes *ply* para la generación del lexer y el parser, y para la ejecución de los tests, *pytest* y *pytest-ordering*, todo esto se logra ejecutando el fichero *requirements.txt* de la forma:

```
pip install -r requirements.txt
```

Para ejecutar el compilador es necesario correr el archivo *cool.sh* ubicado en */src/* dando como entrada la dirección del archivo a compilar de la siguiente forma:

```
cd src/
./cool.sh '../tests/codegen/arith.cl'
```

El archivo principal del compilador es *main.py*, módulo que contiene toda la lógica del compilador, si desea ejecutarlo debe pasar como argumento el *path* del fichero *.cl* con código fuente de COOL que se desea compilar, de la siguiente forma:

```
python3 main.py '../tests/codegen/arith.cl'
```

Un archivo en el mismo path del código fuente será creado, con el mismo nombre, pero con extensión *.mips* que puede ser ejecutado con *spim*.

Para ejecutar las pruebas localmente, debe tener instalado *Python 3.7*, *pip* y *make* que normalmente viene incluido en Linux, para hacerlo ejecute los siguientes comandos:

```
cd src/
make clean
make test
```

2. Arquitectura del Compilador:

El proceso de compilación se desarrolla en tres partes fundamentales: análisis lexicográfico y sintáctico, análisis semántico y generación de código. La estructura de nuestro compilador de manera general se divide en varios módulos independientes ubicados en src correspondientes a cada una de las fases por las que atraviesa el proceso de compilación. Para el desarrollo del compilador se utilizó PLY, una herramienta de construcción de compiladores lex/yacc implementada en Python que incluye soporte al parser LALR(1), validación de entrada y reporte de errores.

A continuación estaremos hablando de cada una de estas fases y sus correspondientes estructuras.

2.1. Análisis Lexicográfico y Sintáctico:

Esta fase se divide a su vez en dos subfases: análisis lexicográfico que consiste en la descomposición del programa fuente en componentes léxicos y análisis sintáctico que se encarga de agrupar estos componentes léxicos en frases gramaticales. De manera más general esta fase se encarga de comprobar la validez de la entrada inicial, tokenizarla y generar el árbol de derivación correspondiente al programa de entrada o Árbol de Sintaxis Abstracta(AST), incluye desde la definición de la gramática hasta la construcción del lexer y el parser. En esta fase como ya se comentó anteriormente, se emplearon las herramientas de construcción de compiladores lex y yacc que brinda PLY. Dentro de src/nodes se encuentra la jerarquía de nodos del AST mientras que el lexer y parser del proyecto se encuentran en src/lexer/lexer.py y src/parser/parser.py respectivamente.

2.1.1. Análisis Lexicográfico:

Aquí se procesa el texto de entrada de izquierda a derecha y se agrupa en componentes léxicos conocidos como tokens, que no son más que secuencias de caracteres con un significado específico, a partir de una colección de reglas en forma de expresiones regulares, este proceso se conoce por el nombre de tokenización. Para la construcción del lexer se utiliza el módulo lex.py que ofrece PLY, este requiere la definición de una variable *tokens* donde se definen los distintos tipos de tokens, además se especifican una serie de expresiones regulares para determinar cada secuencia de caracteres a que tipo de token corresponden, esto mediante la definición por cada token de una función de nombre *t_tipo*, donde "tipo" es el tipo de token específico al que corresponde, con la descripción de la expresión regular en cuestión. Durante este proceso toda información innecesaria, dígase espacios en blanco, comentarios y demás, serán eliminados del programa fuente. Se contemplan además la fila y la columna de cada token para el manejo de errores.

2.1.2. Análisis Sintáctico:

Esta fase conocida como proceso de parsing consiste en analizar la correctitud sintáctica de una secuencia de tokens y producir el árbol de derivación correspondiente. Para esto se utilizó el módulo yacc.py que implementa la componente de parsing de PLY teniendo como salida un árbol de sintaxis abstracta representativo del programa de entrada. Yacc procesa la gramática y genera un parser que utiliza el algoritmo de shift-reduce LALR(1) y tiene como requisito la especificación de la sintaxis en términos de una gramática BNF(notación de Backus-Naur). Se definen entonces la gramática y las acciones para cada producción. Cada regla gramatical tiene una función cuyo nombre empieza con $p\_$, el cuerpo de esa función contiene el código que realiza la acción de esa producción. El parámetro p de la función contiene los resultados de las acciones que se realizaron para parsear el lado derecho de la producción, o sea se puede acceder a estos indexando en p. La parte derecha e izquierda de la producción se separan mediante los dos puntos(:), en $p[1]$ se encuentra el primer símbolo de la parte derecha y en $p[0]$ el resultado de la acción actual. Finalmente se obtiene el árbol de sintaxis abstracta definiendo en cada producción cada uno de los nodos del árbol.

2.2. Análisis Semántico:

En lugar de explicar cada parte de la implementación, se dará una conceptualización de la lógica que rige el proyecto. El objetivo del mismo es inferir los tipos dado un programa determinado.
Añadir que para el **análisis semántico** se definen clases para declarar **atributos**, **variables**, **errores**, **métodos**, **tipos**, **contextos** y **scopes**. Los cuales son un pilar para lo que se explicará a continuación. Luego de este punto, el programa atraviesa por tres filtros, donde cada uno tiene una papel importante en la deducción de los tipos, estos son:

1. **TypeCollector**: Como es el primer filtro, se declaran en el **contexto** todos los tipos que se definen, primeramente se agregan los tipos definidos en el lenguaje: la clase **Object**, **String**, **IO**, **Int**, **Bool**; todos con sus respectivos métodos. Luego se guardan todos los nombres de las clases que se definen en el programa.

2. **TypeBuilder**: Como en el contexto ya se guardaron todos los tipos que pueden existir en el programa, se analizan las funciones y los atributos. También se analiza la herencia de las clases, ya que no permite que ninguna herede de las clases por defecto del programa excepto por **IO**, y por **Object**, la última mencionada es la clase raíz ya que explícita o implícitamente todas las clases heredan de ella.

3. **TypeChecker**: Siendo este el último filtro, es el que termina el trabajo sin dejar que ningún detalle se escape. El mismo se encarga de que el tipo que se asigne a una variable, a una clase, en fin a alguna entidad, sea $\leq$ que el tipo de la entidad.
   Aquí se introduce un nuevo concepto:
   Conformidad: Sea A, C y P tipos:

   a) $A \leq A \; \forall$ tipo de A,
   b) si C hereda de P entonces $C \leq P$,
   c) si $A \leq C$ y $C \leq P$ entonces $A \leq P$.

   También se encarga de chequear los tipos de los parámetros de las funciones, con los tipos de entrada a la hora de llamar a la función.

2.2.1. Implementación

El código correspondiente se encuentra en src/semantic/semantic_analyzer.py. Aquí se realiza el análisis de los filtros explicados en 2.2.
En src/semantic/tools aparecen las declaraciones de las clases que representan a los **atributos**, **variables**, **errores**, **métodos**, **tipos**, **contextos** y **scopes**. Por último se encuentra en src/semantic/visitor las tres clases que representan a los filtros por las cuales se analizan los tokens de entrada: **TypeCollector**, **TypeBuilder** y **TypeChecker**.

2.3. Generación de Código:

Esta fase se divide en dos partes, primeramente se traduce el código de COOL al lenguaje intermedio CIL, que permite generar código de forma más sencilla, para ello se recorre el AST de COOL

y se construye un AST de CIL. La segunda parte consiste en recorrer el AST de CIL y generar a partir de este, el código de MIPS. Para las dos partes se emplea el patrón visitor para recorrer los AST de COOL y CIL.

2.3.1. CIL:

La definición de los nodos del AST de CIL se encuentra en *src/nodes/cil_ast_nodes* y *src/nodes/cil_expr_nodes*. La generación de código intermedio de COOL a MIPS fue implementada en *src/code_generation/cool_to_cil.py*, en este archivo se encuentra un clase que es la que emplea el patrón visitor para recorrer el AST de COOL y formar el AST de CIL. Esta clase también contiene métodos y atributos usados para la construcción del AST de CIL como por ejemplo las variables de instancia types, code y data que son usadas para almacenar los nodos correspondientes a las tres secciones .TYPES, .CODE y .DATA de CIL, o la variable de instancia current_function que tiene mucha utilidad para construir los FunctionNode con sus parámetros, variables e instrucciones. Cuando se visita el primer nodo ProgramNode, se construyen todos los nodos relacionados a los built-in types de COOL con sus respectivos atributos y métodos.

2.3.2. MIPS:

La generación de código de CIL a MIPS fue implementada en *src/code_generation/cil_to_mips.py*, en este archivo se encuentra un clase que es la que emplea el patrón visitor para recorrer el AST de CIL y generar el código de MIPS.

Para el llamado de funciones con los sus respectivos parámetros se empleó a pila, en la cual se colocan en orden inverso al que fueron declarados. La pila también fue usada para el almacenamiento de las variables locales de las funciones, para ello se salva espacio en la pila para guardarlos, y se lleva un diccionario con los respectivos offset de cada uno para su fácil acceso. También se lleva un diccionario para los offset de los métods y otro para los offset de los atributos, de forma que resulte más fácil el manejo de la pila. Utilizamos siempre el registro $a1$ para almacenar el valor de retorno de las funciones luego de su ejecución. Cada función se encarga de sacar de la pila tanto los parámetros como las variables locales.

La organización de los objetos en memoria la hicimos de la siguiente forma, dado su dirección de memoria:

- en el offset 0 se encuentra un número único que identifica la clase,
- en el offset 4 se encuentra un puntero a una dirección del segmento de datos donde está almacenado el nombre de la clase,
- en el offset 8 está un número que representa el tamaño de la clase,
- en el offset 12 se encuentra un puntero a una dirección donde están las funciones declaradas por la clase o heredadas y,
- a partir del offset 16 se encuentran los valores de los atributos declarados o heredados de la clase.

El número único asignado en el offset 0 representa el orden en el que fue visitado por un recorrido DFS por él árbol que representa la jerarquía de clases con nodo inicial Object. Dado que el orden de los atributos es importante, la dirección de estos está definida por el orden en el que fueron declarados, iniciando por los pertenecientes al ancestro más lejano hasta llegar a los de la propia clase. Con respecto a las funciones heredadas que pueden ser redefinidas, se sustituye en el offset de la función heredada por la dirección de su propia definición. El valor void se representa como una dirección estática.

3. Problemas Técnicos:

3.1. Análisis Lexicográfico y Sintáctico:

En esta fase del compilador, el mayor reto que se nos presentó y que cabe resaltar, fue la forma de ignorar los comentarios de múltiples líneas y el manejo de strings. Para esto nos apoyamos en el concepto de estado que contempla *ply*, este permite la definición de nuevos estados con reglas definidas para ellos, lo que nos permite cambiar de un estado específico a otro y regirnos por las reglas definidas para este. Por tanto definimos dos estados, uno para los comentarios y otro para los strings, que se detectan a partir de los caracteres $\backslash *$ y “. Para manejar los comentarior aninados controlamos el balance de $\backslash *$ encontrados.

3.2. Análisis Semántico:

En el momento de controlar las excepciones lógicas de la gramática **Cool**, sobre todo en la parte de la referencia cíclica en la herencia de las clases definidas. Dicho problema se resuelve definiendo un orden topológico de todas las clases definidas.
También en la asignación del tipo **AUTO_TYPE**, generalmente en el retorno de los métodos recursivos y en la definición de los **lets**, **blocks** y **cases**. De la misma forma se resuelve teniendo control de los tipos que se van guardando y dependiendo de la expresión que se esté analizando se asigna el tipo deseado, solamente fue cuestión de volver a revisar las cuestiones que se iban escapando.

3.3. Generación de Código:

En esta fase, un problema que surgió fue como manejar la expresión case, ya dado un tipo T, se debe determinar en tiempo de ejecución, entre un conjunto de tipos T_i, $1 < i \leq n$, cuál es el menor tipo T_i con el que T se conforma. Para la solución de dicho problema, se realizó un recorrido DFS por él árbol que representa la jerarquía de clases con nodo inicial Object, a cada node se le asignó un tag único, el cuál representa el orden en el que fue visitado. A cada tipo se le asigna además un max_tag que se corresponde con el mayor tag del nodo al cual puede llegar con un DFS a partir de él, luego podemos decir que un tipo T se conforma con un tipo T si $tag\_T \leq tag\_T \leq max\_tag\_T$. Entonces, para saber cual es el tipo buscado, recorremos una lista de tipos T_i ordenados de forma descendiente buscando aquel que primero cumpla la condición.

Referencias

[1] The Cool Reference Manual, Alex Aiken.

[2] `https://ply.readthedocs.io/en/latest/index.html`

[3] Appendix, Assemblers, Linkers and the SPIM Simulator, James R. Larus.