Proyecto de Compilación

Autores:

- Arlenis Ching Suárez (a.ching@estudiantes.matcom.uh.cu)
- Lauren Jiménez Martín (l.jimenez@estudiantes.matcom.uh.cu)
- Marcel Ernesto Sánchez Aguilar (m.aguilar@estudiantes.matcom.uh.cu)

Arquitectura del compilador de Cool

- Analizador Lexicográfico
- Analizador Sintáctico
- Chequeo Semántico
- Generación de Código Intermedio
- Generación de Código MIPS

En la primera fase, el Analizador Lexicográfico (Lexer) recibe la cadena de texto a compilar y la transforma en un array de tokens del lenguaje Cool. Luego el Analizador Sintáctico lo transforma en un Árbol de Sintaxis Abstracta (AST) según la gramática atributada definida para el lenguaje. Sobre este árbol se verifica la correctitud del código desde el punto de vista semántico y posteriormente se logra el paso hacia un AST de un lenguaje intermedio entre Cool y MIPS (CIL), el cual facilitará la última fase, la generación del código MIPS.

Gramática

La gramatica diseñada es una grámatica LALR, es decir, es ambigua, pero sus conflictos son solucionados por un parser LALR. Con su diseño se garantiza una correcta precedencia operacional.

Analizador Lexicográfico

Para la implementación de esta primera fase, se utilizaron las facilidades que brinda la biblioteca *Python Lex-Yac* (PLY). En este punto se definieron los tokens de Cool mediante expresiones regulares, además de otras construcciones del lenguaje como los comentarios. Con esta herramienta se asegura que todos los tokens reconocidos sean los definidos en Cool.

Analizador Sintáctico

En esta segunda fase se utiliza la salida del lexer y la gramática atributada para lograr conformar el AST de Cool y a su vez verificar que la organización de los

tokens reconocidos cumpla los requisitos del lenguaje. Fue utilizada la biblioteca *Python Lex-Yac* (PLY).

Chequeo Semántico

Usando el patrón *visitor*, se hacen tres recorridos por el árbol.

El primer recorrido se realiza para recolectar las definiciones de tipos en el código. Nos apoyamos en la clase *TypeNode* para representar la jerarquía de tipos, la cual nos será útil en algunas verificaciones semánticas tales como la no existencia de ciclos en la herencia y la definición única de cada clase.

En el segundo recorrido se construye el contexto de atributos y métodos de las clases. En este se chequea la no sobrecarga de métodos y la no redefinición de atributos. En los tipos *built-in,* se verifica para *IO* que no se sobrescriban sus métodos, mientras que para los restantes que no se herede de estos.

El último recorrido es para verificar la consistencia de tipos en los nodos del AST. Aquí es necesario conocer los ámbitos de cada clase y métodos, utilizamos la clase *Context* para su representación.

Generación de Código Intermedio

Realizando un par de pasadas con el patrón *visitor* sobre el AST de Cool se construirá el AST de CIL correspondiente.

Con el primer recorrido se recolectan todos los tipos en la jerarquía de clases (incluyendo los *build-in* con todas sus funciones y atributos definidos) para poder tener todas sus componentes defindas. Se quiere lograr una representación en la cual, para toda clase A que herede de una clase B, esta contenga primeros los atributos y metodos de su padre, y luego los propios.

En el segundo se declara el conjunto de instruccciones de CIL en cada método que define su estructura.

Se le hicieron modificaciones a la jerarquía de clases de CIL vista el curso pasado, al añadirle clases para la lectura y escritura por la consola, para el manejo de errores, para la busqueda de la clase heredada, entre otras.

Para una mayor simplicidad del problema, se hizo uso de las clases auxiliares *VariableInfo*, *MethodInfo* y *ClassInfo* que recogen la información asociada a las variables, métodos y clases respectivamente. En ellas se almacena información relevante que luego se utilizará en MIPS para facilitar ciertos procesos: como en el caso de las variables locales y los parámetros de un método cual será su posición dentro del *stack,* los nuevos nombres que recibirán ciertas componentes para evitar errores, la posición de los métodos y atributos respecto a su clase (notar que si B hereda de A entonces todo atributo o función herededa ocuparan la misma posición en A y B, incluso si en B redefinimos un método de A).

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.

Para dar solución al conflicto en aquellos lugares donde se necesita un tipo por referencia y llega uno por valor y viseversa, se hizo necesaria la implementación del concepto de *boxing* y *unboxing*. Por ejemplo, cuando se realiza *isvoid,* en todas las asignaciones y en el *typeof*.

Los errores en tiempo de ejecución son menejados en esta etapa.

Generación de Código MIPS

En esta última fase se hace un recorrido por un AST de CIL y se genera el código MIPS correspondiente para cada nodo. Es necesario tener en cuenta varios aspectos para esta etapa.

- Representación de tipos. Para ello se reserva un espacio en memoria en la cual se almacena el nombre del tipo y la referencia a las etiquetas de sus métodos. Esto es lo que se conoce como tabla de métodos virtuales. Al crear una instancia, en la memoria reservada para ella se tienen sus atributos y una referencia a la tabla de su tipo.
- 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.

En esta etapa se realizó la implementación de las funciones *Length*, *Concat* y *Substr*, pertenecientes a la clase *String* de Cool.

Pendiente

- *SELF_TYPE*: Este aspecto del compilador era opcional y no se tuvo en cuenta. Las funciones de los tipos *built-in* que devuelven *SELF_TYPE* fueron ajustadas para su correcto funcionamiento, excepto *copy* que no fue implemetada.

Limitaciones

- La gramática diseñada el uso de las expresiones en Cool de la forma $not\ x$ donde x es una expresión compuesta, se debe hacer de la forma $not(x)$. Análogo con ~.

Requerimientos

Para la correcta ejecución del nuestro compilador de COOL, se requiere:

- Python3.5 o superior.
- PLY3.11.

Forma de ejecución:

- python3 main.py <filename.cl>