Compilación

Cool Compiler 2023

Karlos Alejandro Alfonso Rodríguez
Laura Victoria Riera Pérez
Kevin Talavera Díaz

Cuarto año. Ciencias de la Computación.
Facultad de Matemática y Computación, Universidad de La Habana, Cuba

1 de diciembre de 2023

Repositorio del proyecto

https://github.com/computer-science-crows/cool-compiler-2023

I. Uso del compilador

Para utilizar el compilador, navegue al directorio /src y ejecute el siguiente comando: *python -m coolcmp -h*, lo que mostrará la ayuda con las opciones disponibles. Para compilar el archivo *code.cl*, simplemente ejecute: *python -m coolcmp code.cl*. Este proceso generará un archivo *code.mips*, que puede ser ejecutado en el simulador SPIM mediante el comando: *spim -f code.mips*.

```
Usage: python -m coolcmp [-h] [--tab_width TAB_WIDTH] file_path

COOL Compiler 2023.

Positional arguments:
file_path Path to COOL source file to compile

Optional arguments:
-h, --help Show this help message and exit
--tab_width TAB_WIDTH Tab width for converting tabs to spaces, deault is
4
```

Figura 1: *Uso del compilador*

I. Módulos

El proyecto está estructurado en varios módulos ubicados en la carpeta /src:

```
/src
├── main.py: Interfaz para la configuración y ejecución del compilador.
├── cool_compiler.py: Punto de entrada del compilador que coordina las fases del proceso.
├── /lexic_analysis/lexer.py: Implementación del analizador léxico.
├── /sintactic_analysis/parser.py: Implementación del analizador sintáctico.
├── /sintactic_analysis/parser_tab.py: Archivo generado automáticamente por ply.
├── /semantic_analysis/ast_.py: Clases y estructuras del Árbol de Sintaxis Abstracta (AST).
├── /semantic_analysis/semantic.py: Lógica para el análisis semántico.
├── /type_checker/type_checker.py: Implementación del type checker para la coherencia de tipos.
├── /cil_generation/cil_generator.py: Generación de código intermedio en representación de bajo nivel.
├── /mips_generation/mips_generator.py: Generación de código de máquina MIPS.
├── /utils/constants.py: Definición de constantes utilizadas en el compilador.
├── /utils/errors.py: Clases de errores personalizadas para informar sobre problemas.
└── /utils/environment.py: Implementación de la estructura de entorno utilizada en varias fases.
```

Figura 2: *Estructura del proyecto*

II. Fases

El proceso de compilación consta de varias fases esenciales que transforman el código fuente en COOL en un programa ejecutable. Comienza con el análisis léxico, donde se identifican los componentes básicos del código. Luego, el análisis sintáctico verifica la estructura gramatical correcta. A continuación, el análisis semántico evalúa el significado del código. El chequeo de tipos garantiza la coherencia entre los tipos en el programa. La generación de código intermedio crea una representación más abstracta del programa. Finalmente, la generación de código de máquinas traduce el programa a instrucciones específicas del hardware para su ejecución.

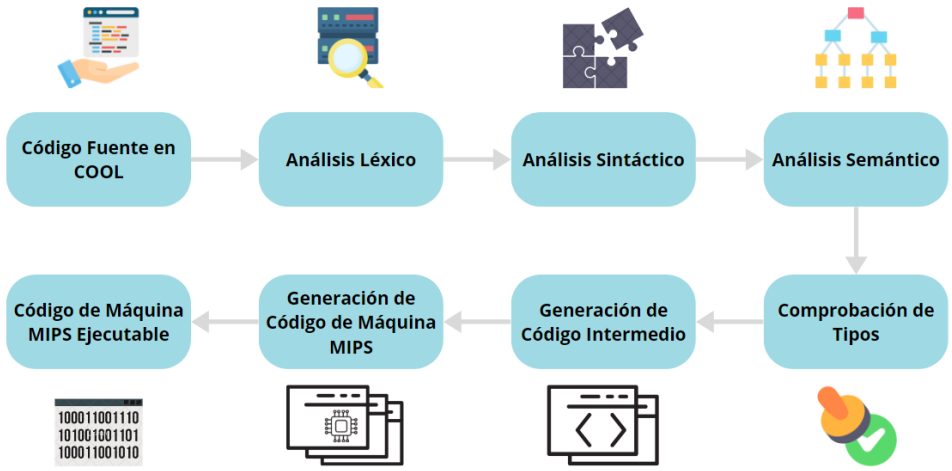

Figura 3: *Flujo del programa*

ii.1. Análisis Léxico

ii.2. Análisis Sintáctico

ii.3. Análisis Semántico

ii.4. Comprobación de Tipos

El chequeo de tipos se realiza utilizando el patrón de diseño Visitor, y puede ser encontrado en type_checker.py.

- **Conformance Test:** Previo al análisis de tipos, es esencial verificar si un nodo U *conforma* con otro nodo V. En este contexto, la conformidad indica que el nodo U puede ser manejado como si perteneciera al tipo del nodo V.
  Durante esta validación, se asegura que las expresiones y operaciones del programa sigan las reglas de tipos establecidas por COOL, posibilitando el polimorfismo al permitir que objetos de clases derivadas se utilicen en contextos donde se espera una clase base. La conformidad de tipos se establece en función del árbol de herencia, donde un tipo se considera conforme a otro si comparten el mismo tipo o están en su cadena de herencia.
  Para llevar a cabo esta tarea, se emplea un Depth-First Search (DFS) que calcula dos valores para cada nodo x: $td(x)$, el tiempo de descubrimiento de x, es decir, el primer momento en que el DFS llega a X; y $tf(x)$, el tiempo de finalización de x, es decir, el último momento en que el DFS está en x. La relación de conformidad entre dos nodos U y V se verifica mediante las propiedades de td y tf. U conforma con V si $td(V) \leq td(U) \leq tf(V)$.
- **Lowest Common Ancestor:** La operación *join* en COOL se utiliza para determinar el tipo estático común más bajo de dos tipos dados. Esta operación es necesaria cuando se realiza una operación de "dispatch.º llamada a método en COOL para garantizar que se esté invocando el método correcto según la jerarquía de tipos.
  Para encontrar el tipo estático común más bajo, se utiliza el concepto de Lowest Common Ancestor (LCA) o ancestro común más bajo de dos nodos en el AST. La implementación sigue un proceso básico: si el nodo U está más lejos de la raíz que el nodo V, se establece que LCA(U, V) = LCA(padre(U), V). Este proceso se repite hasta que U y V son iguales, momento en el cual el LCA es U.
- **Inicialización del Orden:** Durante la traversa DFS del AST, se inicializan los valores td y tf para cada nodo. Además, se verifican las compatibilidades de los métodos heredados y se precálcula el tipo estático de los formales de los métodos antes de la visita.
- **Resolución de Problemas Técnicos:**
  Conformidad de Métodos Inherentes: Se asegura que los métodos heredados sean compatibles con los métodos en la clase hija.
  Ciclos en la Herencia: Se verifica la presencia de ciclos durante la construcción del árbol de herencia.
  Consideraciones Ambiente de Tipos: Se mantiene un entorno actual (current_environment) durante el recorrido del AST para realizar comprobaciones de tipos.
  Ámbito de Clases: Se realiza un seguimiento del ambiente de clases actual (current_class) para lidiar con herencias y definiciones de clases.

ii.5. Generación de Código Intermedio

La **representación de clases** en el proceso de generación de código MIPS implica la transformación de cada clase en una función de inicialización, denominada “FuncInit”. Esta función

tiene la responsabilidad de crear instancias de la clase correspondiente cuando se llama en código MIPS. Las clases contienen una serie de **atributos**, entre ellos los atributos reservados como "_type_info", que almacena la referencia a la dirección de memoria que contiene los datos del tipo de la instancia, y "_size_info", que indica la cantidad de bytes que la instancia ocupará en memoria. Existen atributos específicos para clases particulares, como "_int_literal"para la clase Ïnt", que representa el entero asociado a la instancia, y "_string_length", "_string_literal"para la clase "String", que almacenan la longitud y el contenido de la cadena, respectivamente. Por último, "_bool_literal.en la clase "Boolçontiene un valor booleano (1 para "true.o 0 para "false"). Además de los atributos reservados, las clases pueden tener atributos adicionales declarados por el programador durante la definición de la clase.

Un aspecto clave es el almacenamiento de tiempos de descubrimiento (td(C)) y finalización (tf(C)) para cada clase. Estos tiempos son cruciales para la resolución de dispatches, es decir, llamadas a métodos, ya que determinan la ubicación de la implementación del método en la jerarquía de clases. También son fundamentales en expresiones Case, donde los tiempos de descubrimiento y finalización facilitan la resolución de las diversas ramas del caso.

La **representación de métodos** en el proceso de generación de código MIPS implica la transformación de cada método, denotado como "f", en una "función"(Function). Cuando un método "f.está definido en una clase Ç", se registran datos esenciales asociados a esa función, incluyendo los tiempos de descubrimiento (td(C)), finalización (tf(C)), y el nivel (level) de la clase Ç.en el árbol de herencia. Estos datos son cruciales para la gestión de dispatches y otras resoluciones durante la ejecución del programa.

La función resultante encapsula la lógica y comportamiento del método, y su identificador está compuesto por el nombre de la clase que la contiene seguido por el nombre del método, proporcionando así una forma única de referenciar la función. La información sobre tiempos y niveles se utiliza especialmente en dispatches, donde se requiere una resolución dinámica y eficiente de la llamada a métodos. El nivel indica la distancia desde la raíz del árbol de herencia, lo que facilita la determinación de la clase correcta a la que pertenece el método, mientras que los tiempos de descubrimiento y finalización son fundamentales para garantizar el orden adecuado de ejecución y la coherencia en la resolución de expresiones.

La **asignación de posiciones a variables en el stack** es un paso crítico en el proceso de generación de código MIPS. Cada variable, ya sea una variable de clase (atributo) o una variable local (parámetro formal, variable Let o Case), debe tener una posición predefinida en el stack para garantizar su acceso eficiente durante la ejecución del programa.

Este proceso se lleva a cabo mediante la creación y gestión de un entorno, que es una estructura que mantiene un mapeo entre el nombre de cada variable y su posición correspondiente en el stack. El entorno también tiene un padre, que representa el bloque en el que se encuentra actualmente. Al llegar a bloques como Let, CaseBranch, FuncInit o Function, se crea un nuevo entorno. En la definición de una variable, ya sea en el entorno actual o en bloques superiores, se asigna la posición adecuada y se incrementa el contador de posiciones.

Durante el proceso, se registran las referencias a las variables, indicando si se refieren a atributos de clase o a variables locales. Cuando se sale de un bloque, el entorno se reinicia al padre correspondiente, y se ajustan las posiciones según el estado previo. Este enfoque garantiza una gestión coherente y eficiente de las variables en el stack, facilitando su acceso y manipulación durante la ejecución del programa.

ii.6. Generación de Código de Máquina