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.

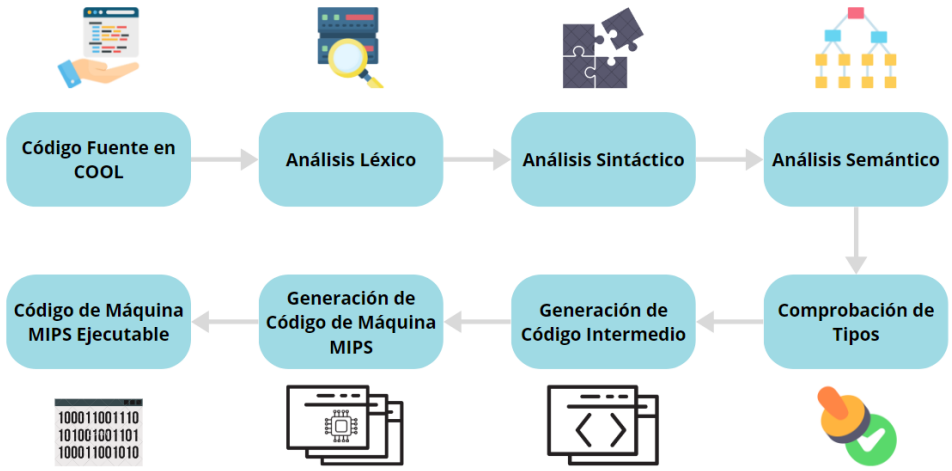

Figura 3: *Flujo del programa*

ii.1. Análisis Léxico

En el proceso de análisis léxico, se ha aprovechado la versatilidad de la biblioteca PLY (Python Lex-Yacc) para simplificar la definición de tokens y reglas en la implementación del lexer. La primera tarea implica la categorización de tokens, abarcando identificadores, palabras clave, literales y operadores, que son las unidades léxicas fundamentales en el lenguaje COOL. Estas categorías se definen mediante expresiones regulares, estableciendo reglas específicas para cada tipo de token. Es destacable la atención especial dedicada a las reglas diseñadas para manejar comentarios y cadenas de caracteres.

En la lógica de creación de tokens, se emplea la funcionalidad de la pila de estados de PLY, permitiendo la transición fluida entre contextos al detectar el inicio de comentarios o cadenas de caracteres. Este enfoque adaptativo posibilita la aplicación de reglas específicas según la situación y la restauración al estado anterior al completar cada sección particular. Además, el código del lexer está equipado con un sólido mecanismo de manejo de errores. Detecta y registra posibles problemas léxicos, como comentarios no cerrados o cadenas de caracteres no terminadas, proporcionando una capa adicional de robustez al proceso de análisis.

ii.2. Análisis Sintáctico

La fase de análisis sintáctico se ha llevado a cabo mediante el uso de la herramienta PLY (Python Lex-Yacc). Inicialmente, se convirtió la gramática de COOL de su forma extendida en la notación de Backus-Naur a la forma estándar. Posteriormente, se generó un esqueleto del archivo ‘parser.py‘ que contiene cada regla gramatical como un método dentro de la clase ‘Parser‘. Este enfoque es necesario para la correcta integración con PLY.

Cada método del parser se completa con la parte derecha de la regla correspondiente. Se crearon clases para modelar cada símbolo de la gramática, representando así los nodos del Árbol de Sintaxis Abstracta (AST). Estas clases se han diseñado para incluir información sobre la línea y columna en la que se encuentra el símbolo correspondiente en el código fuente.

Las reglas del parser abordan distintos aspectos de la gramática de COOL, desde la definición de programas y clases hasta la manipulación de expresiones y operaciones. Por ejemplo, se definen reglas para la construcción de clases, métodos, atributos, expresiones condicionales, bucles, y más.

Además, se han establecido prioridades y asociatividades entre operadores para resolver posibles conflictos durante el análisis sintáctico. Se ha incorporado también manejo de errores en caso de que la gramática no sea seguida adecuadamente, generando excepciones que informan sobre la presencia y ubicación de posibles errores sintácticos.

ii.3. Análisis Semántico

En el contexto del análisis semántico en el compilador de COOL, se inicia verificando las reglas semánticas especificadas en el manual del lenguaje. Un componente esencial de esta fase es la construcción del árbol de herencia, donde cada nodo representa una clase y las relaciones de herencia se establecen a través de nodos hijos. Este árbol se inicia desde la clase base .ºbject", y se implementa un algoritmo de búsqueda en profundidad (DFS) para detectar posibles ciclos, asegurando así que el árbol sea acíclico, como exige COOL.

Una tarea crítica es la gestión de SELF_TYPE. Se establecen las siguientes relaciones de conformance: SELF_TYPEX $\leq$ SELF_TYPEX (reflexividad) y SELF_TYPEC $\leq$ P si C $\leq$ P (transitividad). Para manejar SELF_TYPE, se agrega un nodo SELF_TYPEC como hijo de cada nodo C en el árbol de herencia, garantizando la conformidad para casos reflexivos y transitivos.

La construcción del árbol de herencia también implica la detección de posibles errores, como la redefinición de clases y la validación de herencias desde clases no definidas. Se garantiza que las clases estén ordenadas en el árbol según sus líneas en el código fuente, contribuyendo a un análisis semántico coherente.

Además, se verifica la consistencia de métodos y atributos en las clases. Se implementa un proceso de comprobación para evitar la redefinición no permitida de métodos y atributos. También se realiza una verificación específica para asegurar la existencia de la clase "Mainz la presencia del método "main.[en] esta clase, sin recibir argumentos.

ii.4. Comprobación de Tipos

El chequeo de tipos se realiza utilizando el patrón de diseño Visitor para proporcionar una estructura modular y extensible. Este patrón permite separar las operaciones específicas de los nodos del Árbol de Sintaxis Abstracta (AST) en clases independientes conocidas como visitantes. Esta separación de responsabilidades facilita la extensión del compilador sin modificar directamente las clases de los nodos del AST, proporcionando una organización clara y simplificando el mantenimiento del código.

En el contexto del Chequeo de Tipos, cada clase de nodo en el AST implementa un método accept que acepta un visitante como argumento. Cada visitante tiene métodos específicos para realizar operaciones particulares en los nodos del AST. Por ejemplo, para el Conformance Test, se utiliza un visitante que realiza un recorrido DFS asignando tiempos de descubrimiento y finalización, permitiendo una verificación eficiente de la conformidad de tipos. Además, otro visitante se encarga de responder preguntas sobre el Lowest Common Ancestor, esencial para operaciones como "join". Este enfoque modular y basado en el patrón Visitor contribuye a la flexibilidad y mantenibilidad del compilador COOL.

Previo al análisis de tipos, es esencial realizar la **verificación de conformidad de tipos**. En este contexto, se dice que un nodo U *conforma* con otro nodo V si 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 $td(V) \leq td(U) \leq tf(V)$.

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.

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.^{es}tá 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

En la etapa final de la compilación de COOL, la generación de código MIPS se lleva a cabo con una implementación meticulosa de las funcionalidades detalladas en el manual.

Para la gestión de registros, se adoptó una estrategia sencilla pero efectiva. Un registro se dedica exclusivamente a contener la dirección de memoria del objeto self actual. Los registros temporales se emplean para operaciones temporales, y se reservan registros específicos para "pasarz regresarreferencias de objetos. Cuando es necesario manipular más de un argumento o valor, se recurre al uso de la pila (stack) de MIPS.

La resolución de dispatches, que implica la llamada a métodos en tiempo de ejecución, se lleva a cabo eficientemente. Se almacena información esencial sobre las funciones con nombre f en el Data Segment, organizadas por nivel en orden descendente. Resolver un dispatch implica buscar el ancestro más profundo de la clase actual (C) que contenga una función con nombre f. La complejidad de esta operación es O(#funciones distintas).

El proceso de resolución de expresiones Case sigue un enfoque similar al de los dispatches. Las ramas se reorganizan según el ancestro más profundo, y la resolución se reduce a buscar el primer ancestro que sea un caso válido para la expresión.

En relación con los Strings y el Data Segment, se almacenan literales de Int, String y Bool en este segmento. Para cumplir con el alineamiento requerido por MIPS, se añaden ceros a los literales de String, asegurando que su tamaño sea un múltiplo de 4.