**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

## I. Uso del compilador

### 1. Módulos

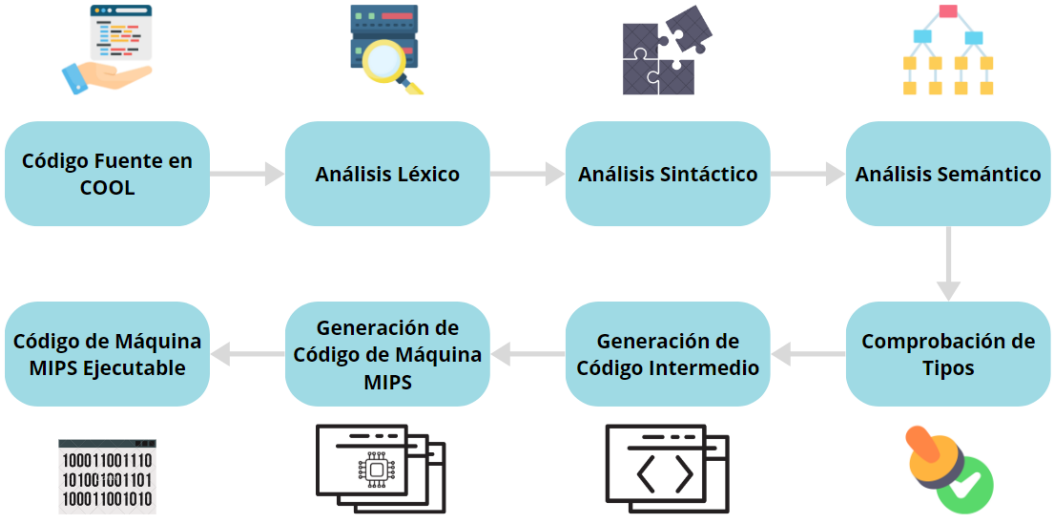
El proyecto está estructurado en varios módulos ubicados en la carpeta /src:

- main.py: Proporciona una interfaz para la configuración y ejecución del compilador.
- cool_compiler.py: El punto de entrada del compilador que coordina la ejecución de las fases del proceso de compilación.
- /lexic_analysis/lexer.py: Contiene la implementación del analizador léxico, responsable de identificar los tokens en el código fuente.
- /sintactic_analysis/parser.py: Implementa el analizador sintáctico, que verifica la estructura gramatical correcta del código.
- /semantic_analysis/ast_.py: Define las clases y estructuras de datos que representan el Árbol de Sintaxis Abstracta (AST).
- /semantic_analysis/semantic.py: Contiene la lógica para el análisis semántico, que evalúa el significado del código.
- /type_checker/type_checker.py: Implementa el type checker, encargado de verificar la coherencia de tipos en el programa.
- /cil_generation/cil_generator.py: Realiza la generación de código intermedio en representación de bajo nivel.
- /mips_generation/mips_generator.py: Se encarga de la generación de código de máquina específico para MIPS.
- /utils/constants.py: Define constantes utilizadas en varias partes del compilador.

- /utils/errors.py: Contiene las clases de errores personalizadas que se utilizan para informar sobre problemas durante el proceso de compilación.
- /utils/environment.py: Implementa la estructura de entorno utilizada en varias fases del compilador.

## 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, seguida de la optimización para mejorar el rendimiento. Finalmente, la generación de código de máquinas traduce el programa a instrucciones específicas del hardware para su ejecución. Cada fase desempeña un papel crucial en la creación de un software funcional y eficiente.




### 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

### ii.6. Generación de Código de Máquina

## III. Gramática

# II. Problemas técnicos

## Repositorio del proyecto

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

## Referencias