Sofía García – 22210
Joaquín Campos – 222155
Julio García Salas – 22076

Compiscript – Analizador Sintáctico, Semántico e IDE

Link al github:

https://github.com/Hayser8/compisanalisissemantico/tree/main

Descripción General

Nuestro proyecto implementa un compilador para el lenguaje académico Compiscript. El entregable incluye cuatro grandes componentes:

1. **Analizador sintáctico**: construido con ANTLR 4.13.1 a partir de la gramática Compiscript.g4.
2. **Análisis semántico**: se encarga de verificar las reglas del lenguaje, validando tipos, ámbitos, uso correcto de variables, constantes, funciones, clases y control de flujo.
3. **Tabla de símbolos**: estructura que almacena información sobre las declaraciones y resuelve el alcance de identificadores en variables, funciones, clases y parámetros.
4. **IDE gráfico (PySide6)**: un entorno minimalista que permite abrir, editar y analizar programas .cps, mostrando errores, símbolos y el árbol sintáctico en una interfaz amigable.

Cómo ejecutar el compilador (CLI)

Requisitos

- Python 3.10 o superior.
- Dependencia principal: antlr4-python3-runtime==4.13.1.

Instalación

1. Entrar al directorio compiscript/program.
2. Crear un entorno virtual:
    - En Linux/MacOS:
      python3 -m venv .venv
      source .venv/bin/activate
    - En Windows (PowerShell):
      py -3.10 -m venv .venv
      .venv\Scripts\activate
3. Instalar dependencias:
   pip install --upgrade pip
   pip install antlr4-python3-runtime==4.13.1

Ejecución del CLI

Sofía García – 22210
Joaquín Campos – 222155
Julio García Salas – 22076

Dentro del entorno virtual:

- Para analizar un archivo:
  python3 -m cli samples/ok_all.cps
- Opciones adicionales:
  • --json: salida en JSON (usado por el IDE).
  • --symbols: muestra la tabla de símbolos en el reporte.
  • --tree: imprime el árbol sintáctico generado por ANTLR.

Ejemplo:
python3 -m cli samples/bad_types.cps --symbols --tree

Cómo ejecutar el IDE

El IDE está construido con PySide6 y sirve como interfaz gráfica para probar programas en Compiscript. Permite abrir carpetas, navegar entre archivos .cps, editar con resaltado de sintaxis y ejecutar el compilador desde un botón.

Requisitos

- Python 3.10 o superior.
- Dependencias: PySide6==6.7.x, antlr4-python3-runtime==4.13.1.

Instalación

1. Entrar al directorio compiscript/ide.
2. Crear y activar un entorno virtual (similar al compilador).
3. Instalar dependencias:
   pip install PySide6==6.7.2 antlr4-python3-runtime==4.13.1

Ejecución

Con el entorno virtual activado:

python main.py

Uso básico

1. Abrir carpeta → seleccionar compiscript/program.
2. Abrir un archivo .cps (ejemplo: samples/ok_all.cps).
3. Guardar cambios en el editor.
4. Presionar el botón ▶ Run para analizar.
5. Revisar los paneles:
   - **Problems**: errores con línea y columna (doble clic para saltar).

- **Output**: logs y salida cruda del CLI.
- **Reporte**: resumen legible con símbolos y errores.
- **Outline**: variables, funciones y clases detectadas.

6. Alternar entre tema claro y oscuro con el botón correspondiente.

Diseño de Implementación (Arquitectura)

Dividimos el compilador en varias fases, cada una en módulos separados:

1. **Parser (ANTLR)**
   La gramática Compiscript.g4 genera el *lexer* y *parser* (CompiscriptLexer, CompiscriptParser).
   Esto produce un árbol sintáctico (parse tree) a partir del código fuente.

2. **DeclarationCollector**
   Primera pasada que recorre el árbol y declara símbolos en la tabla.
   Maneja variables, constantes, funciones (incluyendo anidadas) y clases con herencia.
   También crea los distintos scopes (global, de clase, de función).

3. **TypeLinker**
   Segunda pasada que resuelve anotaciones de tipo (por ejemplo, integer[], float, clases definidas).
   Completa los tipos en la tabla de símbolos y valida que los tipos referenciados existan.

4. **TypeCheckVisitor**
   Tercera pasada que valida las reglas semánticas del lenguaje:
   - Operaciones aritméticas, lógicas y comparaciones.
   - Tipos en asignaciones.
   - Aridad y tipos en llamadas a funciones.
   - Captura de variables en closures.
   - Uso correcto de this, return, break, continue.
   - Chequeo de constructores y herencia en clases.
   - Dead code después de return.
   - Tipos de elementos en arreglos e indexación.

5. **Tabla de símbolos**
   Implementada en symbols.py y scopes.py, mantiene toda la información de variables, parámetros, campos, clases y funciones.

Sofía García – 22210
Joaquín Campos – 222155
Julio García Salas – 22076

Permite resolver nombres y controlar la validez de redeclaraciones, además de manejar scopes anidados.

6. **IDE (frontend)**
   Construido en PySide6. Invoca al CLI (cli.py) con las opciones --json y --symbols. Interpreta el resultado y lo muestra en paneles interactivos (errores, símbolos, outline, árbol).

Discusión

Durante la implementación, lo que más nos costó fue:

- **Closures y funciones anidadas**: lograr que las funciones internas capturaran variables de su entorno correctamente.
- **Herencia y constructores**: coordinar la validación de parámetros en constructores y la resolución de clases base.
- **Dead code y returns obligatorios**: asegurar que el analizador detectara correctamente flujos de retorno faltantes.
- **Integración del IDE**: conectar el backend (CLI) con el frontend gráfico, sobre todo en la comunicación en JSON y el manejo de errores.
- **Tabla de símbolos unificada**: diseñar una estructura suficientemente flexible para variables, campos, constantes y funciones sin sobrecomplicar el código.

Estos retos nos obligaron a iterar varias veces la arquitectura, depurar errores y escribir una batería de pruebas sólida para garantizar la corrección.

Conclusiones

- Se logró implementar un compilador completo para Compiscript con analizador sintáctico, semántico y tabla de símbolos que cumple con todos los requisitos establecidos.
- Se desarrolló un IDE que hace más accesible el uso del compilador y visualiza errores y símbolos de forma amigable.
- El proyecto permitió comprender en profundidad cómo se implementan los compiladores: desde el análisis sintáctico hasta la gestión de tipos y scopes.
- Se considera que el resultado final es robusto y extensible, y sienta las bases para futuras fases como generación de código o un intérprete.