Analizador Semántico para CompiScript

Adrian Rodríguez 21691 Daniel Gómez 21429

Septiembre 2024

1 Introducción

Este documento presenta el desarrollo de un analizador sintáctico y semántico para el lenguaje CompiScript, junto con un IDE interactivo. El proyecto utiliza ANTLR para el análisis sintáctico y visitors para el análisis semántico, con una implementación de IDE basada en Streamlit.

2 Enlace al Repositorio

El código fuente del proyecto está disponible en el siguiente repositorio público:
`https://github.com/adrianrb469/compiler-construction/tree/main/project-1`

3 Enlace al Video de Demostración

`https://youtu.be/4A6uMO9N_bY?si=Z3FeAv1lzEgoGzV3`

4 Arquitectura del Compilador e IDE

4.1 Analizador Sintáctico

El analizador sintáctico se implementó utilizando ANTLR4. La gramática del lenguaje CompiScript se definió en un archivo .g4, que ANTLR utiliza para generar el parser y lexer.

4.2 Analizador Semántico

El análisis semántico se realiza mediante la implementación de visitors en Python. Estos visitors recorren el árbol sintáctico generado por ANTLR y realizan comprobaciones semánticas como:

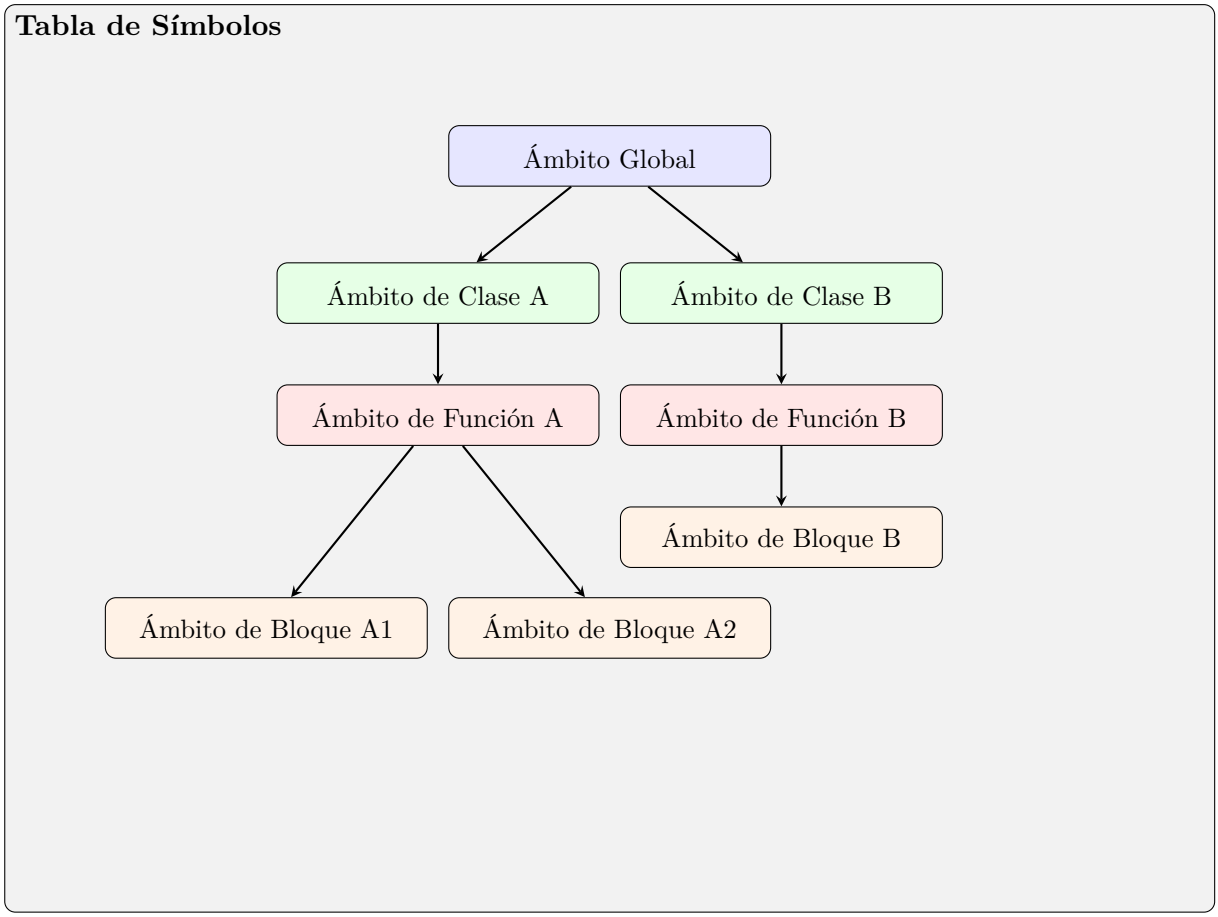
- Verificación de tipos
- Comprobación de declaraciones y ámbitos
- Validación de operaciones y expresiones

4.3 Tabla de Símbolos

La tabla de símbolos se implementó como una estructura de datos que mantiene información sobre los identificadores en el programa. Incluye detalles como:

- Nombre del símbolo
- Tipo de dato
- Ámbito

- Información adicional específica (para funciones, clases, etc.)



Explicación de la Tabla de Símbolos

La tabla de símbolos en CompiScript está implementada con una estructura jerárquica que maneja eficientemente diferentes tipos de símbolos y ámbitos. Su diseño se basa en las siguientes clases y enumeraciones:

Tipos de Símbolos y Datos

- **SymbolType (Enum):** Define los tipos de símbolos:
  - CLASS
  - FUNCTION
  - VARIABLE
- **DataType (Enum):** Define los tipos de datos, incluyendo:
  - Tipos primitivos: INT, FLOAT, STRING, BOOLEAN
  - Tipos complejos: ARRAY, OBJECT
  - Tipos especiales: ANY, VOID, NULL, UNION
- **UnionType:** Representa un tipo de unión, conteniendo un conjunto de DataType.

Clases Principales

Symbol

La clase Symbol sirve como base para todos los símbolos en el sistema. Contiene atributos fundamentales como nombre, tipo de símbolo, tipo de datos, línea y columna donde se declara. Además, incluye un campo de valor y un diccionario de atributos, proporcionando flexibilidad para almacenar información adicional específica de cada símbolo.

ClassSymbol

ClassSymbol extiende la funcionalidad de Symbol para representar clases. Introduce el concepto de herencia a través del atributo superclass, y mantiene colecciones de métodos y campos.

FunctionSymbol

La clase FunctionSymbol, también una extensión de Symbol, está diseñada para representar funciones. Su característica distintiva es la lista de parámetros.

Scope

Mantiene una estructura jerárquica a través de referencias a su ámbito padre y a sus hijos, junto con una colección de símbolos. Proporciona métodos para declarar nuevos símbolos, buscarlos en el ámbito actual o en ámbitos superiores, y actualizar sus valores. Esta clase es clave para implementar las reglas de visibilidad y acceso a variables en diferentes contextos del programa.

SymbolTable

La clase SymbolTable actúa como el gestor principal de toda la estructura de símbolos. Mantiene un ámbito global y un puntero al ámbito actual, permitiendo una navegación eficiente entre diferentes contextos del programa. Ofrece métodos para entrar y salir de ámbitos, esenciales para manejar bloques de código anidados. Además, proporciona funcionalidad para declarar símbolos y manejar clases, centralizando así las operaciones críticas de la tabla de símbolos.

Funcionalidades Clave

- **Manejo de Ámbitos:** La estructura permite ámbitos anidados y búsqueda jerárquica de símbolos.
- **Flexibilidad:** Soporta diferentes tipos de símbolos y datos, incluyendo uniones.
- **Herencia:** ClassSymbol incluye funcionalidad para manejar herencia de clases.
- **Serialización:** Métodos to_dict() y get_json() para representar la tabla en formato JSON.

Esta implementación proporciona una base sólida para el análisis semántico y la generación de código en CompiScript, permitiendo un manejo eficiente de símbolos en diferentes contextos del programa.

4.4 IDE Interactivo

El IDE interactivo se desarrolló utilizando Streamlit, proporcionando una interfaz web para:

- Edición de código CompiScript
- Visualización del análisis sintáctico y semántico en tiempo real

- Resaltado de errores y advertencias

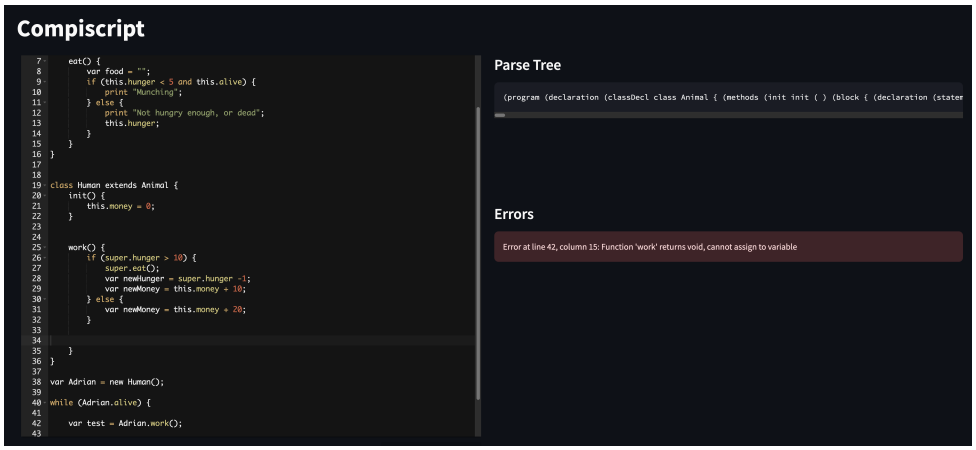


Figure 1: Captura de pantalla del IDE interactivo de CompiScript

5 Guía de Compilación y Ejecución

5.1 Requisitos del Sistema

- Python 3.8+
- ANTLR4
- Streamlit

5.2 Instrucciones de Ejecución

5.2.1 Compilación del Proyecto

Para compilar el proyecto completo, utilizamos Docker. Ejecute los siguientes comandos en la terminal:

```
docker build --rm . -t compiscript && \
docker run --rm -ti -v "$(pwd)/program":/program compiscript
```

Este comando realiza las siguientes acciones:

- Construye la imagen Docker con el tag 'compiscript'.
- Ejecuta un contenedor basado en esta imagen.
- Crea un directorio compartido entre el host y el contenedor para persistir los archivos.

5.2.2 Compilación de la Gramática y Ejecución del Driver

Dentro del contenedor Docker, para compilar la gramática y ejecutar el driver, usar:

```
antlr -Dlanguage=Python3 -visitor Compiscript.g4
```

Este comando genera el parser a partir de la gramática y ejecuta el programa de prueba.

5.2.3 Ejecución del IDE Interactivo

Para ejecutar el IDE interactivo, simplemente use el siguiente comando en su terminal local:

```
streamlit run ide.py
```

Esto iniciará el servidor Streamlit y abrirá el IDE interactivo en su navegador web predeterminado.

6 Ejemplos de Código CompiScript

6.1 Ejemplo 1: Declaración y Uso de Variables

```
var x = 5;
var y = 10;
var z = x + y;
print z;
```

Análisis Semántico:

- Verificación de tipos en la asignación y suma
- Comprobación de declaraciones de variables

6.2 Ejemplo 2: Objetos

```
class Animal {
    init() {
        this.alive = true;
        this.hunger = 10;
    }

    eat() {
        var food = "";
        if (this.hunger < 5 and this.alive) {
            print "Munching";
        } else {
            print "Not hungry enough, or dead";
            this.hunger;
        }
    }
}


class Human extends Animal {
    init() {
        this.money = 0;
    }

    work() {
        if (super.hunger > 10) {
            super.eat();
            var newHunger = super.hunger -1;
            var newMoney = this.money + 10;
        } else {
            var newMoney = this.money + 20;
        }
    }
}

var Adrian = new Human();
```

```
36
37 while (Adrian.alive) {
38
39     var test = Adrian.work();
40
41 }
```

Basado en el código proporcionado, el análisis semántico realizaría las siguientes verificaciones:

- **Herencia y Sobrescritura:** Verificar que la clase `Human` extiende correctamente de `Animal` y que la sobrescritura del método `init()` es válida.
- **Acceso a Miembros:** Comprobar el acceso correcto a los atributos de la clase base usando `super`, como en `super.hunger` y `super.eat()`.
- **Declaración y Uso de Variables:** Verificar la declaración y el ámbito de variables locales como `food`, `newHunger`, y `newMoney`.
- **Comprobación de Tipos:** Asegurar la coherencia de tipos en operaciones como comparaciones (`this.hunger < 5`) y asignaciones.
- **Llamadas a Métodos:** Verificar que las llamadas a métodos como `eat()` y `work()` sean válidas y correspondan a métodos definidos.
- **Inicialización de Objetos:** Comprobar la correcta instanciación de objetos, como en `var Adrian = new Human()`.
- **Estructuras de Control:** Validar el uso correcto de estructuras como `if` y `while`, asegurando que las condiciones sean booleanas.
- **Retorno de Funciones:** Verificar que los métodos que deben retornar valores lo hagan correctamente (por ejemplo, `eat()` no tiene un retorno explícito).

7 Decisiones de Diseño

- Uso de ANTLR para generar el parser debido a su robustez y facilidad de integración con Python
- Implementación de visitors para el análisis semántico, permitiendo una separación clara entre la estructura sintáctica y la lógica semántica
- Utilización de Streamlit para el IDE por su simplicidad y capacidades de actualización en tiempo real

8 Desafíos y Soluciones

8.1 Manejo de Ámbitos en la Tabla de Símbolos

Desafío: Implementar una estructura eficiente para manejar ámbitos anidados en la tabla de símbolos, permitiendo la correcta resolución de variables en diferentes contextos.

Solución: Se implementó una estructura jerárquica utilizando la clase `Scope`. Cada `Scope` mantiene una referencia a su ámbito padre y una lista de ámbitos hijos, junto con un diccionario de símbolos. Esta estructura permite una búsqueda eficiente de símbolos a través de la jerarquía de ámbitos, facilitando la resolución de variables en ámbitos anidados sin necesidad de una pila explícita.

8.2 Gestión del Ámbito Actual

Desafío: Mantener y actualizar correctamente el ámbito actual durante el análisis del código, especialmente al entrar y salir de diferentes bloques de código.

Solución: La clase `SymbolTable` mantiene una referencia al ámbito actual y proporciona métodos `enter_scope()` y `exit_scope()` para navegar entre ámbitos. Esto permite un manejo fluido del contexto actual durante el análisis, asegurando que las declaraciones y referencias a símbolos se procesen en el ámbito correcto.

8.3 Manejo de Herencia en la Tabla de Símbolos

Desafío: Implementar un sistema robusto para manejar la herencia de clases, permitiendo el acceso y la sobrescritura correcta de métodos y atributos heredados.

Solución: Se extendió la clase `ClassSymbol` para incluir una referencia a la superclase y métodos específicos para manejar la herencia. La función `inherit()` permite copiar métodos y campos de la clase padre, mientras que los métodos `add_method()` y `add_field()` facilitan la sobrescritura y adición de nuevos miembros en las subclases.

9 Conclusiones

El proyecto ha logrado implementar un analizador semántico para CompiScript, junto con un IDE interactivo. Las principales áreas de mejora incluyen la optimización del rendimiento para códigos más extensos y la ampliación de las capacidades de análisis semántico.

10 Referencias

- Documentación de ANTLR4: `https://www.antlr.org/`
- Documentación de Streamlit: `https://docs.streamlit.io/`