UNIVERSIDAD COOPERATIVA DE COLOMBIA

VIGILADA MINEDUCACIÓN

Proyecto Integrador Final – Lenguaje para cálculo de impuestos
Compiladores

Profesor:
Ing. Gustavo Sánchez

Estudiantes:
Nelson Andrés Chaves Chaves
Diego Alejandro Rosero Cisneros

Universidad Cooperativa de Colombia “Campus Pasto”
Programa de Ingeniería de Software
Colombia

2025

Contenido

Tabla de Ilustraciones

1. Introducción

En este documento se presenta el desarrollo del proyecto integrador final de la asignatura de Compiladores, cuyo objetivo es construir un mini compilador completo aplicando todas las fases estudiadas en clase como el análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio y traducción exclusiva a código Python.

El compilador se implementó utilizando ANTLR4 para la construcción del lexer y parser, y Python 3 para las fases de análisis semántico, generación de código intermedio y generación de código final. El proyecto sigue el flujo profesional de desarrollo de compiladores exigido en el taller, y se estructura como un repositorio ejecutable, automatizado y testeable.

2. Descripción del tema y del mini-lenguaje

2.1. Contexto del problema

El tema que se escogio para el mini - compilador es un lenguaje simple para definir reglas de cálculo de impuesto sobre ingresos. La idea es permitir que un usuario escriba reglas de negocio como:

- Definir ingresos y variables intermedias.
- Especificar condiciones tipo “si ingreso es mayor al umbral entonces calcular impuesto”.
- Imprimir resultados.

Este tipo de reglas es común en sistemas de nómina, contabilidad básica o simuladores de impuestos.

2.2. Objetivo del lenguaje

El mini-lenguaje se diseñó para permitir declarar y asignar variables numéricas por ejemplo ingreso, impuesto, descuento, escribir reglas condicionales con la palabra clave si, comparando variables con literales numéricos, realizar expresiones aritméticas con +, -, * y /, mostrar resultados usando la palabra clave print y traducir automáticamente el programa escrito en este lenguaje a un script Python ejecutable, respetando la semántica de las reglas.

2.3. Ejemplo general de programa

Un ejemplo de programa válido en el lenguaje es:

```
ingreso = 120000;
si ingreso > 100000: impuesto = ingreso * 0.2;
print(impuesto);
```

Este programa define un ingreso, aplica una regla de impuesto si se supera un umbral y finalmente imprime el valor del impuesto calculado.

3. Diseño del lenguaje y gramática ANTLR4

El diseño del lenguaje se formalizó en el archivo gramatica.g4, donde definimos tanto las reglas léxicas como las sintácticas. A continuación se resume los elementos principales.

3.1. Elementos léxicos

Los elementos léxicos son las piezas más pequeñas del código fuente; son los símbolos básicos que el analizador léxico reconoce antes de construir la estructura del programa. Cada elemento léxico se convierte en un *token*, que indica qué tipo de símbolo es (palabra clave, identificador, número, operador, etc.) y en qué posición aparece dentro del archivo.

Los tokens definidos incluyen:

- **Identificadores (ID):** nombres de variables como ingreso, impuesto.
- **Números (NUMBER):** literales numéricos enteros o decimales.
- **Palabras clave:**
    - **Si**: para las reglas condicionales.
    - **Print:** para imprimir en pantalla.

- **Operadores aritméticos:** +, -, *, /.
- **Operadores relacionales:** >, <, >=, <=, ==, !=.
- **Símbolos de puntuación**: =, :, ;, paréntesis.
- **Comentarios:** líneas que comienzan con //, ignoradas por el compilador.
- **Espacios en blanco:** se consumen sin generar tokens, para simplificar la gramática.

3.2. Estructura sintáctica

Mientras que el análisis léxico identifica los tokens, la estructura sintáctica define cómo esos tokens se pueden combinar para formar sentencias válidas en el lenguaje. Es decir, la sintaxis describe las “frases” correctas del mini - lenguaje y determina qué patrones están permitidos para asignaciones, reglas si y sentencias print.

A nivel sintáctico definimos un programa (program) como una secuencia de sentencias y Tipos de sentencias (statement):

- Asignaciones simples.
- Reglas condicionales.
- Impresiones.

De forma resumida, la estructura es:

- Asignación:
- ID = expr ;
- Regla condicional:
- si cond : ID = expr ;
- Impresión:
- print(ID) ;

Las expresiones (expr, term, factor) respetan la precedencia de operadores como la multiplicación y división tienen mayor precedencia que suma y resta, tambien se permiten paréntesis para agrupar.

3.3. Semántica informal

La semántica informal explica el significado de los programas más allá de que estén bien escritos sintácticamente. En esta sección no me enfoco en la forma, sino en lo que realmente ocurre cuando se ejecuta cada sentencia: cómo se interpretan las asignaciones, qué implica que una condición sea verdadera o falsa y qué efectos tienen las expresiones sobre los valores de las variables.

La semántica del lenguaje se define de manera informal así:

- Una variable debe ser declarada/asignada antes de cualquier uso en expresiones o en print.
- La instrucción si cond : ID = expr; significa:
    - Evaluar la condición cond.
    - Si la condición es verdadera, evaluar expr y asignar el resultado a ID.

- Las condiciones (cond) comparan una variable con un número utilizando operadores relacionales.
- Las expresiones aritméticas trabajan sobre valores numéricos.
- print(ID); imprime el valor actual de la variable.

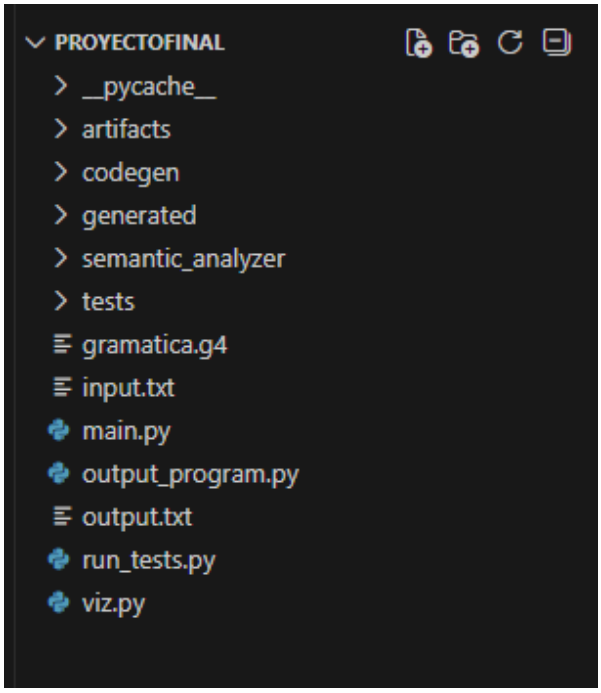
4. Arquitectura del mini-compilador

La arquitectura del mini-compilador está organizada por capas que siguen directamente las fases clásicas de un compilador: léxica, sintáctica, semántica, generación de código intermedio y generación de código final en Python. Cada carpeta del proyecto representa una responsabilidad clara dentro de ese flujo, lo que facilita entender el recorrido del código fuente desde que entra como archivo ".txt" hasta que termina convertido en un script Python ejecutable.

4.1. Estructura de carpetas

A nivel físico, el proyecto queda organizado en una carpeta raíz llamada ProyectoFinal, donde se agrupan tanto los módulos del compilador como los archivos de entrada, salida y pruebas. Desde esta raíz se puede ver la separación entre el código principal (main.py), la gramática (gramatica.g4), los módulos generados por ANTLR, el análisis semántico, la generación de código y los directorios auxiliares para pruebas y artefactos. Esta estructura refleja de forma directa la arquitectura por fases que pide el taller.

Ilustración 1 Estructura de carpetas



4.2. Flujo por fases del compilador

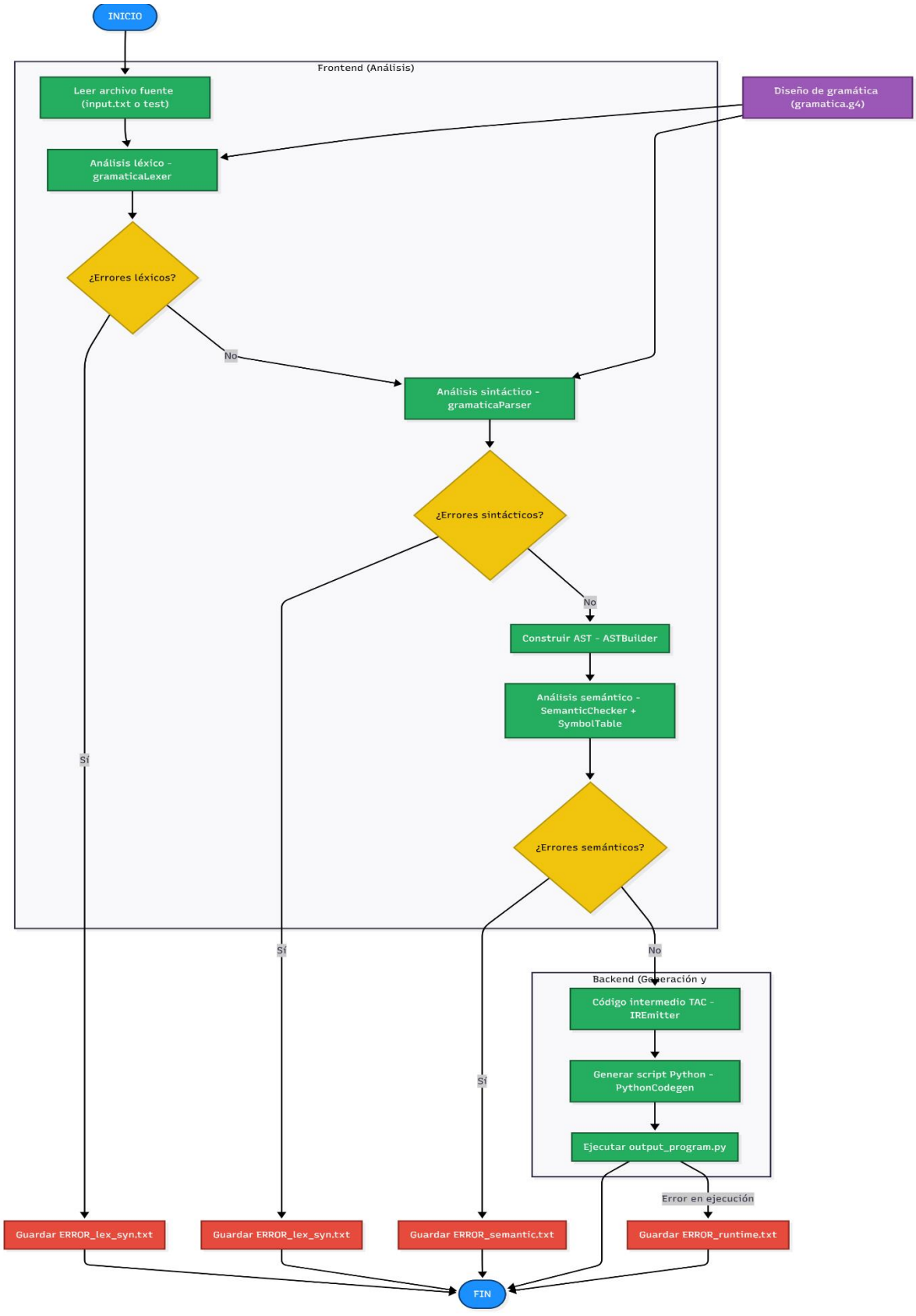
El flujo real del compilador es el siguiente:

a. Lectura del archivo fuente (input.txt o un test).
b. Análisis léxico con gramaticaLexer.
c. Análisis sintáctico con gramaticaParser.
d. Construcción del Árbol de Sintaxis Abstracta (AST) con ASTBuilder.
e. Análisis semántico con SemanticChecker y SymbolTable.
f. Generación de código intermedio (IR/TAC) con IREmitter.
g. Generación de código final en Python con PythonCodegen.
h. Ejecución del script Python generado y captura de la salida.

4.3. Diagrama de la arquitectura

En la siguiente figura se presenta el diagrama de flujo general de la arquitectura de mi mini-compilador. En la parte izquierda se observa el frontend (análisis)**,** donde a partir del archivo fuente input.txt y de la gramática gramatica.g4 se realizan el análisis léxico, sintáctico, la construcción del AST y el análisis semántico, detectando posibles errores en cada fase. Una vez que el programa es válido, el flujo continúa en la parte derecha con el backend (generación y ejecución)**,** donde se produce el código intermedio TAC, se genera el script output_program.py en Python y finalmente se ejecuta, registrando también los errores de tiempo de ejecución. Este diagrama resume de manera visual cómo el compilador transforma el código fuente del mini-lenguaje hasta llegar a un programa

Ilustración 2 diagrama de flujo



5. Análisis léxico

El análisis léxico es la primera fase del compilador y se encarga de leer el archivo fuente carácter por carácter para agruparlos en tokens significativos. En esta etapa todavía no se valida la estructura completa del programa, sino que simplemente se clasifica cada fragmento como identificador, número, palabra clave, operador o símbolo especial, preparando la entrada que usará después el analizador sintáctico.

5.1. Decisiones de diseño

Para el análisis léxico, se usó ANTLR4 a partir de la sección léxica del archivo gramatica.g4. Aquí se definieron tokens para identificadores, números, palabras clave, operadores, símbolos y comentarios. Los comentarios y espacios en blanco se ignoran en el canal de análisis, para simplificar las fases posteriores.

El lexer generado se almacena en generated/gramaticaLexer.py y es utilizado por main.py para leer el archivo fuente y producir la secuencia de tokens.

5.2. Manejo de errores lexicós

Se implementa un ErrorListener personalizado para capturar errores léxicos. Cuando aparece un carácter no reconocido por ejemplo $, el sistema va a genera un archivo ERROR_lex_syn.txt dentro de la carpeta artifacts/<CASO>/. Paso a seguir, registra un mensaje claro indicando la posición del error y por ultimo marca el caso de prueba como fallido durante la ejecución de run_tests.py.

6. Análisis sintáctico.

El análisis sintáctico toma la secuencia de tokens producida por el lexer y verifica que siga las reglas de la gramática del lenguaje. En esta fase el compilador comprueba que las sentencias estén bien formadas (por ejemplo, que no falten **;** o **:**) y construye el árbol de sintaxis concreta que representa la estructura jerárquica del programa. Si algo no encaja con la gramática, aquí se reportan los errores sintácticos.

6.1. Reglas principales

El parser se genera también a partir de gramatica.g4 y se encapsula en generated/gramaticaParser.py. En main.py se utiliza la regla inicial program para construir el árbol de sintaxis concreta.

Se soportan las secuencias de sentencias terminadas en punto y coma (;). Tambien se concreta asignaciones, reglas condicionales y sentencias de impresión y expresiones con precedencia y asociatividad adecuadas.

6.2. Manejo de errores sintácticos

De forma similar al lexer, se agrega un ErrorListener para errores sintácticos. Cuando el parser encuentra una secuencia inválida como por ejemplo falta de punto y coma (;) o dos puntos (:), se procede a generar un archivo ERROR_lex_syn.txt en la carpeta correspondiente de artifacts/. Despues se detiene las fases posteriores para ese caso y el script run_tests.py marca el test como fallido, clasificado como error sintáctico.

7. Análisis semántico

El análisis semántico revisa que un programa que ya es léxica y sintácticamente correcto también tenga sentido desde el punto de vista de las reglas del lenguaje. En esta etapa no se mira la forma del código, sino su significado: si las variables han sido declaradas antes de usarse, si las operaciones son válidas y si se cumplen las restricciones que definí para mi mini-lenguaje. Para ello uso un AST y una tabla de símbolos.

7.1. Tabla de símbolos y ámbitos

El análisis semántico se implementa en el módulo semantic_analyzer/ realizando los siguientes pasos:

- symbol_table.py define una tabla de símbolos sencilla que almacena las variables declaradas en el programa.
- Se mantiene el conjunto de identificadores definidos para verificar usos posteriores.
- Aunque el lenguaje es simple y trabaja en un único ámbito global, la estructura permite extenderlo a más scopes si fuera necesario.

7.2. Reglas semánticas implementadas

Las principales reglas semánticas son:

- **Uso antes de definición:** cualquier variable utilizada en el lado derecho de una asignación, en una condición si o en print debe haber sido definida antes.

- **Variables en reglas**: en una regla si, tanto la variable de la condición como la del lado derecho deben existir previamente o definirse de forma coherente.
- **División por cero literal:** cuando se detecta una expresión del tipo expr / 0, se genera un error semántico específico.

Estas reglas se aplican recorriendo el AST con la clase SemanticChecker, definida en semantic_analyzer/semantic_checker.py.

7.3. Manejo de errores semánticos

Cuando se viola alguna de las reglas anteriores en primer lugar se lanza una excepción SemanticError con un mensaje explicativo, después el mensaje se registra en ERROR_semantic.txt dentro de la carpeta artifacts/<CASO>/, y por último el archivo output.txt del caso deja evidencia del tipo de error detectado.

Ejemplos de errores semánticos que se prueban:

- Imprimir una variable no declarada.
- Usar variables no definidas en condiciones.
- Dividir por cero explícitamente.

8. Generación de código intermedio

A partir del AST validado por la fase semántica, el módulo codegen/ir_emitter.py genera un código intermedio en forma de TAC (Three Address Code) definido en codegen/ir.py.

Las instrucciones principales son:

- IRAssign: representa asignaciones simples x = expr.
- IRIfAssign: representa reglas condicionales if cond then x = expr.
- IRPrint: representa la impresión del valor de una variable.
- IRProgram: encapsula la lista completa de instrucciones del programa.

Este IR se guarda para cada caso en un archivo 5_ir.txt dentro de artifacts/<CASO>/, lo que permite inspeccionar y validar la traducción intermedia.

9. Traducción a código Python

La última fase traduce el IR del paso 8 a Python 3 usando el módulo codegen/python_codegen.py realizando los siguientes pasos:

- **IRAssign** se traduce a una sentencia Python x = ...
- **IRIfAssign** se traduce a una estructura if con la condición y la asignación correspondiente.
- **IRPrint** se traduce directamente a print(x).

Para cada caso de prueba se genera un archivo 6_output_program.py en su carpeta de artifacts y cuando el compilador se ejecuta sobre input.txt, el script final se copia a la raíz como “output_program.py”, este archivo es python puro y se ejecuta con el intérprete estándar de Python 3.

La ejecución de output_program.py se captura en “7_stdout.txt” que es el resultado de la ejecución, y 7_stderr.txt si ocurre algún error en tiempo de ejecución.

10. Pruebas y validación

Para comprobar que el mini-compilador funciona de forma correcta y robusta, diseñé una fase de pruebas y validación basada en varios archivos de entrada que cubren tanto casos válidos como casos con errores intencionales. La idea es no solo demostrar que el compilador acepta programas correctos, sino también que detecta y reporta adecuadamente los errores léxicos, sintácticos y semánticos, apoyándome en una estructura de tests automatizados.

10.1. Organización de los casos de prueba

Las pruebas se organizan en la carpeta “tests/valid” las cuales contiene al menos 10 programas válidos desde el “V1.txt” hasta el “V10.txt” que cubren las asignaciones simples, las reglas condicionales, las expresiones aritméticas y las impresiones.

En la carpeta “tests/invalid” se tiene al menos 10 programas con errores desde el archivo “I1_lex.txt” al archivo “I10_syn.txt”, repartidos en errores léxicos, errores sintácticos y errores semánticos.

Cada archivo de prueba incluye comentarios con metadatos del estilo “ @title: ejemplo de prueba” y “@expect: ok” o también “@expect: error_semantic”.

10.2. Ejemplos de casos validos

Un caso válido típico define un ingreso, aplica varias reglas condicionales y finalmente imprime el resultado. Estos casos permiten verificar que todas las fases se ejecutan sin errores y que el código generado produce la salida esperada.

10.3. Ejemplos de casos con error

En los casos inválidos se prueban explícitamente:

- Caracteres ilegales para forzar errores léxicos.
- Sintaxis incorrecta (por ejemplo, omitir ; o :) para forzar errores sintácticos.
- Uso de variables no declaradas o división por cero para forzar errores semánticos.

Cada error queda registrado en los archivos el archivo “ERROR_lex_syn.txt” o en “ERROR_semantic.txt” dentro de artifacts.

10.4. Automatización de pruebas

El archivo run_tests.py automatiza la validación con esto recorre todos los archivos de “tests/valid” y “tests/invalid”. Despues extrae las anotaciones “@expect” de cada caso. Ejecuta el compilador mediante main.py generando los artefactos correspondientes. Verifica si el resultado coincide con lo esperado (ok, error léxico/sintáctico, error semántico) y muestra un resumen global de las pruebas. De esta forma se cumple el requisito del taller de contar con pruebas automatizadas y verificables.

11. Problemas encontrados y soluciones

Durante el desarrollo del mini-compilador se presentaron algunos problemas prácticos que se fueron resolviendo paso a paso:

- **Problema 1: entender el flujo completo del compilador**
  Al inicio se nos dificulto ver cómo se conectaban todas las fases; solo ejecutaba main.py sin tener claro qué pasaba primero y qué artefactos se generaban en cada etapa.

  Solución: dibujamos el flujo completo (código fuente, léxico, sintáctico, semántico, IR, Python) y revisar las carpetas de artifacts después de cada ejecución. Esto nos ayudó a entender qué archivo pertenece a cada fase y a relacionar mejor la teoría con la implementación.

- **Problema 2: organizar la estructura de carpetas como en el taller**
  Otro problema inicial fue ordenar los módulos del proyecto (gramática, semántica, generación de código, pruebas) de forma similar a los ejemplos

vistos en clase. Al principio teníamos archivos mezclados y las importaciones en Python fallaban.

Solución: se reorganizó el proyecto creando las carpetas semantic_analyzer, generated, codegen, tests y artifacts, y se ajustaron los imports en main.py. Con esto, la arquitectura quedó más clara y fue más fácil ubicar cada responsabilidad.

- **Problema 3: diferenciar los tipos de errores**
  En una primera versión, cualquier error aparecía solo en la terminal, y no era sencillo distinguir si el problema era léxico, sintáctico o semántico.

  Solución: se implementó archivos separados para los errores de cada fase (ERROR_lex_syn.txt y ERROR_semantic.txt) y un output.txt resumen por caso. De esta forma puedo saber rápidamente qué tipo de fallo ocurrió y en qué etapa del compilador se produjo.

Estos problemas y sus soluciones fortalecieron la comprensión del flujo completo de un compilador y de la importancia de separar claramente cada responsabilidad en módulos.

12. Conclusiones

El mini-compilador desarrollado cumple con las fases exigidas en el proyecto integrador final: análisis léxico, sintáctico, semántico, generación de código intermedio y traducción a Python, integrando ANTLR4 con Python en un entorno real.

A lo largo del proyecto, diseñamos un mini-lenguaje específico para reglas de impuesto sobre ingresos. Formalizamos su sintaxis en una gramática ANTLR4 y la probamos con casos simples y complejos. Implementamos un análisis semántico con tabla de símbolos y reglas claras sobre uso de variables y división por cero. Construimos un IR/TAC sencillo y una fase de generación de código Python que produce scripts ejecutables y organizamos un conjunto de pruebas válidas e inválidas, acompañadas por artefactos y automatización, que muestran la robustez del compilador.

Este trabajo nos permitió aplicar de manera práctica los conceptos vistos en la asignatura de Compiladores y comprender cómo se conectan todas las fases de un proceso de traducción real, desde el código fuente hasta la ejecución de un programa en un lenguaje objetivo que en este caso fue Python.