

Universidad Mariano Gálvez De Guatemala  
Sede De Boca Del Monte, Villa Canales  
Ingeniería en Sistemas de Información Ing.  
Ezequiel Urizar Araujo  
Compiladores- Sección B



### **Proyecto Final**

## **Diseño e Implementación de un Compilador con Optimización Manual y Automática en LLVM IR**

**Enlace del proyecto**

**<https://github.com/rramirezg18/ProyectoFinalCompiladores.git>**

<b>Nombre:</b>	<b>Carnet:</b>	<b>participación</b>
<b>Bryan Manuel Pineda Orozco</b>	<b>7690-16-8869</b>	<b>100%</b>
<b>Jonathan Joel Chán Cuellar</b>	<b>7690-22-1805</b>	<b>100%</b>
<b>Roberto Antonio Ramirez Gómez</b>	<b>7690-22-12700</b>	<b>100%</b>
<b>Jean Klaus Castañeda Santos</b>	<b>7690-22-892</b>	<b>100%</b>
<b>Edwin Rolando Ixcoy Tot</b>	<b>7690-16-1582</b>	<b>100%</b>

**Guatemala, 23 de mayo del 2025**

## Índice

Introducción .....	3
Diseño Del Lenguaje.....	5
Análisis Semántico.....	7
Generación de Código Intermedio .....	10
Arquitectura General del Compilador y Optimizaciones .....	12
Pruebas .....	14
Conclusiones .....	21

## Introducción

A continuación, se expone el desarrollo y funcionamiento del proyecto final de la asignatura de Compiladores, cuyo objetivo principal fue la implementación de un compilador funcional para el lenguaje. Este lenguaje fue diseñado con fines educativos, permitiendo al usuario crear y modificar figuras geométricas mediante instrucciones sencillas, integrando además estructuras de control, funciones y operaciones básicas de programación.

La construcción del compilador implicó la aplicación práctica de los conocimientos adquiridos durante el curso, abarcando desde el análisis léxico y sintáctico hasta la generación de código intermedio optimizado. Para ello, se utilizó ANTLR4 como herramienta principal para la generación del parser a partir de una gramática formal, así como Python para el desarrollo del núcleo del compilador y LLVM para la producción y optimización del código intermedio. Estas tecnologías permitieron modelar todas las fases esenciales del proceso de compilación: análisis, interpretación semántica, representación mediante árbol de sintaxis abstracta (AST) y posterior traducción a código intermedio optimizable.

El proyecto se estructura en diversos módulos que organizan cada una de las etapas del compilador, e incluye también una colección de archivos de prueba que demuestran las capacidades del lenguaje. En estos se pueden observar instrucciones para la creación de figuras como círculos, cuadrados, polígonos, así como operaciones para modificar colores, tamaños y posiciones, validando el

correcto funcionamiento de cada componente del sistema.

En este documento podemos encontrar tanto el diseño como la ejecución del compilador, explicando la función de cada uno de sus componentes principales, la estructura del repositorio, los lenguajes y herramientas empleadas, y las funcionalidades implementadas. Todo esto con el objetivo de demostrar la integración de teoría y práctica en el campo de la construcción de lenguajes de programación.

## **Diseño Del Lenguaje**

Esta gramática especifica la estructura sintáctica del lenguaje, incluyendo sus reglas y tokens utilizando ANTLR4

### **Estructura General del Lenguaje**

define un lenguaje de programación personalizado que tiene una estructura específica. Los programas en este lenguaje comienzan con un identificador de programa seguido de llaves que contienen una función principal obligatoria (main) y funciones adicionales opcionales.

### **Regla Principal**

La regla principal de la gramática es, que define la estructura general del programa: un identificador de variable (nombre del programa), seguido de un bloque principal main y funciones adicionales opcionales

La estructura básica del lenguaje es:

```
programx{  
    main(){  
        // bloque de instrucciones  
    }  
    // funciones adicionales  
}
```

### **Componentes Principales**

#### **Programa**

- Un programa comienza con un identificador (VARIABLE), seguido de un bloque principal y funciones opcionales

- La función `main()` es obligatoria

### **Función Principal (main)**

- Definida como `main()` seguida de un bloque de instrucciones

### **Instrucciones**

El lenguaje soporta diversos tipos de instrucciones, incluyendo:

- Declaración y asignación de variables
- Impresión (sentencia `print`)
- Estructuras de control condicional (`if-else`)
- Bucles (`while`, `for`)
- Retorno de valores
- Llamadas a funciones

### **Tipos de Datos**

Soporta cuatro tipos de datos básicos:

- Enteros (`int`)
- Números de punto flotante (`float`)
- Booleanos (`boolean`)
- Cadenas de texto (`string`)

### **Expresiones**

La gramática implementa una jerarquía clara de precedencia para las expresiones, siguiendo este orden:

1. Expresiones relacionales (comparaciones)
2. Expresiones aditivas (suma, resta)
3. Expresiones multiplicativas (multiplicación, división, módulo)
4. Potenciación

5. Expresiones unarias (negación)
6. Átomos (variables, números, cadenas, booleanos, llamadas a funciones, expresiones entre paréntesis)

## **Tokens y Operadores**

El lenguaje define tokens para:

- Palabras clave (main, if, else, while, for, print, return)
- Operadores aritméticos (+, -, \*, /, ^, %)
- Operadores relacionales (==, !=, <, >, <=, >=)
- Operadores de incremento/decremento (++ , --)

## **Implementación**

La gramática es procesada por ANTLR4 para generar:

- **GramaticaLexer.py**: Analiza el texto de entrada y lo convierte en tokens
- **GramaticaParser.py**: Analiza la secuencia de tokens según las reglas gramaticales

Estos componentes trabajan juntos para analizar programas escritos en este lenguaje personalizado.

## **Características Adicionales**

La gramática también maneja espacios en blanco y comentarios de línea, que son ignorados durante el análisis.

## **Análisis Semántico**

La fase de análisis semántico representa una etapa crucial dentro del proceso de compilación, ya que, valida la coherencia lógica y contextual del programa, más allá de su correcta estructura sintáctica. En el compilador, esta etapa se lleva a cabo una vez construido el árbol de sintaxis abstracta (AST), el cual representa de

manera jerárquica y estructurada cada una de las instrucciones del lenguaje fuente.

Para realizar el análisis semántico, se implementó una tabla de símbolos mediante la clase `tabla_simbolos.py`, la cual se encarga de gestionar toda la información necesaria sobre variables, funciones, tipos de datos y ámbitos. Esta tabla funciona como un diccionario jerárquico donde se almacena el identificador de cada símbolo, su tipo (`int`, `float`, `bool`, `string`), su valor si aplica, y el contexto o alcance en el que fue definido (`global` o `local`).

Durante el recorrido del AST, que se efectúa a través del archivo `listener.py` y el uso del patrón listener de ANTLR, se implementan múltiples validaciones semánticas, tales como:

- Verificación de que toda variable haya sido declarada antes de su uso.
- Comprobación de que los tipos de datos coincidan en operaciones aritméticas o booleanas.
- Validación de parámetros y retornos en las llamadas a funciones.
- Control del alcance de variables para evitar colisiones de nombres en ámbitos anidados.
- Comprobación semántica de las propiedades de las figuras, como el uso adecuado de coordenadas, tamaños y colores.

Por ejemplo, al momento de procesar una instrucción para crear un cuadrado, se verifica semánticamente que los valores de entrada para posición y tamaño sean del tipo correcto (por ejemplo, flotantes o enteros), y que no se intente asignar a un color un valor fuera del rango permitido.

Además, la tabla de símbolos permite gestionar funciones definidas por el usuario, almacenando tanto su firma como su cuerpo para su posterior reutilización. Esto habilita un correcto control del flujo de ejecución y la posibilidad de realizar validaciones más complejas, como el control de tipos entre parámetros y



argumentos.

El análisis semántico implementado en el compilador garantiza la corrección lógica del programa antes de pasar a la generación de código intermedio, evitando así errores que podrían provocar resultados inesperados o incluso fallos durante la ejecución.

### **Manejo de Ámbitos y Contextos**

La gestión de ámbitos es fundamental para garantizar la validez semántica del código. El compilador implementa un sistema de contextos jerárquicos mediante una pila de tablas de símbolos. Cada vez que se ingresa a un nuevo bloque (función, condicional, ciclo), se crea una nueva tabla de símbolos que hereda del ámbito anterior.

#### **Ejemplo de alcance:**

```
main() {  
    int x = 10;  
    if (x > 5) {  
        int y = 20;  
        print(y); // OK  
    }  
    print(y); // ERROR: y no está en el ámbito global  
}
```

El compilador detecta este error porque la variable `y` solo está definida en el contexto del bloque `if`.

Este enfoque permite:

- Evitar colisiones de nombres en funciones anidadas.
- Aislar variables locales y temporales.
- Gestionar recursividad y llamadas encadenadas correctamente.

## Tipos de Errores Semánticos Detectados

El compilador implementa una validación semántica exhaustiva que permite detectar varios errores típicos. A continuación, se presenta una tabla con ejemplos:

Tipo de Error	Descripción	Ejemplo	Resultado
Variable no declarada	Uso de identificadores sin previa declaración	<code>print(x);</code> sin declarar <code>x</code>	Error
Reasignación incompatible	Asignar un tipo a otro incompatible	<code>int x = "hola";</code>	Error
Retorno inválido	<code>return</code> sin valor en funciones no <code>void</code>	función <code>int</code> sin <code>return</code>	Error
Argumentos incorrectos	Número o tipo incorrecto en llamada a función	<code>sumar(4, "texto");</code>	Error
Sombra de variables	Reutilizar nombre de variable en mismo ámbito	<code>int x; int x;</code>	Error
Uso fuera de ámbito	Acceso a variables locales desde otro bloque	<code>print(y);</code> fuera del <code>if</code>	Error

## Generación de Código Intermedio

Superada la fase semántica, el compilador avanza hacia una de las etapas más técnicas y estructuralmente relevantes del proceso de traducción: la generación de código intermedio, utilizando para ello el lenguaje LLVM IR (Intermediate Representation). Esta fase es implementada en el módulo `generador_ir.py`, el cual toma como entrada el AST previamente validado y genera como salida un archivo en formato `.ll`, ubicado en el directorio `codigoIR/`.

LLVM IR es un lenguaje de representación intermedia de bajo nivel, que permite describir instrucciones de forma detallada pero aún independiente de la arquitectura física. Gracias a su diseño modular y su compatibilidad con herramientas de

optimización, se convierte en una excelente elección para la etapa intermedia de un compilador moderno.

En el proyecto, la generación de código intermedio se realiza mediante una traducción recursiva del AST, donde cada nodo se transforma en una o varias instrucciones LLVM. Esta transformación incluye:

- Declaración de variables y estructuras de datos en memoria.
- Instrucciones aritméticas y lógicas equivalentes a las expresadas en.
- Creación de figuras gráficas mediante funciones que generan primitivas en el IR, interpretables por un motor gráfico o por el backend de ejecución.
- Control de flujo (condicionales y ciclos), utilizando etiquetas y saltos condicionales (br, label).
- Definición de funciones y manejo de retornos (ret) con sus respectivos bloques básicos.

El compilador construye paso a paso el IR correspondiente a cada figura o instrucción en. Por ejemplo, una declaración para crear un triángulo con color rojo y tamaño fijo será transformada en una serie de instrucciones que reservan memoria, almacenan los valores requeridos y luego llaman a una subrutina interna que representa gráficamente el triángulo.

Este código generado puede luego ser optimizado usando herramientas como opt de LLVM. El proyecto incluye dos carpetas: opt/, para optimizaciones automáticas generadas mediante comandos LLVM, y optManual/, donde se almacenan versiones del IR optimizadas manualmente, posiblemente para casos de prueba específicos como algoritmos de ordenamiento (e.g., selection\_optManual.ll).

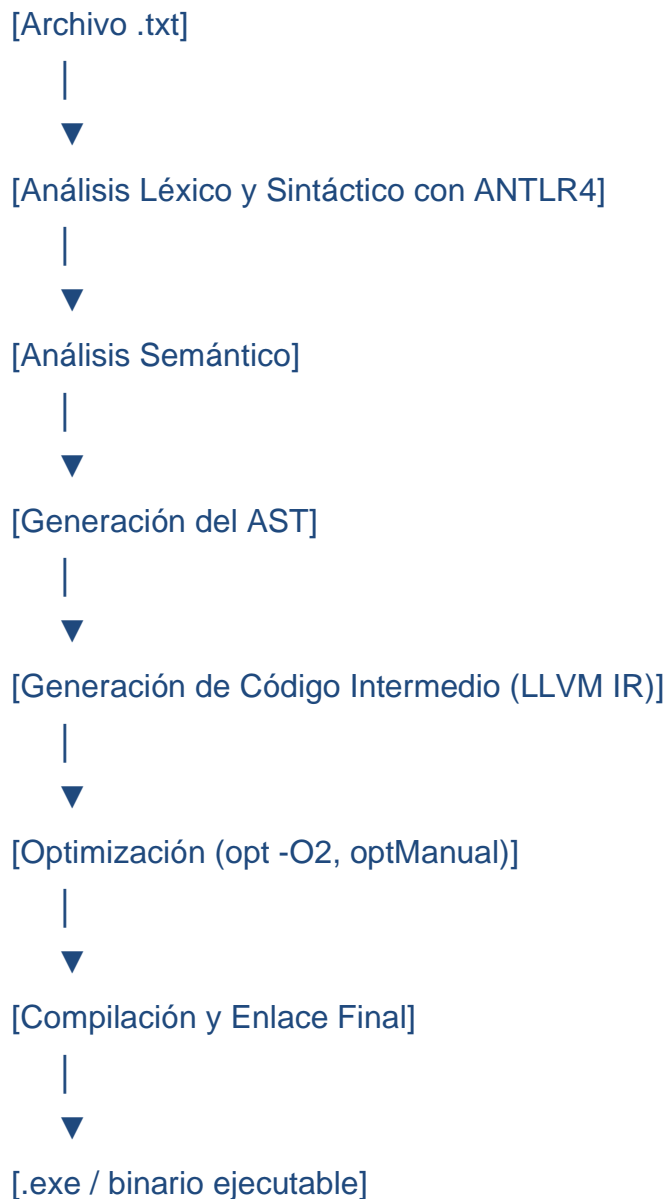
Finalmente, el código LLVM puede ser convertido a un archivo objeto (.o) y luego enlazado con otros componentes gráficos o librerías del sistema para producir un ejecutable final, alojado en el directorio ejecutableEXE/. Este flujo modular y bien

estructurado facilita no solo el mantenimiento del compilador, sino también su posible extensión en el futuro.

## Arquitectura General del Compilador y Optimizaciones

El compilador está organizado en múltiples módulos que trabajan en conjunto para traducir un programa fuente escrito en el lenguaje hasta su ejecución como binario. A continuación, se describe brevemente el flujo general del compilador y el papel de cada componente:

### Flujo de compilación:



### **Módulos principales:**

- **Gramatica.g4:** define la gramática del lenguaje y genera el analizador léxico y sintáctico usando ANTLR4.
- **listener.py:** contiene las acciones semánticas asociadas al recorrido del AST generado por ANTLR.
- **tabla\_simbolos.py:** administra la tabla de símbolos y los distintos ámbitos durante el análisis semántico.
- **generador\_ir.py:** traduce el AST validado en instrucciones LLVM IR.
- **optManual/ y opt/:** contienen las versiones optimizadas del código IR.
- **ejecutableEXE/:** almacena los ejecutables finales generados tras la compilación con llc y clang.

Esta arquitectura modular permite un desarrollo mantenible y escalable, y facilita futuras extensiones como nuevas primitivas gráficas, optimizaciones avanzadas o generación de código para múltiples plataformas.

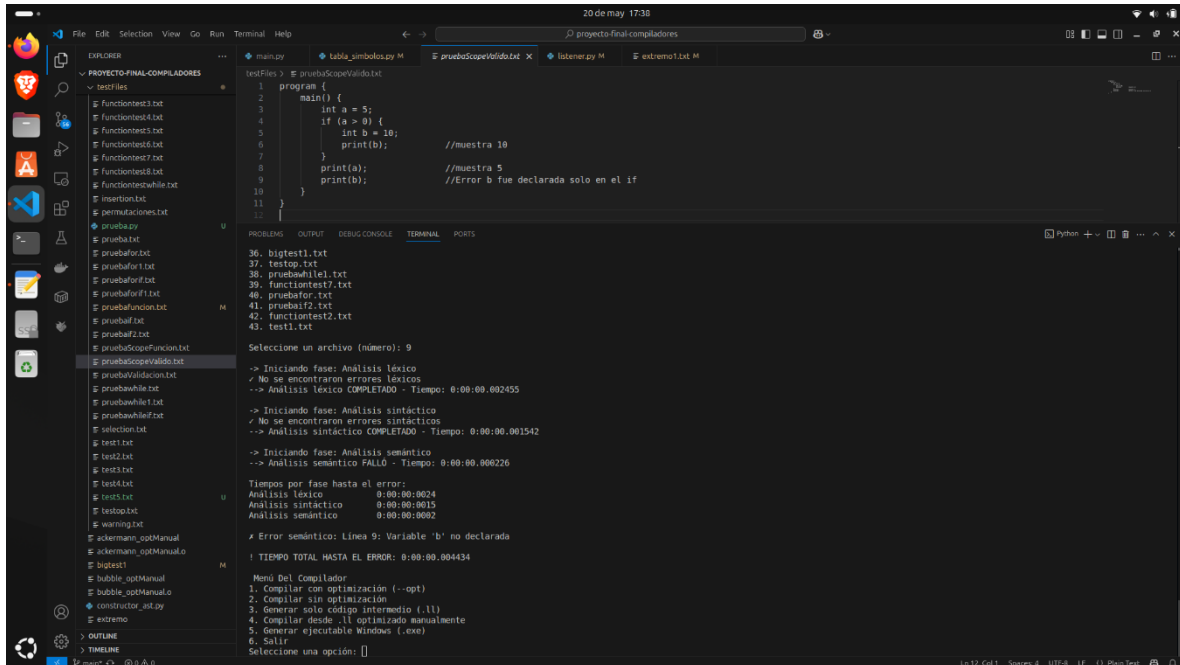
## Pruebas

### Prueba 1 - Ámbito de Funciones y Variables

Archivo: pruebaScopeValido.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/pruebaScopeValido.txt>

Resultado:



```
program {
2   main() {
3       int a = 5;
4       if (a > 0) {
5           int b = 10;
6           print(b); //muestra 10
7       }
8       print(a); //muestra 5
9       print(b); //Error b fue declarada solo en el if
10  }
11 }
12 }
```

Seleccione un archivo (número): 9

-> Iniciando fase: Análisis léxico  
✓ No se encontraron errores léxicos  
-> Análisis léxico COMPLETADO - Tiempo: 0:00:00.002455

-> Iniciando fase: Análisis sintáctico  
✓ No se encontraron errores sintácticos  
-> Análisis sintáctico COMPLETADO - Tiempo: 0:00:00.001542

-> Iniciando fase: Análisis semántico  
-> Análisis semántico FALLO - Tiempo: 0:00:00.000226

Tiempos por fase hasta el error:  
Análisis léxico 0:00:00:0024  
Análisis sintáctico 0:00:00:0015  
Análisis semántico 0:00:00:0002

# Error semántico: Línea 9: Variable 'b' no declarada

! TIEMPO TOTAL HASTA EL ERROR: 0:00:00.004434

Menú Del Compilador  
1. Compilar con optimización (--opt)  
2. Compilar sin optimización  
3. Generar solo código intermedio (.ll)  
4. Compilar desde .ll optimizado manualmente  
5. Generar ejecutable Windows (.exe)  
6. Salir

Seleccione una opción: []

El compilador devuelve un error semántico, porque se intenta utilizar la variable `b` que fue declarada dentro del ámbito de un `if` y se intenta utilizar fuera de ese ámbito.

Este tipo de errores semánticos detiene el proceso de compilación.

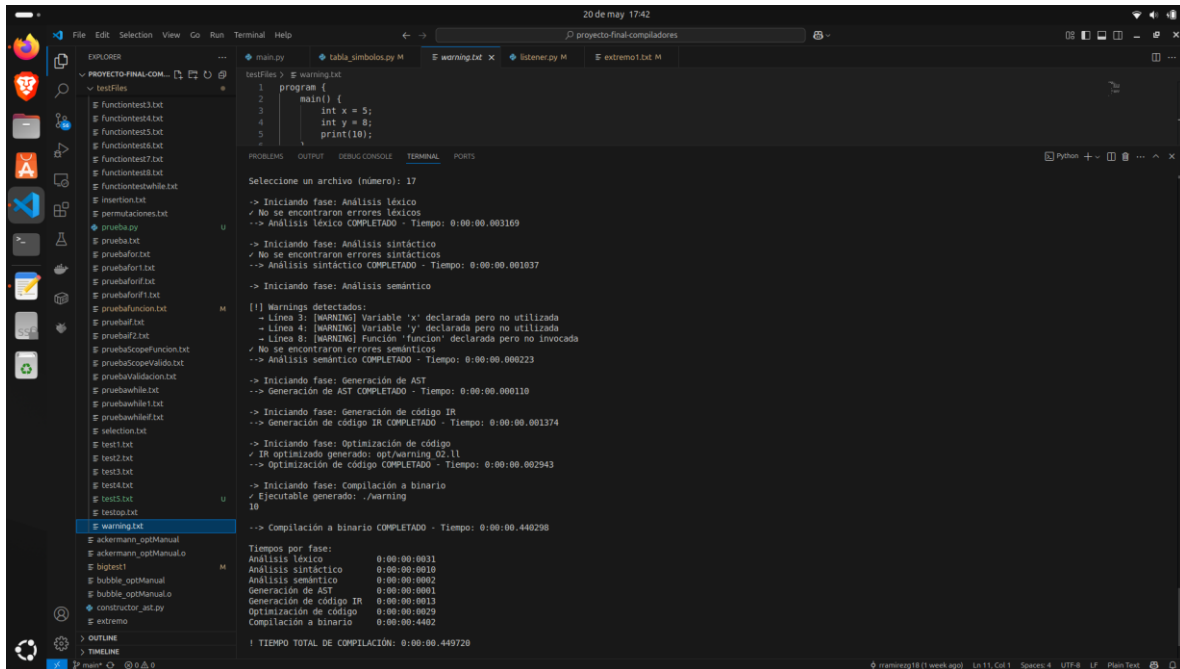
No se genera el resultado esperado.

## Prueba 2 - Warning - Variables y Funciones declaradas, pero no utilizadas

Archivo: warning.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/warning.txt>

Resultado:



```
testFiles > E warning.txt
1 program {
2   main() {
3     int x = 5;
4     int y = 8;
5     print(10);
6   }
7 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Seleccione un archivo (numero): 17
-> Iniciando fase: Analisis léxico
✓ No se encontraron errores léxicos
-> Analisis léxico COMPLETADO - Tiempo: 0:00:00.003169
-> Iniciando fase: Analisis sintáctico
✓ No se encontraron errores sintácticos
-> Analisis sintáctico COMPLETADO - Tiempo: 0:00:00.001037
-> Iniciando fase: Analisis semántico
[!] Warnings detectados:
- Linea 3: [WARNING] Variable 'x' declarada pero no utilizada
- Linea 4: [WARNING] Variable 'y' declarada pero no utilizada
- Linea 8: [WARNING] Función 'funcion' declarada pero no invocada
✓ No se encontraron errores semánticos
-> Analisis semántico COMPLETADO - Tiempo: 0:00:00.000223
-> Iniciando fase: Generación de AST
-> Generación de AST COMPLETADO - Tiempo: 0:00:00.000110
-> Iniciando fase: Generación de código IR
-> Generación de código IR COMPLETADO - Tiempo: 0:00:00.001374
-> Iniciando fase: Optimización de código
✓ IR optimizado generado: opt/warning.02.ll
-> Optimización de código COMPLETADO - Tiempo: 0:00:00.002943
-> Iniciando fase: Compilación a binario
✓ Ejecutable generado: ./warning
10
-> Compilación a binario COMPLETADO - Tiempo: 0:00:00.440720

Tiempos por fase:
Analisis léxico      0:00:00.0031
Analisis sintáctico  0:00:00.0010
Analisis semántico   0:00:00.0002
Generación de AST    0:00:00.0001
Generación de código IR 0:00:00.0013
Optimización de código 0:00:00.0029
Compilación a binario 0:00:00.4407

! TIEMPO TOTAL DE COMPILACIÓN: 0:00:00.440720
```

El compilador detecta variables y funciones que no fueron utilizadas y las reporta como warning.

Los Warning no detienen el proceso del compilador.

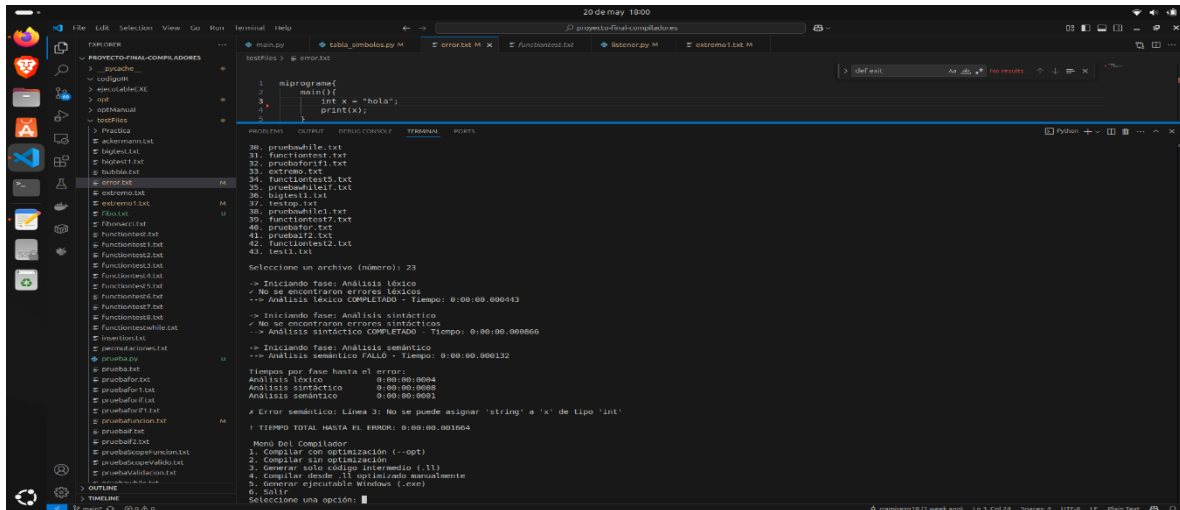
Se genera el resultado esperado.

## Prueba 3 - Validación de tipos

Archivo: error.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/warning.txt>

Resultado:



```
20 de may 18:00
main.py  tabla_symbol.py M  error.txt M  funciontest.txt  listener.py M  extramo1.txt M
testFiles > error.txt
1  miPrograma{
2      main(){
3          int x = "hola";
4          print(x);
5      }
6  }

Selecione un archivo (numero): 23
-> Iniciando fase: Analisis léxico
# No se encontraron errores léxicos
--> Analisis léxico COMPLETADO - Tiempo: 0:00:00.000443
-> Iniciando fase: Analisis sintáctico
# No se encontraron errores sintácticos
--> Analisis sintáctico COMPLETADO - Tiempo: 0:00:00.000806
-> Iniciando fase: Analisis semántico
--> Analisis semántico FALLÓ - Tiempo: 0:00:00.000132
Tiempo por fase hasta el error:
Analisis léxico      0:00:00.0004
Analisis sintáctico  0:00:00.0008
Analisis semántico   0:00:00.0001
# Error semántico: Línea 3: No se puede asignar 'string' a 'x' de tipo 'int'
! TIEMPO TOTAL HASTA EL ERROR: 0:00:00.001064

Menú Del Compilador
1. Compilar con optimización (-O2)
2. Compilar sin optimización
3. Generar solo código intermedio (.ll)
4. Compilar desde el optimizador manualmente
5. Generar ejecutable windows (.exe)
6. Salir
Selecione una opción: █
```

El compilador detecta y reporta un error porque se intenta asignar un valor de un tipo a una variable que se está declarando con otro tipo de dato diferente.

Este tipo de errores interrumpe el proceso de compilación.

No se genera el resultado esperado por el programa.

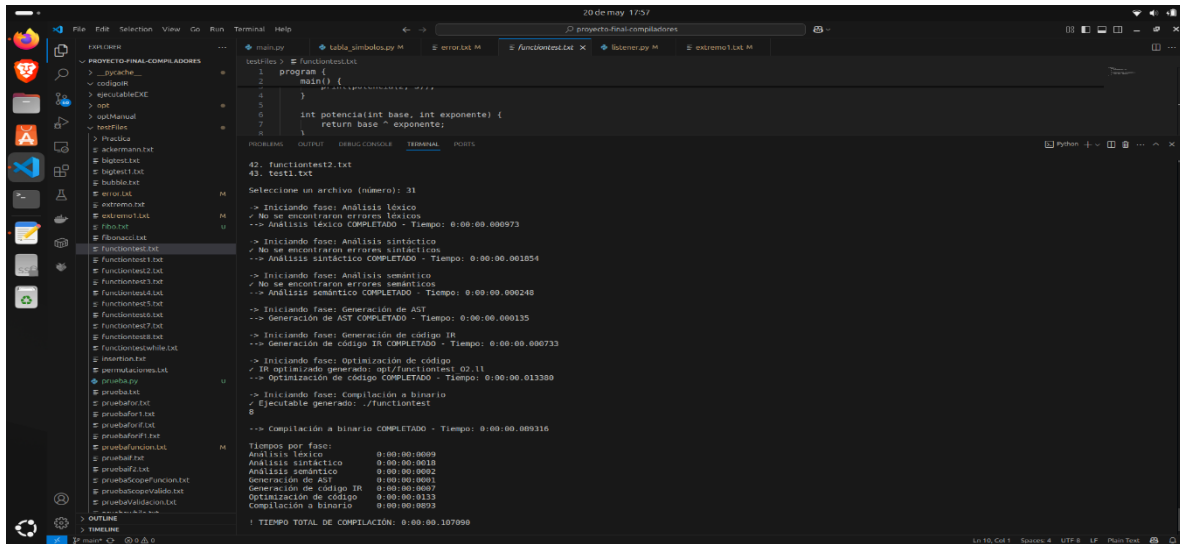


## Prueba 4 - Funciones

Archivo: functiontest.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/functiontest.txt>

Resultado:



```
42. functiontest2.txt
43. test1.txt

Seleccione un archivo (numero): 31

-> Iniciando fase: Analisis lexico
✓ No se encontraron errores lexicos
--> Analisis lexico COMPLETADO - Tiempo: 0:00:00.000973

-> Iniciando fase: Analisis sintactico
✓ No se encontraron errores sintacticos
--> Analisis sintactico COMPLETADO - Tiempo: 0:00:00.001854

-> Iniciando fase: Analisis semantico
✓ No se encontraron errores semanticos
--> Analisis semantico COMPLETADO - Tiempo: 0:00:00.000240

-> Iniciando fase: Generacion de AST
--> Generacion de AST COMPLETADO - Tiempo: 0:00:00.000135

-> Iniciando fase: Generacion de codigo IR
--> Generacion de codigo IR COMPLETADO - Tiempo: 0:00:00.000733

-> Iniciando fase: Optimizacion de codigo
✓ IR optimizado generado: ./functiontest.o2.11
--> Optimizacion de codigo COMPLETADO - Tiempo: 0:00:00.013300

-> Iniciando fase: Compilacion a binario
✓ Ejecutable generado: ./functiontest
8
--> Compilacion a binario COMPLETADO - Tiempo: 0:00:00.009310

Tiempos por fase:
Analisis lexico      0:00:00.000973
Analisis sintactico  0:00:00.001854
Analisis semantico   0:00:00.000240
Generacion de AST    0:00:00.000135
Generacion de codigo IR 0:00:00.000733
Optimizacion de codigo 0:00:00.013300
Compilacion a binario 0:00:00.009310

! TIEMPO TOTAL DE COMPILACION: 0:00:00.107990
```

El compilador procesa correctamente las funciones, sin generar errores.

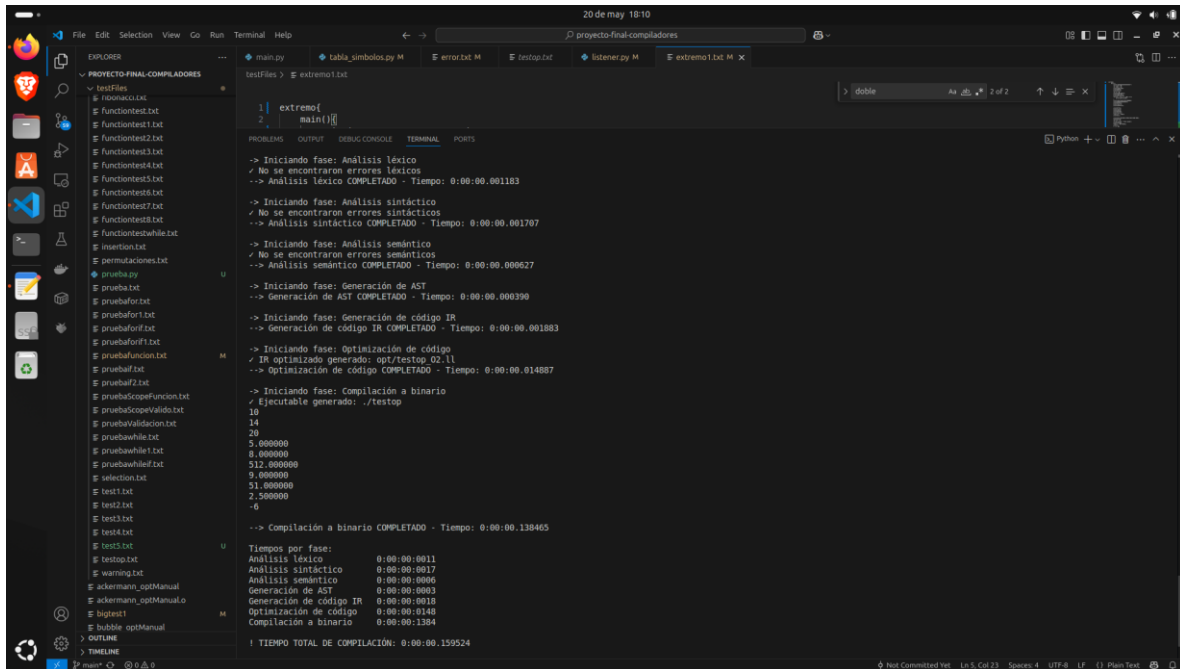
Muestra el resultado correcto del programa compilado.

## Prueba 5 - Expresiones Matemática

archivo: testop.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/testop.txt>

**Resultado:**



The screenshot shows a VS Code terminal window with the following output:

```
testFile > E extremo1.txt
1 | extremo{
2 |     main();
}

-> Iniciando fase: Análisis léxico
✓ No se encontraron errores léxicos
-> Análisis léxico COMPLETADO - Tiempo: 0:00:00.001183

-> Iniciando fase: Análisis sintáctico
✓ No se encontraron errores sintácticos
-> Análisis sintáctico COMPLETADO - Tiempo: 0:00:00.001707

-> Iniciando fase: Análisis semántico
✓ No se encontraron errores semánticos
-> Análisis semántico COMPLETADO - Tiempo: 0:00:00.000627

-> Iniciando fase: Generación de AST
-> Generación de AST COMPLETADO - Tiempo: 0:00:00.000390

-> Iniciando fase: Generación de código IR
-> Generación de código IR COMPLETADO - Tiempo: 0:00:00.001083

-> Iniciando fase: Optimización de código
✓ IR optimizado generado: opt/testop 02.ll
-> Optimización de código COMPLETADO - Tiempo: 0:00:00.014887

-> Iniciando fase: Compilación a binario
✓ Ejecutable generado: ./testop
10
14
20
5.000000
8.000000
512.000000
9.000000
51.000000
2.500000
-6

-> Compilación a binario COMPLETADO - Tiempo: 0:00:00.130465

Tiempos por fase:
Análisis léxico      0:00:00.0011
Análisis sintáctico  0:00:00.0017
Análisis semántico   0:00:00.0006
Generación de AST    0:00:00.0003
Generación de código IR 0:00:00.0018
Optimización de código 0:00:00.0148
Compilación a binario 0:00:00.1304

! TIEMPO TOTAL DE COMPILACIÓN: 0:00:00.150524
```

El compilador no detecta errores de ningún tipo durante el proceso de compilación.

Se evalúan correctamente las expresiones y se muestran los resultados del programa compilado.

## Prueba 6 - Comparación de Tiempos de Compilación

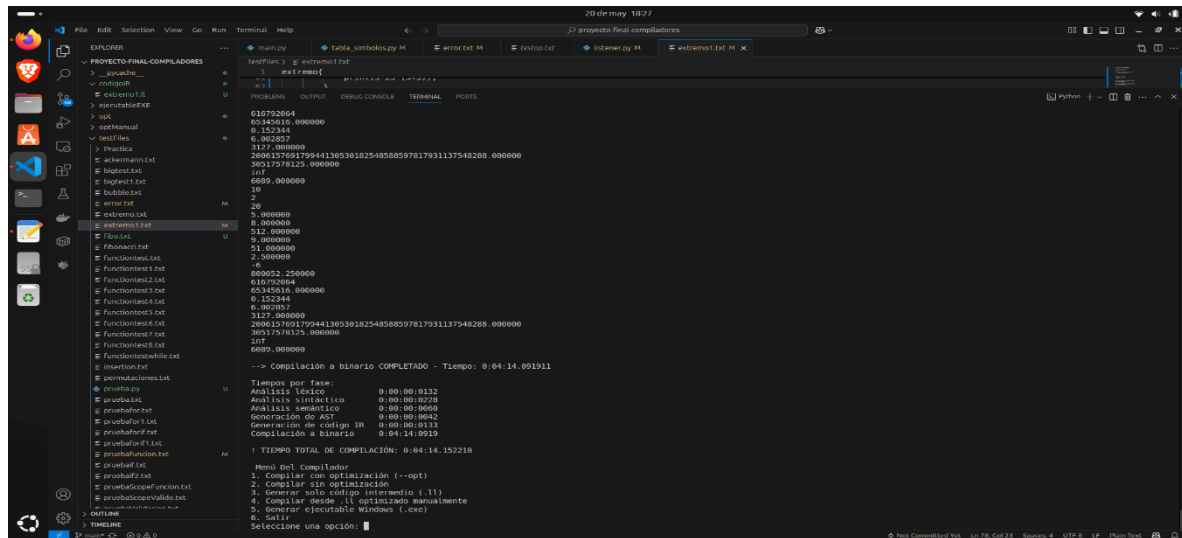
En esta prueba se comparan los tiempos de compilación de un programa sin aplicar optimizaciones contra el mismo programa, pero con una optimización opt O2.

Archivo: extremo1.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/extremo1.txt>

## Resultados

### Compilación sin aplicar optimización



```
20 de may 18:27
C:\proyectoFinalCompiladores> g++ -c extremo1.txt
extremo1.o
C:\proyectoFinalCompiladores> g++ extremo1.o
extremo1.exe
C:\proyectoFinalCompiladores>

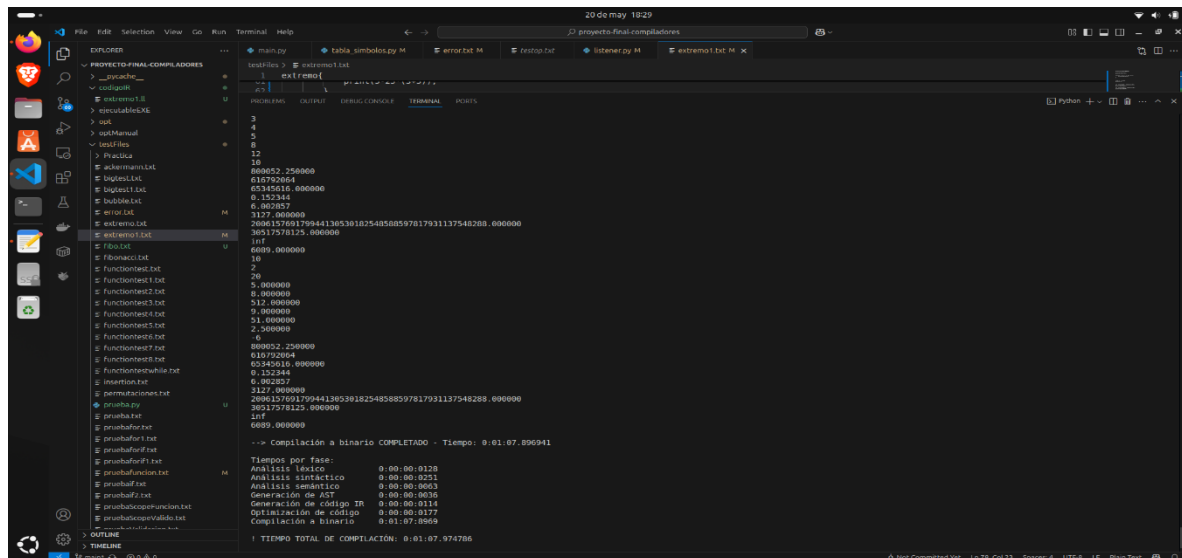
--> Compilación a binario COMPLETADO - Tiempo: 0:04:14.891911

Tiempo por fase:
Análisis léxico      0:00:00:0132
Análisis sintáctico 0:00:00:0278
Análisis semántico  0:00:00:0609
Generación de AST   0:00:00:0642
Generación de código IR 0:00:00:0173
Compilación a binario 0:04:14:0919

! TIEMPO TOTAL DE COMPILACIÓN: 0:04:14.152219

Menú Del Compilador
1. Compilar con optimización (--opt)
2. Compilar sin optimización
3. Generar solo código intermedio (.ll)
4. Compilar desde .ll optimizado manualmente
5. Generar ejecutable Windows (.exe)
6. Salir
Seleccione una opción: 1
```

### Compilación aplicando opt O2



```
20 de may 18:29
C:\proyectoFinalCompiladores> g++ -O2 -c extremo1.txt
extremo1.o
C:\proyectoFinalCompiladores> g++ extremo1.o
extremo1.exe
C:\proyectoFinalCompiladores>

--> Compilación a binario COMPLETADO - Tiempo: 0:01:07.896941

Tiempo por fase:
Análisis léxico      0:00:00:0128
Análisis sintáctico 0:00:00:0251
Análisis semántico  0:00:00:0663
Generación de AST   0:00:00:0636
Generación de código IR 0:00:00:0114
Optimización de código 0:00:00:0177
Compilación a binario 0:01:07:8969

! TIEMPO TOTAL DE COMPILACIÓN: 0:01:07.974786
```

**Conclusión:** Se puede notar que el tiempo de compilación se reduce significativamente de un archivo sin opt a un archivo con opt O2.

## Prueba 7 - Generación de ejecutable .exe para sistema operativo Windows

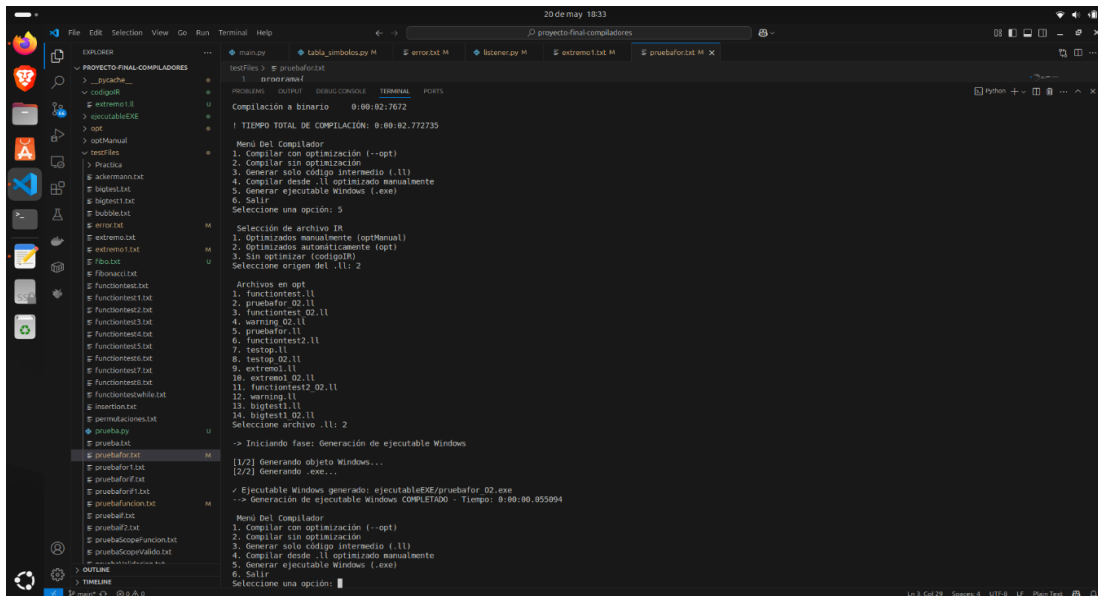
Archivo: pruebafor.txt

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/testFiles/pruebafor.txt>

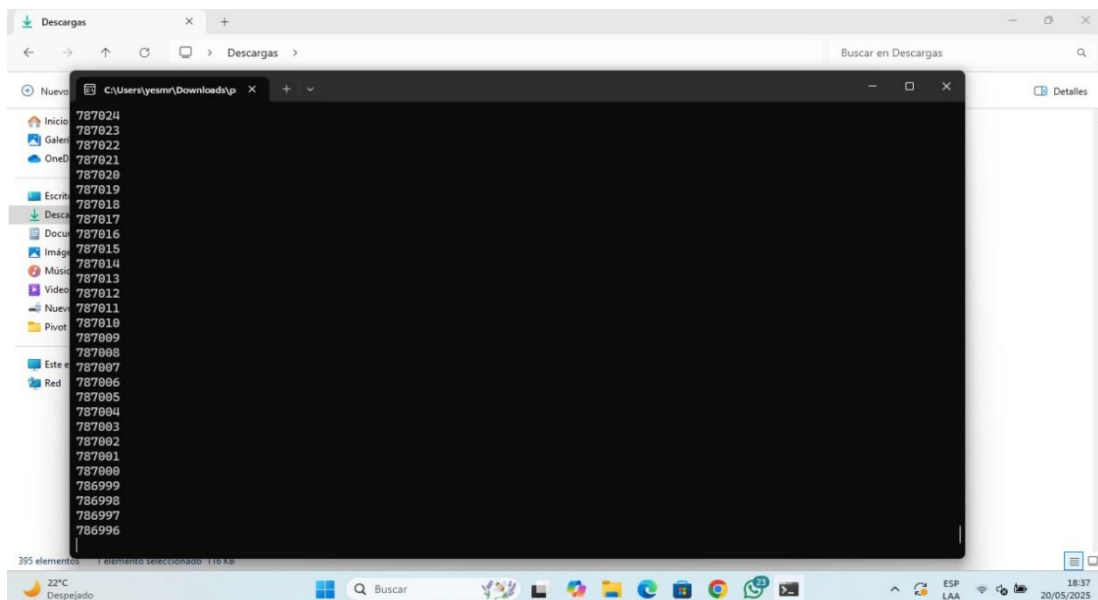
Ejecutable:

<https://github.com/rramirezg18/ProyectoFinalCompiladores/blob/main/ejecutableEXE/pruebafor.exe>

Resultado:



Se genera correctamente el archivo .exe ejecutable para Windows.



## Conclusiones

- **Integración de componentes del compilador**

La implementación del compilador permitió consolidar el conocimiento adquirido sobre las distintas fases del proceso de compilación, desde el análisis léxico y sintáctico con ANTLR4, hasta la generación de código intermedio en LLVM IR. Cada componente fue integrado de forma modular, permitiendo una arquitectura clara, mantenible y extensible.

- **Validación semántica robusta**

Gracias a la construcción de una tabla de símbolos jerárquica y al recorrido del AST mediante el patrón listener, se logró implementar un sistema de validación semántica efectivo, capaz de detectar errores comunes como el uso de variables no declaradas, conflictos de tipos y problemas de alcance, contribuyendo a una mayor confiabilidad del código fuente analizado.

- **Eficiencia mediante generación y optimización de IR**

La utilización de LLVM IR como lenguaje de código intermedio resultó ser una estrategia eficaz, al facilitar la generación de código portable y optimizable. Las pruebas demostraron que aplicar técnicas de optimización, como `opt -O2`, mejora significativamente el rendimiento sin comprometer la funcionalidad del programa.

- **Valor académico y proyección futura**

El proyecto representa no solo un logro técnico, sino también un ejercicio académico enriquecedor que demuestra la capacidad de diseñar e implementar un lenguaje de programación funcional. Además, su diseño modular abre posibilidades para futuras mejoras, como agregar manejo de errores más detallado, nuevas primitivas gráficas o una interfaz de usuario.