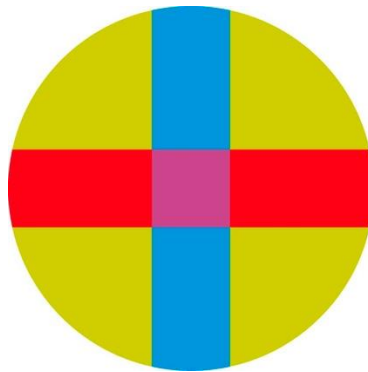


UNIVERSIDAD SAN PABLO - CEU

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA DE SISTEMAS DE INFORMACIÓN



TRABAJO FIN DE GRADO

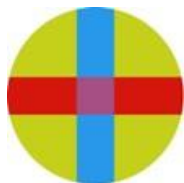
Desarrollo de un Compilador

Development of a Compiler

Autor: Miguel Merino Plaza

Tutor: Rafael Núñez Hervás

Junio 2024



UNIVERSIDAD SAN PABLO-CEU

ESCUELA POLITÉCNICA SUPERIOR

División de Ingeniería

Calificación del Trabajo Fin de Grado

Datos del alumno

NOMBRE: MIGUEL MERINO PLAZA

Datos del Trabajo

TÍTULO DEL PROYECTO: DESARROLLO DE UN COMPILADOR

Tribunal calificador

PRESIDENTE:

FDO.:

SECRETARIO:

FDO.:

VOCAL:

FDO.:

Reunido este tribunal el ____/____/_____, acuerda otorgar al Trabajo Fin de Grado presentado por D./Dña. _____ la calificación de _____

Resumen

Para este Trabajo Fin de Grado se ha realizado el desarrollo de un compilador. Este desarrollo implica varias etapas fundamentales:

En primer lugar, se comienza con la creación de una tabla de símbolos. Esta tabla almacena información sobre identificadores utilizados en el programa, como variables y funciones. En un primer momento, esta tabla está enlazada al analizador más básico, es decir, el léxico. Pero luego evoluciona para enlazarse al analizador más complejo, es decir, el sintáctico donde están el resto de los analizadores.

En segundo lugar, se desarrolla un analizador morfológico o léxico, que examina la estructura de las palabras del código fuente para identificar sus partes y clasificarlas gramaticalmente. Por lo tanto, se realiza un análisis de un fichero dado que contiene el código fuente y se divide en tokens significativos.

En tercer lugar, se desarrolla un analizador sintáctico para estructurar estos tokens según la gramática del lenguaje de programación. En un primer momento, se desarrolla únicamente el analizador sintáctico, pero luego se le incorpora la gestión de errores, el analizador semántico y la generación de código.

En cuarto lugar, se lleva a cabo un análisis semántico para verificar la coherencia del código en términos de significado y contexto. Para un mejor desarrollo se enlaza con el analizador sintáctico mediante la herramienta Bison.

En quinto lugar, se genera el código intermedio, que es una representación abstracta del programa, que se optimiza para mejorar su eficiencia. Esta optimización implica técnicas como la eliminación de código que no se utiliza, la eliminación de código duplicado o la reorganización de expresiones para reducir la complejidad. Este código se genera mediante reglas implementadas en el analizador sintáctico mediante Bison.

Finalmente, se genera el código objeto (máquina) a partir del código intermedio optimizado.

Palabras Clave

Tabla de símbolos

Analizador léxico

Análisis sintáctico

Análisis semántico

Código fuente

Código intermedio

Código objeto

Gramática

Compilador

Tokens

Bison

Flex

Abstract

For this Final Degree Project, the development of a compiler was carried out. This development involves several basic steps:

First, it begins with the creation of a symbol table. This table stores information about identifiers used in the program, such as variables and functions. Initially, this table is linked to the most basic parser, the lexicon. But then it evolves to be linked to our most complex parser, the syntactic parser, where the rest of the parsers are located.

Secondly, a morphological or lexical analyzer is developed, which examines the structure of the words in the source code to identify their parts and classify them grammatically. In this way, a given file containing source code is analyzed and broken down into meaningful tokens.

Third, a syntactic parser is developed to structure these tokens according to the grammar of the programming language. Initially, only the syntactic parser is developed, but then error handling, semantic parser and code generation are integrated.

Fourth, a semantic analysis is developed to verify the coherence of the code in terms of meaning and context. For better development, it is linked to the syntactic analyzer using the Bison tool.

Fifth, the intermediate code, which is an abstract representation of the program, is generated and optimized to improve its efficiency. This optimization includes techniques such as removing unused code, eliminating duplicate code, or reorganizing expressions to reduce complexity. This code is generated by rules implemented in the parser using Bison.

Finally, the object (machine) code is generated from the optimized intermediate code.

Keywords

Symbol table

Lexical analyzer

Syntactic analysis

Semantic analysis

Source code

Intermediate code

Object code

Grammar

Compiler

Tokens

Bison

Flex

Índice de contenidos

Capítulo 1 Introducción	8
1.1 Contexto del TFG	8
1.2 Objetivos.....	10
1.3 Organización del trabajo.....	11
Capítulo 2 Gestión del proyecto	12
2.1 Modelo de ciclo de vida.....	12
2.2 Planificación.....	13
Capítulo 3 Análisis.....	19
3.1 Especificación de requisitos.....	19
3.2 Análisis de los Casos de Uso	21
Capítulo 4 Diseño e implementación	23
4.1 Arquitectura del sistema	23
4.1.1 Arquitectura física	24
4.1.2 Arquitectura lógica	24
4.2 Diseño de la interfaz de usuario	24
4.3 Estructura de clases	25
4.4 Construcción del entorno	28
4.5 Referencia al repositorio de software	30
Capítulo 5 Desarrollo del Sistema	31
5.1 Tabla de Símbolos.....	31
5.2 Analizador Léxico/Morfológico.....	32
5.3 Analizador Sintáctico	34
5.4 Analizador Semántico	36
5.5 Analizadores.....	41
5.6 Generación de código	42
5.7 Interfaz de usuario.....	46
Capítulo 6 Conclusiones y líneas futuras	47
6.1 Conclusiones	47
6.2 Líneas futuras.....	48
Bibliografía.....	49
Anexo I – Código	50
Anexo I.I – Código Makefile	50
Anexo I.II – Código Tabla de Símbolos	51
Anexo I.III – Código Analizador Léxico	52
Anexo I.IV – Código Analizador Sintáctico	54
Anexo I.V – Código Analizador Semántico	72
Anexo I.VI – Código Generación de Código	72

Índice de figuras

Figura 1. Imagen de un compilado, https://www.campusmvp.es/recursos/post/una-introduccion-a-los-compiladores-como-hablar-con-una-computadora-pre-siri.aspxr ...	8
Figura 2. Imagen de Código Fuente a Código Máquina, https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=392:la-maquina-virtual-java-jvm-o-java-virtual-machine-compilador-e-interprete-bytecode-cu00611b&catid=68&Itemid=188	9
Figura 3. Imagen de un modelo de ciclo de vida incremental, https://ingenieriadesoftwaretdea.weebly.com/ciclo-de-vida-desarrollo-incremental.html	12
Figura 4. Imagen de entrega por incrementos dentro del modelo incremental, https://www.researchgate.net/figure/Figura-10-Modelo-de-proceso-incremental-Fuente_fig6_326571456	13
Figura 5. Imagen de fases para completar TFG	18
Figura 6. Casos de uso Desarrollador - Cliente (Diagrams.net)	22
Figura 7. Arquitectura Sistema (Diagrams.net)	23
Figura 8. Frontend y Backend (Diagrams.net)	23
Figura 9. Capas de nuestro sistema (Diagrams.net)	24
Figura 10. Arquitectura física (Diagrams.net)	24
Figura 11. Arquitectura Lógica (Diagrams.net)	24
Figura 12. Interfaz de usuario (Interfaz programa)	25
Figura 13. Respuesta de error compilador (Interfaz programa)	25
Figura 14. Respuesta de éxito compilador (Interfaz programa)	25
Figura 15. Esquema General Compilador	26
Figura 16. Estructura src (index.html)	28
Figura 17. Diagrama de ejemplo clase main (index.html)	28
Figura 18. Terminal msys2 UCRT64	29
Figura 19. Terminal UCRT64: Ejemplo instalación herramienta Flex	29
Figura 20. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta Flex	29
Figura 21. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta opcional graphviz	29
Figura 22. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta opcional doxygen	30
Figura 23. Terminal UCRT64: Ejemplo verificación de funcionamiento correcto de la tabla de símbolos	32
Figura 24. Terminal UCRT64: Ejemplo verificación de funcionamiento correcto del analizador léxico	33
Figura 25. Terminal UCRT64: Ejemplo verificación de analizador léxico para Error – token no X reconocido en línea Y	34
Figura 26. Terminal UCRT64: Ejemplo verificación de funcionamiento del analizador sintáctico	36

Figura 27. Terminal UCRT64: Ejemplo verificación de analizador sintáctico para Error de sintaxis – Error de sintaxis en línea X	36
Figura 28. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X previamente declarado	37
Figura 29. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X de clase previamente declarado	38
Figura 30. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X de función previamente declarado	38
Figura 31. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador X no declarado previamente	39
Figura 32. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador de función X no declarado previamente	39
Figura 33. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador de función X no declarado previamente	40
Figura 34. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error tipo de retorno incompatible con el tipo de función.	40
Figura 35. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error tipo de variable incompatible con otro tipo de otra variable.	41
Figura 36. Terminal UCRT64: Ejemplo verificación de analizador semántico funciona de forma correcta	41
Figura 37. Terminal UCRT64: Ejemplo verificación analizadores funcionan de forma correcta.....	42
Figura 38. Terminal UCRT64: Ejemplo verificación generación de código intermedio funciona de forma correcta	43
Figura 39. Terminal UCRT64: Ejemplo verificación generación de código intermedio funciona y se guarda en fichero de forma correcta	43
Figura 40. Terminal UCRT64: Ejemplo verificación generación de código objeto funciona y se guarda en fichero de forma correcta	45
Figura 41. Terminal UCRT64: Ejemplo verificación compilador funciona de forma correcta.....	45
Figura 42. Terminal UCRT64: Ejemplo verificación de interfaz de usuario	46
Figura 43. Terminal UCRT64: Ejemplo verificación de interfaz de usuario – Selección fichero .alfa a compilar	46

Índice de tablas

Tabla 1. Tabla con la planificación del TFG.....	18
Tabla 2. Requisitos Funcionales.....	21

Capítulo 1

Introducción

1.1 Contexto del TFG

Los compiladores son de gran importancia en el campo de la informática y la programación. El papel de los compiladores es fundamental en estos ámbitos ya que traducen el código fuente de un lenguaje de programación a un código ejecutable, facilitando así la creación de software complejo y funcional.

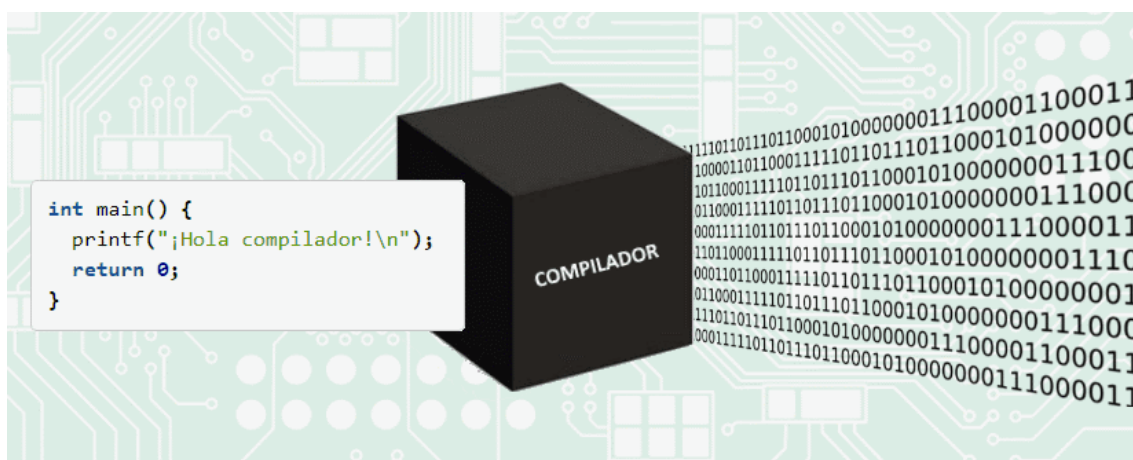


Figura 1. Imagen de un compilado, <https://www.campusmvp.es/recursos/post/una-introduccion-a-los-compiladores-como-hablar-con-una-computadora-pre-siri.aspx>

La decisión de abordar el desarrollo de un compilador surge de varios factores fundamentales. Por un lado, existe un interés en comprender en profundidad el funcionamiento interno de los compiladores y explorar las técnicas y algoritmos involucrados en su diseño y desarrollo. Esta comprensión del funcionamiento del compilador proporciona la capacidad de crear un compilador para cualquier lenguaje y gramática, y permite mejorar en la compilación de un programa de lenguaje cualquiera. Por otro lado, cabe destacar la importancia de los compiladores en la industria del software. Un compilador eficiente y preciso puede marcar la diferencia en

términos de rendimiento y calidad del software final. Por lo tanto, este proyecto busca contribuir al campo de la compilación mediante la creación de un compilador funcional.

Este proyecto implica la implementación de analizadores, generadores de código y un software capaz de traducir el código fuente de un lenguaje de programación específico a un código objeto. Durante este proyecto se exploran las diferentes etapas del proceso de compilación, desde el análisis léxico y sintáctico hasta la generación de código intermedio y su transformación a código máquina.

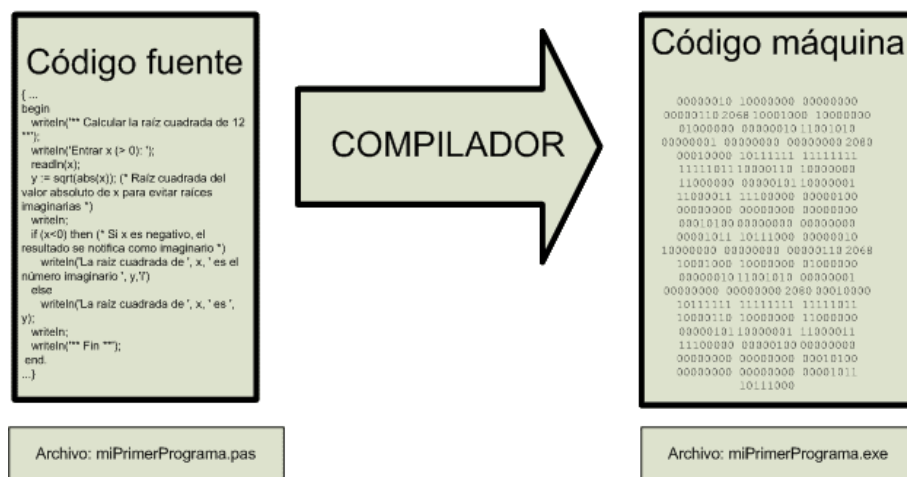


Figura 2. Imagen de Código Fuente a Código Máquina,
https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=392:la-maquina-virtual-java-jvm-o-java-virtual-machine-compiler-e-interprete-bytecode-cu00611b&catid=68&Itemid=188

La relevancia técnica de este TFG radica en la importancia de los compiladores en el desarrollo de software. Los compiladores son fundamentales para traducir código de alto nivel a lenguaje máquina, permitiendo a los programadores escribir programas en un formato comprensible y portable. El desarrollo de un compilador proporciona una comprensión profunda de los conceptos de compilación, mejora las habilidades de diseño de software y contribuye al avance en técnicas de optimización y análisis de código, aspectos cruciales en la ingeniería de software moderna.

Además, cabe destacar que, con el desarrollo de nuevas tecnologías como la computación cuántica, la inteligencia artificial y el cómputo distribuido, se necesitan

compiladores especializados en estas plataformas, lo cual requiere un mayor conocimiento del compilador y su desarrollo.

1.2 Objetivos

El objetivo principal es desarrollar un compilador funcional que analice y traduzca el código fuente de un lenguaje de programación .alfa (*Ver Anexo II*) específico a código objeto (máquina).

En cuanto a los subobjetivos, nos encontramos tareas para lograr el objetivo principal:

- Investigar y comprender en profundidad los fundamentos teóricos de la compilación, incluyendo el análisis léxico, sintáctico y semántico.
- Construir una tabla de símbolos para almacenar información sobre identificadores como variables y funciones.
- Diseñar e implementar un analizador léxico y morfológico capaz de dividir el código fuente en tokens significativos, examinar la estructura de las palabras del código fuente y clasificarlas gramaticalmente.
- Desarrollar un analizador sintáctico que pueda estructurar los tokens según la gramática del lenguaje de programación objetivo. Este analizador realiza un análisis en árbol mediante la creación de nodos.
- Implementar un analizador semántico para verificar la coherencia del código en términos de significado y contexto.
- Generar un código intermedio optimizado que sirva como representación abstracta del programa.
- Generar código objeto específico para la plataforma de destino a partir del código intermedio optimizado.
- Probar el compilador con diferentes ficheros .alfa para garantizar su correcto funcionamiento y precisión.
- Documentar adecuadamente el proceso de desarrollo, incluyendo el diseño, la implementación y los resultados obtenidos.

1.3 Organización del trabajo

En el capítulo 1, se realiza la introducción al TFG “Desarrollo de un compilador”. Se comentan el contexto del TFG, los objetivos que se buscan cumplir y cómo se va a organizar el trabajo en los diferentes capítulos.

En el capítulo 2, se lleva a cabo la gestión del proyecto donde se detalla el modelo de ciclo de vida empleado, la planificación establecida y el presupuesto asignado, en este caso se omitirá este presupuesto ya que es todo gratuito para llevar a cabo el proyecto de manera eficiente.

En el capítulo 3 se realiza un análisis que profundiza en la especificación de requisitos del compilador, analiza los casos de uso y evalúa los aspectos de seguridad, si corresponde.

En el capítulo 4, se lleva a cabo el diseño e implementación, se describe la arquitectura del sistema, el diseño de datos, la interfaz de usuario y se presenta el entorno de construcción utilizado. Además, se hace referencia al repositorio de software donde se aloja el código fuente del compilador.

En el capítulo 5, se lleva a cabo el desarrollo de cada uno de los componentes del compilador (analizadores, generadores de código...)

Finalmente, en el capítulo 6, se exponen las conclusiones obtenidas tras el desarrollo del proyecto y se proponen posibles líneas de trabajo futuro para continuar mejorando y ampliando el compilador desarrollado.

En los anexos se encuentran los distintos códigos para el desarrollo de las clases como `lex.l`, `syntax.y...` para la implementación del compilador.

Capítulo 2

Gestión del proyecto

2.1 Modelo de ciclo de vida

Para la realización del TFG se ha optado por seguir un modelo de ciclo de vida incremental.

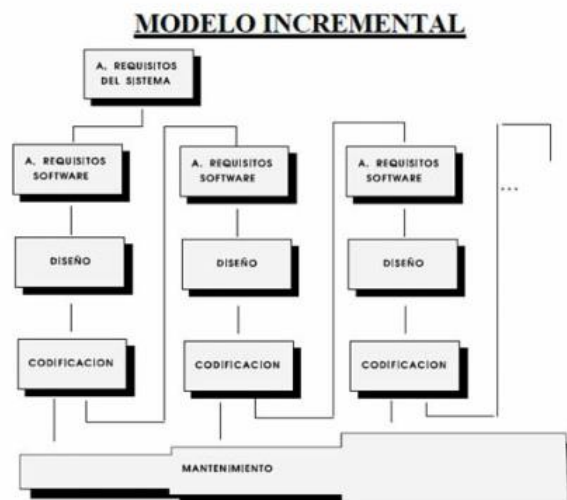


Figura 3. Imagen de un modelo de ciclo de vida incremental,
<https://ingenieriadesoftwaretdea.weebly.com/ciclo-de-vida-desarrollo-incremental.html>

El modelo de ciclo de vida incremental permite dividir el proyecto en una serie de etapas bien definidas, cada una de las cuales produce un entregable funcional. Esto facilita la gestión del proyecto y permite una mayor flexibilidad para adaptarse a que puedan surgir a lo largo del desarrollo del compilador.

Este enfoque se basa en dividir el proyecto en pequeñas iteraciones (incrementos), donde cada iteración agrega nuevas funcionalidades al producto en desarrollo. Cada iteración comienza con la planificación de las tareas a realizar, seguida de la implementación, pruebas y entrega de las nuevas funcionalidades.

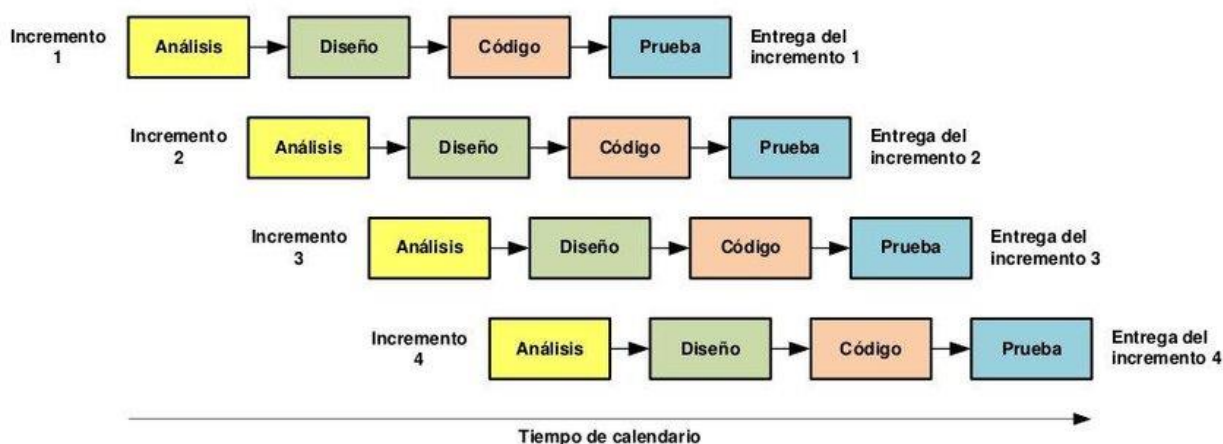


Figura 4. Imagen de entrega por incrementos dentro del modelo incremental,
https://www.researchgate.net/figure/Figura-10-Modelo-de-proceso-incremental-Fuente_fig6_326571456

Se ha elegido este enfoque incremental ya que permite obtener retroalimentación temprana del cliente (en este caso del tutor del TFG), lo que facilita la detección y corrección de posibles problemas a medida que avanza el proyecto.

Además, al final de cada iteración, se produce un entregable funcional que puede ser evaluado y probado, lo que contribuye a un mayor control del progreso del proyecto.

Por último, este modelo permite abordar gradualmente las diversas etapas del proceso de desarrollo, desde el análisis y diseño hasta la implementación, pruebas y entrega final del producto.

2.2 Planificación

Tarea 1: Investigación y Diseño

Subtarea 1: Investigación sobre compiladores existentes

Descripción: Investigación de compiladores existentes en la actualidad para comprender sus características y funcionalidades.

Fecha de inicio estimada: /02/2024

Fecha de finalización estimada: /02/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 20 horas

Subtarea 2: Definición de requisitos

Descripción: Identificar y documentar requisitos funcionales y no funcionales del compilador.

Fecha de inicio estimada: /02/2024

Fecha de finalización estimada: - (surgen nuevos requisitos a lo largo del proyecto)

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 30 horas

Subtarea 3: Diseño de la arquitectura

Descripción: Definir arquitectura del compilador.

Fecha de inicio estimada: /02/2024

Fecha de finalización estimada: /03/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 25 horas

Subtarea 4: Documentación de la Fase 1

Descripción: Documentar tareas de la fase 1 en la memoria del TFG.

Fecha de inicio estimada: /02/2024

Fecha de finalización estimada: /03/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 10 horas

Hito: Finalización Fase 1

Descripción: Finalización de todas las tareas establecidas para completar la fase 1 de forma correcta.

Tarea 2: Desarrollo e Implementación**Subtarea 1: Tabla de Símbolos**

Descripción: Desarrollo de una tabla de símbolos.

Fecha de inicio estimada: /03/2024

Fecha de finalización estimada: - (Este archivo se tiene que ir cambiando a lo largo que se desarrollan el resto de componentes)

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 25 horas

Subtarea 2: Analizador Léxico

Descripción: Desarrollar el componente del compilador encargado de dividir el código fuente en tokens significativos.

Fecha de inicio estimada: /04/2024

Fecha de finalización estimada: /06/2024

Recursos necesarios: Herramientas de programación y Entorno.

Estimación de esfuerzo: 50 horas (léxico + morfológico)

Subtarea 3: Analizador Morfológico

Descripción: Desarrollar el componente del compilador encargado de examinar la estructura de las palabras del código fuente y clasificarlas gramaticalmente.

Fecha de inicio estimada: /04/2024

Fecha de finalización estimada: /06/2024

Recursos necesarios: Herramientas de programación, Alfa, Bison, Entorno.

Estimación de esfuerzo: 0 horas (El total del analizador léxico y morfológico es 50)

Subtarea 4: Analizador Sintáctico

Descripción: Desarrollar el componente del compilador encargado de estructurar los tokens según la gramática del lenguaje de programación objetivo.

Fecha de inicio estimada: /04/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Herramientas de programación, Alfa, Bison, Entorno.

Estimación de esfuerzo: +300 horas

Subtarea 5: Analizador Semántico

Descripción: Desarrollar el componente del compilador encargado de verificar la coherencia del código en términos de significado y contexto.

Fecha de inicio estimada: /06/2024

Fecha de finalización estimada: /07/2024

Recursos necesarios: Herramientas de programación, Alfa, Bison, Entorno.

Estimación de esfuerzo: 70 horas

Subtarea 6: Código Intermedio

Descripción: Desarrollar el componente del compilador encargado de generar una representación abstracta del programa.

Fecha de inicio estimada: /07/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Herramientas de programación.

Estimación de esfuerzo: 70 horas

Subtarea 7: Código Objeto

Descripción: Desarrollar el componente del compilador encargado de generar código objeto específico para la plataforma de destino.

Fecha de inicio estimada: /08/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Herramientas de programación.

Estimación de esfuerzo: 50 horas

Subtarea 8: Pruebas

Descripción: Realizar pruebas (introduciendo ficheros) para verificar el correcto funcionamiento del compilador y su conformidad con los requisitos establecidos.

Fecha de inicio estimada: /04/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Herramientas de programación.

Estimación de esfuerzo: 50 horas

Subtarea 9: Documentación de la Fase 2

Descripción: Documentar las tareas de la fase 2 en la memoria del TFG

Fecha de inicio estimada: /04/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 80 horas

Hito: Finalización Fase 2

Descripción: Finalización de todas las tareas establecidas para completar la fase 2 de forma correcta.

Tarea 3: Optimización y Otros

Subtarea 1: Optimización

Descripción: Optimización general.

Fecha de inicio estimada: /08/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Herramientas de programación.

Estimación de esfuerzo: 20 horas

Subtarea 2: Otras mejoras

Descripción: Revisión de posibles mejoras.

Fecha de inicio estimada: /08/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 20 horas

Subtarea 3: Otras comprobaciones

Descripción: Otras comprobaciones finales para dar por concluida la realización del TFG

Fecha de inicio estimada: /08/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 10 horas

Subtarea 4: Documentación de la Fase 3

Descripción: Documentar toda las tareas de la fase 3 en la memoria del TFG

Fecha de inicio estimada: /08/2024

Fecha de finalización estimada: /08/2024

Recursos necesarios: Acceso a bibliografía y recursos en línea.

Estimación de esfuerzo: 25 horas

Hito: Finalización Fase 3

Descripción: Finalización de todas las tareas establecidas para completar la fase 3 de forma correcta.

Hito: Finalización TFG

Descripción: Finalización de todas las tareas establecidas para dar por concluida la realización del TFG, Desarrollo de un Compilador, de forma correcta.

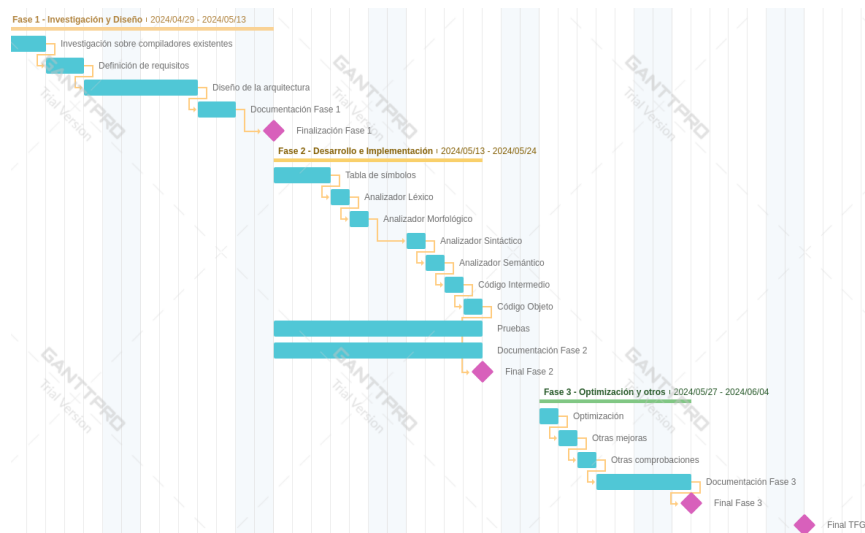


Figura 5. Imagen de fases para completar TFG

Tabla 1. Tabla con la planificación del TFG

1	<input type="checkbox"/> Fase 1 - Investigación ...		
1.1	Investigación sobre...	● Abierto	sin asignar
1.2	Definición de requi...	● Abierto	sin asignar
1.3	Diseño de la archit...	● Abierto	sin asignar
1.4	Documentación Fa...	● Abierto	sin asignar
1.5	Finalización Fase 1	● Abierto	sin asignar
2	<input type="checkbox"/> Fase 2 - Desarrollo e I...		
2.1	Tabla de símbolos	● Abierto	sin asignar
2.2	Analizador Léxico	● Abierto	sin asignar
2.3	Analizador Morfoló...	● Abierto	sin asignar
2.4	Analizador Sintáctico	● Abierto	sin asignar
2.5	Analizador Semánti...	● Abierto	sin asignar
2.6	Código Intermedio	● Abierto	sin asignar
2.7	Código Objeto	● Abierto	sin asignar
2.8	Pruebas	● Abierto	sin asignar
2.9	Documentación Fa...	● Abierto	sin asignar
2.10	Final Fase 2	● Abierto	sin asignar
3	<input type="checkbox"/> Fase 3 - Optimización ...		
3.1	Optimización	● Abierto	sin asignar
3.2	Otras mejoras	● Abierto	sin asignar
3.3	Otras comprobacio...	● Abierto	sin asignar
3.4	Documentación Fa...	● Abierto	sin asignar
3.5	Final Fase 3	● Abierto	sin asignar
4	Final TFG	● Abierto	sin asignar

Capítulo 3

Análisis

3.1 Especificación de requisitos

Para el desarrollo del compilador nos encontramos los siguientes tipos de requisitos:

- **Las funcionalidades del compilador serán:**

RF01 El sistema debe tener una tabla de símbolos que almacena la información sobre identificadores utilizados en el programa, como variables y funciones.

RF02 El sistema debe tener un analizador morfológico, que examinará la estructura de las palabras del código fuente para identificar sus partes y clasificarlas gramaticalmente.

RF03 El sistema debe ser capaz de realizar un análisis léxico del código fuente para identificar y clasificar los tokens, como identificadores, palabras clave, literales, etc.

RF04 Se requiere un análisis sintáctico para verificar la estructura sintáctica del código conforme a la gramática del lenguaje de programación objetivo.

RF05 El compilador debe realizar un análisis semántico para verificar la coherencia del código en términos de significado y contexto.

RF06 El compilador debe generar código intermedio que represente el programa de manera abstracta y optimizarlo para mejorar su eficiencia.

RF07 El sistema debe ser capaz de generar código objeto específico para la plataforma de destino a partir del código intermedio optimizado.

RF08 **Interfaz de usuario:** Se requiere una interfaz de usuario para facilitar la interacción con el compilador, permitiendo cargar archivos de código fuente, visualizar errores y resultados de compilación. Esta interfaz es

simple basándose en reducir la dificultad que puede llegar a tener para ciertos usuarios el compilar mediante comandos, de tal forma que con un simple botón de la interfaz se pueda compilar. De esta forma se consigue que sea accesible a gente experta y no experta en la informática.

- **Rendimiento:** El compilador debe tener un rendimiento adecuado, con tiempos de respuesta razonables para la compilación de programas de tamaño medio. No se contempla una mejora de rendimiento para programas de gran tamaño. Para la generación de la documentación docs la configuración está hecha para que tenga un tiempo de generación medio.
- **Capacidad:** Se establecerá la capacidad del compilador para manejar un número específico de líneas de código por segundo, garantizando que pueda procesar programas de tamaño considerable en un tiempo razonable.
- **Requisitos sobre entorno tecnológico y de comunicaciones:**
 1. **Requisitos de hardware:**
 - 4096MB de RAM
 - Almacenamiento: 10 GB (paquete MSYS2)
 - Acceso a Internet para descargar desde MSYS2 las herramientas a usar
 2. **Requisitos de software:**
 - Windows 10
 - En Windows, descargar MSYS2 (o en máquina virtual Ubuntu mediante Oracle VM VirtualBox)
 - Herramienta Flex
 - Herramienta Bison
 - Interfaz mediante herramienta Gtk
 - Para la documentación, herramientas: Doxygen y Graphviz
 3. **Compatibilidad con sistemas operativos:**
 - Linux
 - Windows (mediante MSYS2 o máquina virtual)
 4. **Herramientas de construcción y gestión de proyectos:**

- Makefile

Tabla 2. Requisitos Funcionales

Ref.	Descripción
RF01	La tabla de símbolos guarda información relevante sobre cada identificador encontrado en el programa fuente. Esto incluye su nombre, tipo, alcance (scope), valor (en el caso de constantes), dirección de memoria (en el caso de variables) y otros atributos necesarios para su correcto uso en el programa.
RF02	Examina la estructura de las palabras del código fuente para identificar sus partes y clasificarlas gramaticalmente. Paso inicial que se completa con el análisis léxico (RF03).
RF03	Analiza el flujo de caracteres del código fuente y dividirlo en unidades léxicas significativas, llamadas tokens. Estos tokens representan los elementos básicos del lenguaje de programación, como identificadores, palabras clave, literales, operadores, delimitadores, entre otros.
RF04	Verifica que el código fuente utilice la gramática del lenguaje correcta. Detecta y reporta errores sintácticos de manera precisa, indicando la ubicación y naturaleza del error.
RF05	Verifica la coherencia del código en términos de significado y contexto. Detecta y reporta errores semánticos, como variables no declaradas, tipos de datos incompatibles, etc.
RF06	Representa de manera abstracta el programa fuente, manteniendo su semántica. Maneja estructuras de control, expresiones y funciones de manera adecuada.
RF07	Genera código objeto específico para la plataforma de destino a partir del código intermedio optimizado. Genera un código objeto válido y ejecutable en la plataforma objetivo.
RF08	Interfaz de usuario que permita fácilmente compilar un fichero, mediante un apartado para introducir el fichero y una ejecución automática.

3.2 Análisis de los Casos de Uso

La definición de casos de uso para el desarrollador y el cliente proporciona una estructura clara y detallada de las interacciones esperadas entre los usuarios y el

sistema de compilación. A través de estos casos de uso se identifican las diversas acciones que los usuarios pueden llevar a cabo, desde la gestión de la tabla de símbolos hasta la compilación del código fuente.

Para el desarrollador, los casos de uso delimitan las actividades involucradas en el desarrollo y el análisis del código fuente, así como en la utilización de las herramientas proporcionadas por el compilador, como los analizadores léxico, sintáctico y semántico. Además, se destaca la importancia de la interfaz de usuario como medio de interacción con el sistema de compilación durante todo el proceso de desarrollo.

Por otro lado, para el cliente, los casos de uso se centran en la utilización de la interfaz de usuario para cargar archivos de código fuente, configurar opciones de compilación y realizar el proceso de compilación en sí mismo. Estos casos de uso reflejan la experiencia del usuario final al utilizar el compilador para generar código objeto a partir del código fuente proporcionado.

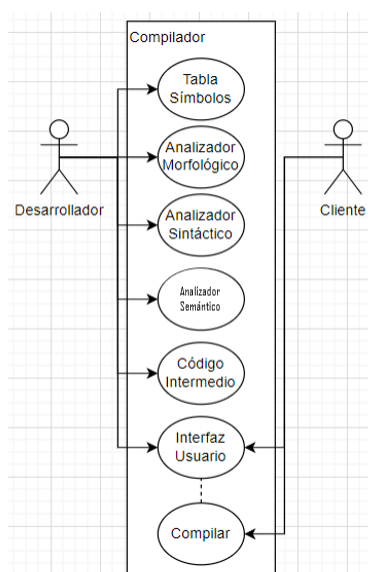


Figura 6. Casos de uso Desarrollador - Cliente (Diagrams.net)

En conclusión, los casos de uso proporcionan una visión holística de las funcionalidades y las interacciones del sistema de compilación, desde la perspectiva tanto del desarrollador como del cliente. Estos casos de uso sirven como una guía útil para el diseño, desarrollo y prueba del sistema, asegurando que cumpla con los requisitos y las expectativas de los usuarios finales.

Capítulo 4

Diseño e implementación

4.1 Arquitectura del sistema

En cuanto a la arquitectura del sistema, el cliente utiliza el compilador mediante una interfaz de usuario que le permite compilar un fichero.

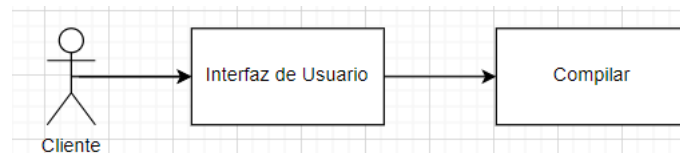


Figura 7. Arquitectura Sistema (Diagrams.net)

En el Frontend se encuentra la construcción y el diseño de la interfaz de usuario de la aplicación. Por otro lado, en el Backend se desarrolla la arquitectura interna de la aplicación, es decir, desde la tabla de símbolos hasta la generación de código.

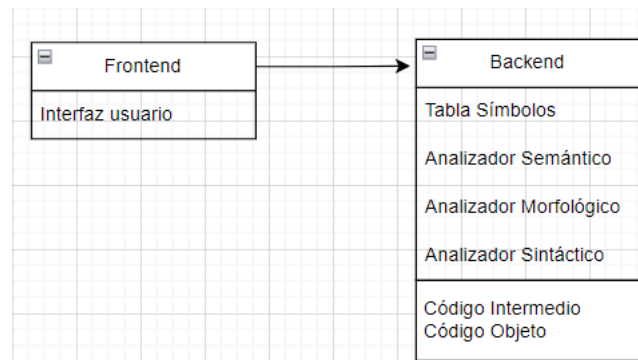


Figura 8. Frontend y Backend (Diagrams.net)

También, podemos ver la arquitectura del sistema en capas. Se distinguen 3 capas, la de Análisis donde se encuentran los Analizadores (morfológico, sintáctico, semántico), la de Generación de Código con el código objeto y código intermedio, y la de Usuario con la interfaz de usuario.

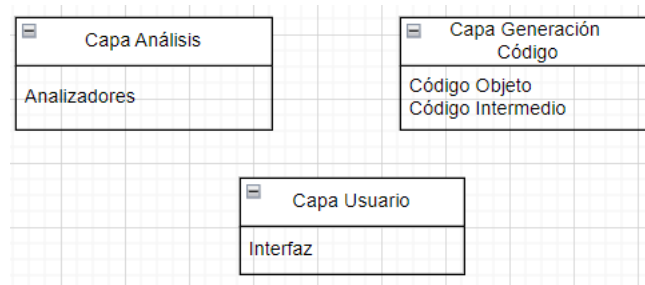


Figura 9. Capas del sistema (Diagrams.net)

4.1.1 Arquitectura física

En cuanto a la arquitectura física, el Cliente puede utilizar la aplicación de forma local, es decir, el compilador se utiliza mediante una interfaz de usuario que lo hace más accesible.



Figura 10. Arquitectura física (Diagrams.net)

4.1.2 Arquitectura lógica

En cuanto a la arquitectura lógica, la estructura del compilador se basa en 3 capas:

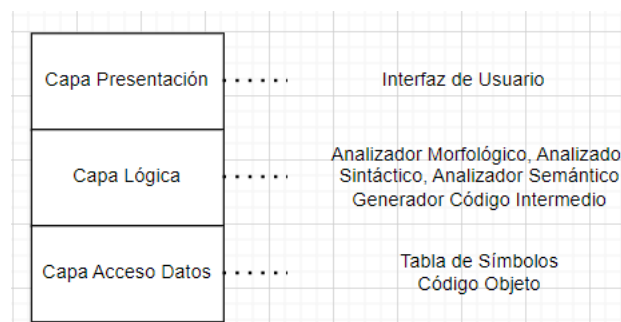


Figura 11. Arquitectura Lógica (Diagrams.net)

4.2 Diseño de la interfaz de usuario

Se desarrolla una interfaz de usuario básica donde se puede introducir un fichero que es el que queremos compilar. Una vez introducido se compila de forma automática ejecutándose la compilación, de tal forma que el usuario no tenga que escribir el

comando en la consola para compilar el fichero, y que sea accesible a los usuarios que no sean expertos en esta área.

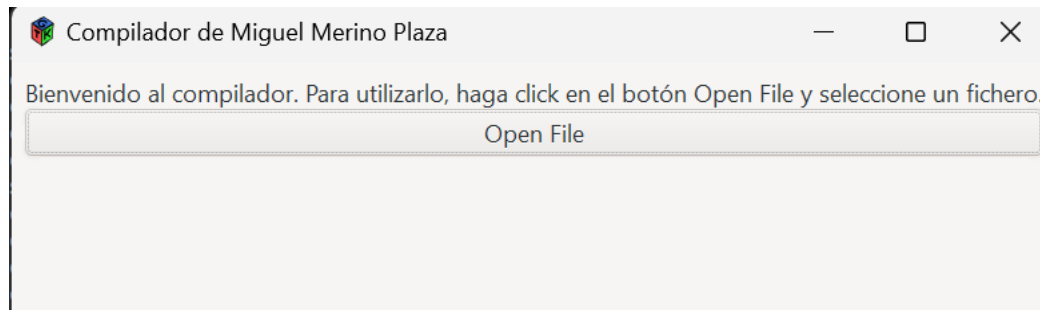


Figura 12. Interfaz de usuario (Interfaz programa)

Una vez se compila, sale un mensaje de error o de éxito en función de la compilación del fichero. En caso de que se produzca un error:

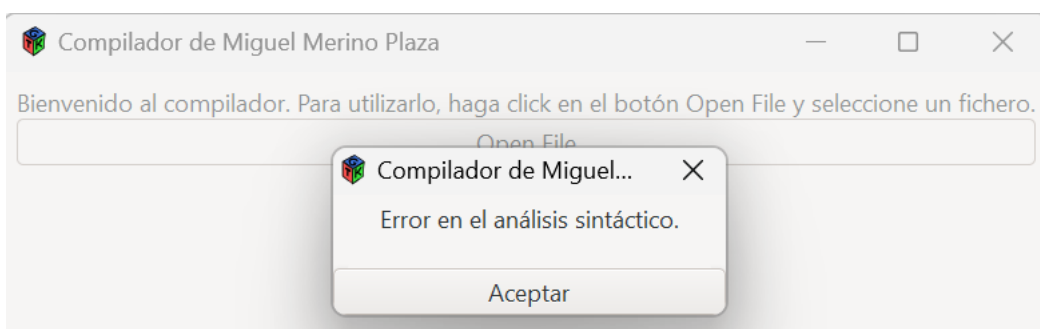


Figura 13. Respuesta de error compilador (Interfaz programa)

En caso de que el código compile de forma correcta y sin errores:

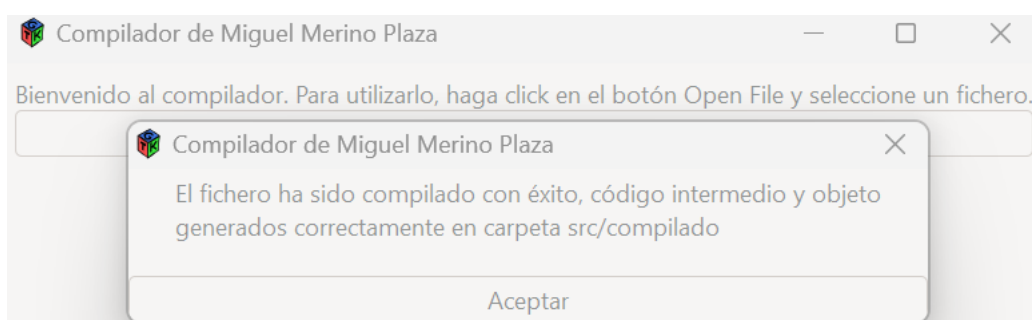


Figura 14. Respuesta de éxito compilador (Interfaz programa)

4.3 Esquema general

En la carpeta src la estructura es la siguiente:

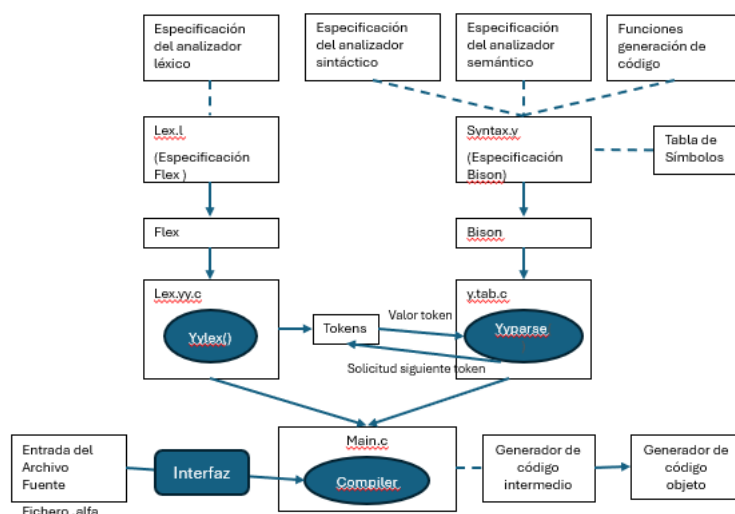


Figura 15. Esquema General Compilador

El proceso comienza con la entrada del archivo fuente. Este archivo, que contiene el código en un lenguaje de programación específico, es proporcionado por el usuario. El archivo puede ser seleccionado a través de una interfaz gráfica (GTK+) o proporcionado directamente como argumento en la línea de comandos (./compiler archivo.c). En ambos casos, el archivo fuente es enviado a la siguiente etapa del proceso de compilación.

El primer paso en el proceso de compilación es el análisis léxico. Este análisis se encarga de convertir el código fuente en una secuencia de tokens, que son las unidades básicas del lenguaje (como palabras clave, identificadores, operadores, etc.). Para realizar esta tarea, se utiliza una especificación del analizador léxico que está definida en el archivo Lex.l. Flex es la herramienta que lee este archivo y genera el código necesario en Lex.yy.c, que contiene la implementación de la función yylex(). Esta función es responsable de leer el código fuente y generar los tokens. Si se detecta un error léxico, como tokens desconocidos o malformados, se informa al usuario a través de la interfaz gráfica.

Después del análisis léxico, los tokens generados son enviados al análisis sintáctico. Esta etapa se encarga de construir un Árbol de Sintaxis Abstracta (AST) a partir de los

tokens. Las reglas gramaticales del lenguaje se especifican en el archivo `Syntax.y`, y Bison se encarga de leer este archivo para generar el código necesario en `y.tab.c`, que contiene la implementación de la función `yyparse()`. `yyparse()` solicita tokens a `yylex()` y verifica que las estructuras gramaticales del código fuente sean válidas según las reglas definidas. Si se detecta un error sintáctico, como una estructura gramatical incorrecta, se informa al usuario a través de la interfaz gráfica.

El siguiente paso es el análisis semántico, que realiza comprobaciones más profundas en el Árbol de Sintaxis Abstracta (AST). Durante este análisis, se valida que los tipos de datos sean compatibles, que las variables estén declaradas antes de ser utilizadas, que no haya identificadores duplicados, entre otras reglas semánticas. La Tabla de Símbolos juega un papel crucial en esta etapa, ya que mantiene un registro de las declaraciones de variables, tipos de datos y funciones. Si se detecta un error semántico, como una incompatibilidad de tipos o una variable no declarada, se muestra un mensaje de error en la interfaz gráfica.

Si el código fuente pasa con éxito las etapas de análisis léxico, sintáctico y semántico, se procede a la generación de código intermedio. Este código intermedio es una representación más abstracta y simplificada del código fuente, que es independiente del hardware. El código intermedio se guarda en un archivo llamado `codigo_intermedio.txt`, y sirve como un puente entre el código fuente y el código objeto. Finalmente, se realiza la generación de código objeto, que convierte el código intermedio en código objeto específico para la máquina objetivo. Este código objeto es la salida final del proceso de compilación y se guarda en un archivo llamado `codigo_objeto.txt`. La interfaz gráfica notifica al usuario si el proceso de compilación ha sido exitoso, mostrando un mensaje de éxito si se han generado correctamente tanto el código intermedio como el código objeto, o mostrando mensajes de error si se detectaron problemas durante cualquier etapa del proceso.

En cuanto a la estructura:

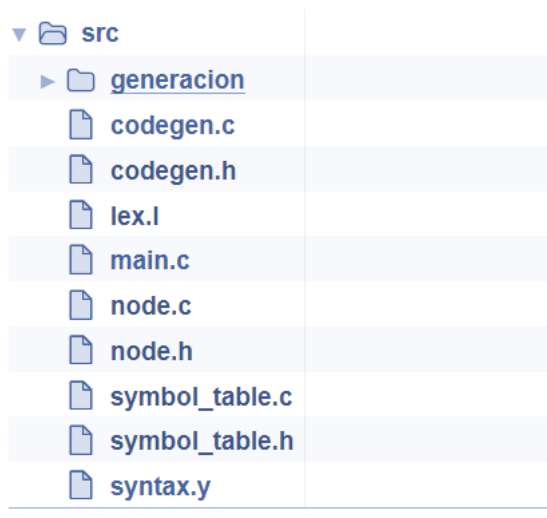


Figura 16. Estructura src (index.html)

Para ver más detalles de cada clase y diagramas acceder a index.html: make docs

TFG - DESARROLLO DE UN COMPILADOR

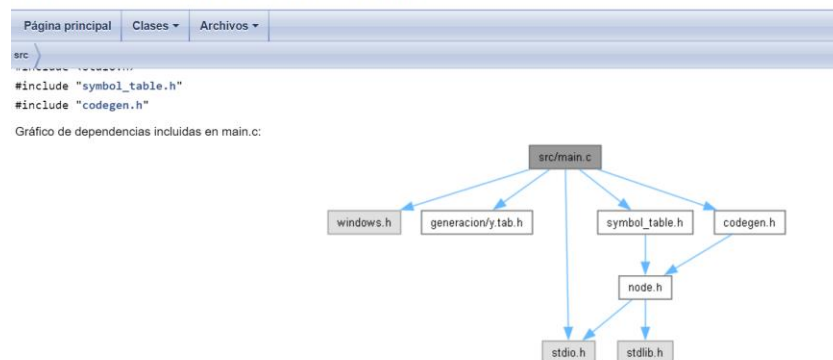


Figura 17. Diagrama de ejemplo clase main (index.html)

4.4 Construcción del entorno

Para la construcción del entorno se puede elegir entre usar msys2 o utilizar una máquina virtual como Ubuntu.

En este caso, se elige la construcción del entorno mediante msys2. Para ello, se accede a la web de msys2 para comenzar la instalación: <https://www.msys2.org/>

Una vez accedemos se siguen los pasos de instalación que se describen en la web:

- 1) Descarga el instalador

- 2) Una vez instalado, ejecutarlo (se requiere Windows 10)
- 3) Introduce una carpeta de instalación
- 4) Haz clic en finish
- 5) Una vez terminada la instalación, ya se puede utilizar la terminal:

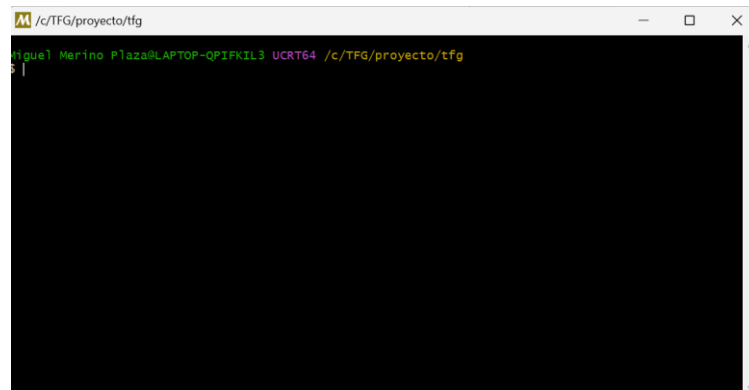


Figura 18. Terminal msys2 UCRT64

- 6) Ahora se instalan todas las herramientas necesarias para el desarrollo del compilador:

```
pacman -S mingw-w64-ucrt-x86_64-gcc
pacman -S flex
pacman -S bison
```

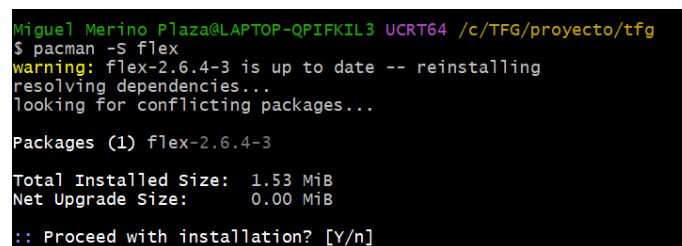


Figura 19. Terminal UCRT64: Ejemplo instalación herramienta Flex

- 7) Para comprobar que se tiene la herramienta:

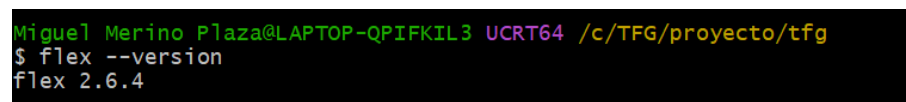


Figura 20. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta Flex

- 8) (Opcional) Para generar la documentación del proyecto se instalan 2 herramientas:

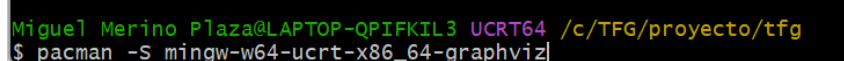


Figura 21. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta opcional graphviz

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg  
$ pacman -S doxygen
```

Figura 22. Terminal UCRT64: Ejemplo verificación de instalación correcta de la herramienta opcional doxygen

- 9) Con estas herramientas instaladas ya se tiene todo lo necesario para el desarrollo del compilador y se completa la construcción del entorno de desarrollo.
- 10) Para comenzar con el desarrollo del compilador:
 - a. Añadir un README.md donde se tiene que explicar el compilador y su funcionamiento...,
 - b. un .gitignore para que no se suban los ficheros que se generan (src/generación) con Flex, Bison y los ficheros .o para el desarrollo del compiler.exe,
 - c. establecer una estructura de directorios: carpeta src que contiene carpeta compilado donde se guarda el código que se genere (intermedio y objeto), carpeta ficheros donde están los ficheros a subir para ser compilados (inputs), carpeta generación donde se guarda todo lo que se genere para el compiler.exe (Flex, Bison, ficheros .o ...) y los distintos ficheros del compilador con sus correspondientes cabeceras para una mejor estructuración y uso (codegen, lex.l, main, node, semantic, symbol_table, syntax.y),
 - d. y desarrollar un Makefile para automatizar la ejecución, limpieza y compilación del compilador (*ver Anexo I.I*).

4.5 Referencia al repositorio de software

<https://github.com/miguelmp02/tfg>

Capítulo 5

Desarrollo del Sistema

5.1 Tabla de Símbolos

La tabla de símbolos está diseñada para mantener un registro de los identificadores (como nombres de variables y funciones) utilizados en un programa y proporcionar información relevante sobre éstos, como su tipo y su ámbito de visibilidad. En una primera versión, se pueden introducir los tokens del léxico en la tabla de símbolos, pero para un correcto uso, posteriormente se desarrolla con los tokens que se va encontrando el analizador sintáctico cuando realiza un recorrido por el fichero en forma de árbol.

Esta es la primera fase de creación del compilador, para ello se crea la versión final (integrada con el syntax.y con Bison) en un fichero `symbol_table.c` (*Ver Anexo I.II*) el cual tiene una cabecera `symbol_table.h` para crear una estructura más robusta.

Se añade a cada token el `printf` para comprobar su funcionamiento. Por ejemplo introducimos un código simple en un `input.alfa` que pasaremos al compilador:

```
x = 3  
y - 5 = x
```

Para saber si funciona de forma correcta, nos debe imprimir por la terminal lo siguiente:

```
Miguel Merino Plaza@LAPTOP-QP1FKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Insertado: x
IDENTIFIER(x) Token: 259 (x)
EQUAL Insertado: =
Token: 260 (=)
NUMBER(3) Insertado: 3
Token: 258 (3)
NEWLINE
Token: 266 (
)
Insertado: y
IDENTIFIER(y) Token: 259 (y)
MINUS Insertado: -
Token: 263 (-)
NUMBER(5) Insertado: 5
Token: 258 (5)
EQUAL Insertado: =
Token: 260 (=)
Insertado: x
IDENTIFIER(x) Token: 259 (x)
Tabla de símbolos
Nombre: x, Tipo: 259
Tabla de símbolos
Nombre: =, Tipo: 260
Tabla de símbolos
Nombre: 5, Tipo: 258
Tabla de símbolos
Nombre: -, Tipo: 263
Tabla de símbolos
Nombre: y, Tipo: 259
Tabla de símbolos
Nombre: 3, Tipo: 258
Tabla de símbolos
Nombre: =, Tipo: 260
Tabla de símbolos
Nombre: x, Tipo: 259
```

Figura 23. Terminal UCRT64: Ejemplo verificación de funcionamiento correcto de la tabla de símbolos

5.2 Analizador Léxico/Morfológico

Para el analizador léxico se usa la herramienta Flex. Flex es una herramienta que permite generar analizadores léxicos funcionales. En base a un conjunto de expresiones regulares, Flex busca similitudes en un fichero de input y ejecuta acciones relacionadas a estas expresiones. Es compatible casi al 100% con Lex, una herramienta clásica de Unix para la generación de analizadores léxicos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL. La salida de Flex va a ser el fichero 'lex.yy.c' que contiene la función de análisis 'yylex()', varias tablas usadas por esta para emparejar tokens, y unas cuantas rutinas auxiliares y macros.

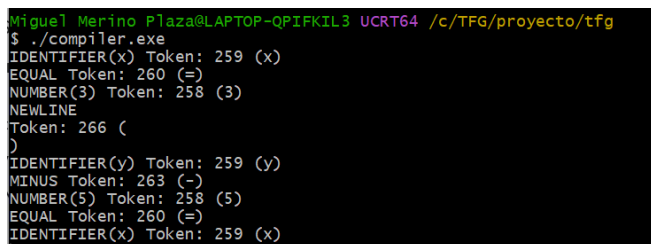
Se utiliza Flex como acompañante del analizador de gramáticas Bison. Los analizadores generados mediante Bison necesitan una función llamada 'yylex()' para devolverles el siguiente token del input. Esa función devuelve el tipo del próximo token y además puede poner cualquier valor asociado en la variable global yylval. Para usar y relacionar Flex con Bison, generalmente se especifica la opción -d de Bison para que genere el fichero 'y.tab.h' que contiene las definiciones de todos los '%tokens' que aparecen en Bison.

Para crear el analizador léxico se crea un fichero lex.l que contiene la definición del analizador léxico, que será el responsable de la segunda fase del proceso de compilación. En lex.l (*Ver Anexo I.III*) se definen reglas que corresponden a expresiones regulares para identificar cada tipo de token y acciones que se ejecutan cuando se reconoce un token.

Se añade a cada token el printf para comprobar su funcionamiento. Por ejemplo introducimos un código simple en un input.alfa que pasaremos al compilador:

```
x = 3
y - 5 = x
```

Para saber si funciona de forma correcta, nos debe imprimir por la terminal lo siguiente:



```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
IDENTIFIER(x) Token: 259 (x)
EQUAL Token: 260 (=)
NUMBER(3) Token: 258 (3)
NEWLINE
Token: 266 (
)
IDENTIFIER(y) Token: 259 (y)
MINUS Token: 263 (-)
NUMBER(5) Token: 258 (5)
EQUAL Token: 260 (=)
IDENTIFIER(x) Token: 259 (x)
```

Figura 24. Terminal UCRT64: Ejemplo verificación de funcionamiento correcto del analizador léxico

En cuanto a la gestión de los errores, para ver si el análisis léxico no funciona de forma correcta, implementamos:

```
%option yylineno
```

```
.\n { printf("Error token: '%s' no reconocido en la línea %d\n", yytext, yylineno); return
ERROR_TOKEN; }
```

Esta línea imprime todos los tokens que no estén reconocidos y la línea donde se encuentran (para ello es necesario implementar la option yylineno) del fichero en lex.l.

Por ejemplo, si quitamos del código lex.l:

```
"," { insert_symbol(yytext, SEMICOLON); /*printf("SEMICOLON ");*/ return SEMICOLON; }
"&" { insert_symbol(yytext, AMPERSAND); /*printf("AMPERSAND ");*/ return AMPERSAND; }
```

Y, además, para que nos compile el código, quitamos estos tokens de syntax.y, y si introducimos un fichero .alfa con un código que contenga esos tokens, deberemos recibir lo siguiente:

```
Miguel Merino Plaza@LAPTOP-QP1FKIL3 UCRT64 /c/TF6/proyecto/tfg
$ ./compiler.exe
Error token: 'X' no reconocido en la línea 4
Error token: 'X' no reconocido en la línea 4
Error token: 'X' no reconocido en la línea 6
Error token: 'X' no reconocido en la línea 6
Error token: 'X' no reconocido en la línea 7
Error token: 'X' no reconocido en la línea 8
Error token: 'X' no reconocido en la línea 8
Error token: 'X' no reconocido en la línea 8
Error token: 'X' no reconocido en la línea 8
Error token: 'X' no reconocido en la línea 8
Error token: 'X' no reconocido en la línea 29
Error token: 'X' no reconocido en la línea 35
Error token: 'X' no reconocido en la línea 39
Error token: 'X' no reconocido en la línea 55
Error token: 'X' no reconocido en la línea 61
Error token: 'X' no reconocido en la línea 65
Error en el análisis léxico.
```

Figura 25. Terminal UCRT64: Ejemplo verificación de analizador léxico para Error – token no X reconocido en línea Y

Con esto concluimos el desarrollo del analizador léxico, el cual puede expandirse en un futuro añadiendo todo tipo de tokens que se incorporen a la gramática alfa para tener un análisis léxico de cualquier fichero .alfa de forma completa y exacta.

5.3 Analizador Sintáctico

Un analizador sintáctico, también conocido como parser, es una componente crítica de un compilador o intérprete, cuya función principal es analizar la secuencia de tokens generada por el analizador léxico para construir una estructura de datos (generalmente un árbol de análisis gramatical) que representa la estructura gramatical del programa fuente. Este proceso implica comprobar y validar la sintaxis del código fuente contra las reglas de una gramática formal definida para el lenguaje de programación, asegurando que el código esté estructurado correctamente.

El analizador sintáctico utiliza estas reglas para reconocer las construcciones del lenguaje, como expresiones, declaraciones y bloques de código, y organizar estos elementos en una forma que refleje su jerarquía y relación en el código fuente. El resultado de este análisis es esencial para las etapas posteriores de la compilación, como el análisis semántico y la generación de código.

Para la tercera fase de desarrollo del compilador se usa Bison que es un generador de analizadores que convierte una descripción gramatical independiente del contexto en

un programa en C que analiza esa gramática. Bison toma una especificación de gramática en forma de reglas gramaticales o producciones y genera código en C que parsea texto de acuerdo con esas reglas. Las reglas definen cómo se deben formar las sentencias del lenguaje desde los componentes más básicos (tokens), que son identificados por un analizador léxico como Flex. Bison está disponible bajo la licencia GPL, permitiendo su uso y modificación de manera libre en proyectos que respeten esta licencia, lo que es atractivo para proyectos de software libre y de código abierto.

Para crear el analizador sintáctico se crea un fichero `syntax.y` que contiene la definición del analizador sintáctico, que es el responsable de la tercera etapa del proceso de compilación. En `syntax.y` se definen reglas que corresponden a expresiones regulares para identificar cada tipo de token y acciones que se ejecutan cuando se reconoce un token.

Cabe destacar que para cada regla se guardan los valores en un árbol de tal forma que cada regla llama a una función específica que crea unos nodos que formarán el árbol de análisis. El analizador sintáctico recorrerá el código en forma de árbol y esto se define en una clase `node.c`.

La versión final de `syntax.y` contiene las reglas sintácticas y, además, incorpora para una mejor gestión con Bison las reglas semánticas oportunas. Además, se añade en aquellas operaciones de asignación, lógica y aritméticas, llamadas a una función que posteriormente, sirve para llevar a cabo la generación del código intermedio.

A continuación se observa el código del `syntax.y` (*Ver Anexo I.IV*)

Se pone en el código un `printf` para poder observar que funciona de forma correcta y se procede a ejecutar el compilador introduciendo el fichero siguiente:

```
main
{
int z;
int x;
int y;
x=2+y;
printf y;
}
```

Con este código se observa:

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 4.
Declarando variable 'z' de tipo 'int' en la línea 4.
Tipo detectado: int en la línea 5.
Declarando variable 'x' de tipo 'int' en la línea 5.
Numero detectado: 0 en la línea 7.
Asignando a 'z' el resultado de la expresion en la línea 7.
Numero detectado: 2 en la línea 8.
Identificador detectado: z en la línea 8.
Operacion PLUS en línea 8.
Asignando a 'x' el resultado de la expresion en la línea 8.
Llamando a 'printf' con 'z' en la línea 10.
Programa main procesado correctamente en la línea 12.
Análisis sintactico completado correctamente.
Análisis lexico completado de forma correcta.
Tabla de simbolos completada de forma correcta.
```

Figura 26. Terminal UCRT64: Ejemplo verificación de funcionamiento del analizador sintáctico

En cuanto a la gestión de los errores, para ver si el análisis sintáctico no funciona de forma correcta, se implementa el mismo código de antes, pero con un cambio para que nos de un error de sintaxis, como puede ser quitar el “;” a “int z”:

```
main
{
int z
int x;
int y;
x=2+y;
printf y;
}
```

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Error: falta ';' despues de la declaracion de 'z' en la línea 4.
Error de analisis en la línea 4: Falta ';' despues de la declaracion
Tipo detectado: int en la línea 4.
Declarando variable 'x' de tipo 'int' en la línea 4.
Tipo detectado: int en la línea 5.
Declarando variable 'y' de tipo 'int' en la línea 5.
Numero detectado: 2 en la línea 6.
Identificador detectado: y en la línea 6. Tipo: 0
Operacion PLUS en línea 6.
Asignando a 'x' el resultado de la expresion en la línea 6.
Llamando a 'printf' con 'y' en la línea 7.
Programa main procesado correctamente en la línea 10.
Error en el analisis sintactico.
```

Figura 27. Terminal UCRT64: Ejemplo verificación de analizador sintáctico para Error de sintaxis – Error de sintaxis en línea X

5.4 Analizador Semántico

Un analizador semántico es una parte esencial de un compilador que se encarga de verificar la corrección semántica del programa, asegurándose de que cumple con las reglas del lenguaje de programación. Este proceso implica la comprobación de tipos, la validez de las operaciones, la correcta declaración y uso de variables, entre otros aspectos. El analizador semántico sirve para garantizar que un programa no solo esté

bien escrito desde el punto de vista de la sintaxis, sino también que tenga sentido lógico y funcional.

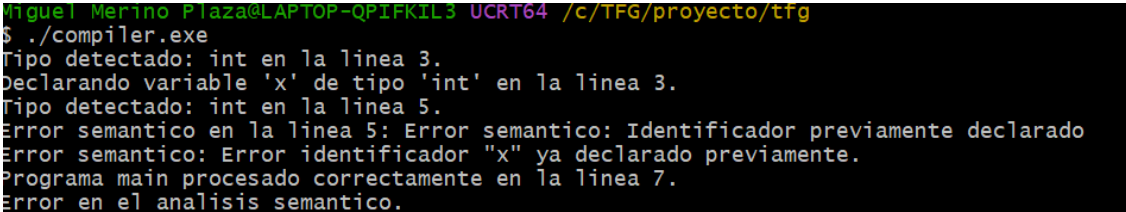
La integración del analizador semántico con el sintáctico en Bison se realiza insertando acciones semánticas dentro de las reglas gramaticales definidas en el archivo de especificación de Bison. Estas acciones son fragmentos de código que se ejecutan cuando se reconoce una regla gramatical específica.

Para desarrollar el analizador semántico se añaden reglas en el `syntax.y` ya definido anteriormente y donde se encuentra el analizador sintáctico desarrollado en el apartado anterior.

A continuación, se detalla lo que es la cuarta fase de desarrollo del compilador con las distintas restricciones semánticas que se han implementado mediante reglas en el `syntax.y` (este código se encuentra desarrollado de forma completa en el analizador sintáctico anteriormente mencionado) :

1. No se permite la declaración de un identificador ya declarado previamente. Por ejemplo, si se introduce un código simple donde se declara dos veces el mismo identificador, se puede observar que da un error semántico:

```
main
{
    int x;
    int x;
}
```



```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'x' de tipo 'int' en la línea 3.
Tipo detectado: int en la línea 5.
Error semántico en la línea 5: Error semántico: Identificador previamente declarado
Error semántico: Error identificador "x" ya declarado previamente.
Programa main procesado correctamente en la línea 7.
Error en el análisis semántico.
```

Figura 28. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X previamente declarado

2. No se permite la declaración de un identificador de clase ya declarado previamente. Por ejemplo, si se introduce un código simple donde se declara dos veces una clase con el mismo identificador, se puede observar que dará un error semántico:

```
main
{
}
cerrar{
```

```
    return y;
}
cerrar{
```

```
    return z;
}
```

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'x' de tipo 'int' en la línea 3.
Tipo detectado: int en la línea 5.
Declarando variable 'y' de tipo 'int' en la línea 5.
Identificador detectado: y en la línea 10.
Declaracion de retorno en la línea 10.
Definiendo clase 'cerrar' en la línea 12.
Identificador detectado: z en la línea 14.
Declaracion de retorno en la línea 14.
Error semantico en la línea 15: Error semantico: Identificador de clase previamente declarado
Programa main procesado correctamente en la línea 15.
Error en el analisis semantico.
```

Figura 29. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X de clase previamente declarado

3. No se permite la declaración de un identificador de función ya declarado previamente. Por ejemplo, si se introduce un código simple donde se declara dos veces una misma función con el mismo identificador, se puede observar que da un error semántico:

```
main
```

```
{
    function int suma (int a;int b)
    {
        int s;
    }
    function int suma (int a;int b)
    {
        int s;
    }
}
```

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Tipo detectado: int en la línea 3.
Tipo detectado: int en la línea 3.
Tipo detectado: int en la línea 5.
Declarando variable 's' de tipo 'int' en la línea 5.
Definicion de funcion 'suma' con tipo 'int' y retorno asegurado en la línea 6.
Tipo detectado: int en la línea 7.
Tipo detectado: int en la línea 7.
Tipo detectado: int en la línea 7.
Tipo detectado: int en la línea 9.
Error semantico en la línea 9: Error semantico: Identificador previamente declarado
Error semantico: Error identificador "s" ya declarado previamente.
Error semantico en la línea 10: Error semantico: Identificador de funcion previamente declarado
Programa main procesado correctamente en la línea 11.
Error en el analisis semantico.
```

Figura 30. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error Identificador X de función previamente declarado

4. No se permite el uso de un identificador que no ha sido declarado previamente. Por ejemplo, si se introduce un código simple donde se utiliza una variable que no ha sido declarada previamente, se puede observar que da un error semántico:

```
main
```

```
{
```



```

int y;
z = 0;
}
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'y' de tipo 'int' en la línea 3.
Numero detectado: 0 en la línea 5.
Error semantico en la línea 5: Error semantico: Identificador no declarado previamente
Error semantico: Error identificador "z" no declarado previamente.
Programa main procesado correctamente en la línea 6.
Error en el analisis semantico.

```

Figura 31. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador X no declarado previamente

5. No se permite el uso de un identificador de función que no ha sido declarado previamente. Por ejemplo, si se introduce un código simple donde se utiliza una función que no ha sido declarada previamente, se puede observar que da un error semántico:

```

main
{
    int c;
    function int resta (int a;int b)
    {
        int s;
    }
    c=suma(0,1);
}
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'c' de tipo 'int' en la línea 3.
Tipo detectado: int en la línea 4.
Tipo detectado: int en la línea 4.
Tipo detectado: int en la línea 4.
Tipo detectado: int en la línea 6.
Declarando variable 's' de tipo 'int' en la línea 6.
Definición de funcion 'resta' con tipo 'int' y retorno asegurado en la línea 7.
Numero detectado: 0 en la línea 8.
Numero detectado: 1 en la línea 8.
Error semantico en la línea 8: Error semantico: Identificador de funci|n no declarado previamente
Error semantico: Error identificador "suma" de funci|n no declarado previamente.
Asignando a 'c' el resultado de la llamada a funcion en la línea 8.
Programa main procesado correctamente en la línea 9.
Error en el analisis semantico.

```

Figura 32. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador de función X no declarado previamente

6. No se permite el uso de una llamada a función que no tiene el mismo número de argumentos que la función declarada previamente. Por ejemplo, si se introduce un código simple donde se llama a una función con unos argumentos distintos que la función declarada previamente, se puede observar que da un error semántico:

```

main
{
    function int suma (int a;int b)
    {
        return (suma(n-1));
    }
}

```

```

}
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Tipo detectado: int en la línea 3.
Tipo detectado: int en la línea 3.
Declaración de función 'suma' con tipo 'int' en la línea 4.
Identificador detectado: n en la línea 5.
Número detectado: 1 en la línea 5.
Operación MINUS en línea 5.
Error semántico en la línea 5: Error semántico: Número incorrecto de argumentos en la llamada a la función
Error semántico: Número incorrecto de argumentos en la llamada a la función 'suma'. Esperados 2, encontrados 1 en la línea 5.
Declaración de retorno en la línea 5.
Definición de función 'suma' con tipo 'int' y retorno asegurado en la línea 7.
Programa main procesado correctamente en la línea 8.
Error en el análisis semántico.

```

Figura 33. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error uso de identificador de función X no declarado previamente

7. No se permite dentro de una función de un tipo el retorno de una variable/identificador de otro tipo distinto a la función. Por ejemplo, si se introduce un código simple donde se tiene una función de un tipo que retorna un identificador de otro tipo, se puede observar que da un error semántico:

```

main
{
    boolean z;
    function int suma (int a)
    {
        return z;
    }
}
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: boolean en la línea 3.
Declarando variable 'z' de tipo 'boolean' en la línea 3.
Tipo detectado: int en la línea 4.
Tipo detectado: int en la línea 4.
Declaración de función 'suma' con tipo 'int' en la línea 5.
Error semántico en la línea 6: Error semántico: Tipo de retorno incompatible.
Error semántico: Tipo de retorno incompatible. Tipo de 'z': 3, Tipo esperado: 0
Definición de función 'suma' con tipo 'int' y retorno asegurado en la línea 8.
Programa main procesado correctamente en la línea 9.
Error en el análisis semántico.

```

Figura 34. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error tipo de retorno incompatible con el tipo de función.

No se permite que se realicen operaciones aritméticas, lógicas y de asignación entre tipos de datos que no son compatibles. Por ejemplo, si se declara una variable de un tipo y otra variable de otro tipo diferente, si se realiza la suma de ambas variables se puede observar que da un error semántico:

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'z' de tipo 'int' en la línea 3.
Tipo detectado: boolean en la línea 4.
Declarando variable 'x' de tipo 'boolean' en la línea 4.
Identificador detectado: x en la línea 5. Tipo: 4
Identificador detectado: z en la línea 5. Tipo: 0
Error semántico en la línea 5: Error semántico: Operación PLUS solo permitida entre enteros.
Error semántico: Operación PLUS solo permitida entre enteros. Tipo de operando 1: 4, Tipo de operando 2: 0
Asignando a 'x' el resultado de la expresión en la línea 5.
Programa main procesado correctamente en la línea 6.
Error en el análisis semántico.
```

Figura 35. Terminal UCRT64: Ejemplo verificación de analizador semántico para Error semántico – Error tipo de variable incompatible con otro tipo de otra variable.

Estas reglas semánticas están ampliadas a todos los distintos casos que se desarrollan en el analizador sintáctico, y sólo se muestran capturas de ejemplo de algunos errores. Una vez solucionados todos los errores semánticos, al compilar el fichero se debe observar que todo funciona de forma correcta:

```
main
{
    int z;
    int x;
    x=x+z;
}
```

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Tipo detectado: int en la línea 3.
Declarando variable 'z' de tipo 'int' en la línea 3.
Tipo detectado: int en la línea 4.
Declarando variable 'x' de tipo 'int' en la línea 4.
Identificador detectado: x en la línea 5. Tipo: 0
Identificador detectado: z en la línea 5. Tipo: 0
Operación PLUS en línea 5.
Asignando a 'x' el resultado de la expresión en la línea 5.
Programa main procesado correctamente en la línea 6.
Análisis léxico completado de forma correcta.
Tabla de símbolos completada de forma correcta.
Análisis sintáctico completado de forma correcta.
Análisis semántico completado de forma correcta.
```

Figura 36. Terminal UCRT64: Ejemplo verificación de analizador semántico funciona de forma correcta

5.5 Analizadores

Una vez desarrollados todos los analizadores que forman el compilador, se procede con la compilación de este. Una vez compila, se pasa a ejecutar el compilador introduciendo un fichero.

Si este fichero no tiene ningún error ni léxico, ni sintáctico, ni semántico, es decir, todos los analizadores han funcionado de forma correcta, se observa lo siguiente:

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Análisis léxico completado de forma correcta.
Análisis sintáctico completado de forma correcta.
Tabla de símbolos completada de forma correcta.
Análisis semántico completado de forma correcta.
<-----Análisis terminados de forma correcta----->
```

Figura 37. Terminal UCRT64: Ejemplo verificación analizadores funcionan de forma correcta.

5.6 Generación de código

Una vez terminada la parte de los analizadores, falta la última fase que es la de generación de código.

La generación del código intermedio es un paso crucial en el proceso de compilación que se lleva a cabo en compiladores de múltiples pasadas. Su objetivo es crear una representación intermedia del programa fuente que permita la optimización del código y la portabilidad a múltiples plataformas. Este proceso involucra la generación de tuplas (cuádruplas o ternas) que representan operaciones parciales y sus resultados intermedios.

El proceso de generación de cuádruplas utiliza una gramática de atributos, donde las acciones semánticas se definen para cada producción de la gramática.

Para la creación del código intermedio se crea una función “generate_quad” que esta en la clase codegen.c, y que genera las distintas cuádruplas.

```
void generate_quad(Operation op, char *arg1, char *arg2, char *result) {
    if (quad_index >= MAX_QUADS) {
        fprintf(stderr, "Error: Exceeded maximum number of quadruples\n");
        exit(1);
    }
    quads[quad_index].op = op;
    quads[quad_index].arg1 = strdup(arg1);
    quads[quad_index].arg2 = strdup(arg2);
    quads[quad_index].result = strdup(result);
    quad_index++;
}
```

Para continuar con el desarrollo, se realizan llamadas a esta función para crear las cuádruplas, desde el analizador sintáctico y semántico, es decir, utilizando Bison desde el syntax.y, desarrollado previamente (se puede observar en el código del analizador sintáctico, donde se incluyen estas llamadas en las operaciones lógicas, aritméticas y de asignación).

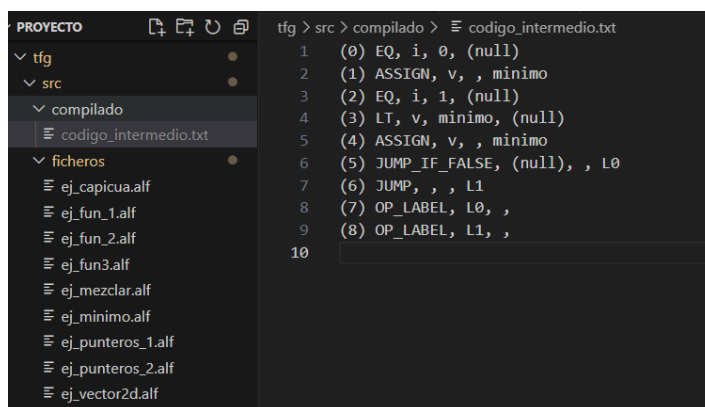
Como se puede observar en la imagen, al meterle un fichero se pueden observar las cuádruplas generadas por un código simple como este:

```
main
{
    int a;
    int b;
    int c;
    if (a < b) {
        c = 1;
    } else {
        c = 2;
    }
}
```

```
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Análisis léxico completado de forma correcta.
Análisis sintáctico completado de forma correcta.
Tabla de símbolos completada de forma correcta.
Análisis semántico completado de forma correcta.
<-----Análisis terminados de forma correcta----->
Codigo Intermedio:
(0) LT, a, b, (null)
(1) ASSIGN, 1, , c
(2) ASSIGN, 2, , c
(3) JUMP_IF_FALSE, (null), , L0
(4) JUMP, , , L1
(5) OP_LABEL, L0, ,
(6) OP_LABEL, L1, ,
```

Figura 38. Terminal UCRT64: Ejemplo verificación generación de código intermedio funciona de forma correcta

Una vez se genera el código intermedio, interesa guardar el resultado en un fichero que tendremos en la carpeta /src/compilado, el cual se crea de forma automática en caso de que no exista previamente ahí.



```
tfg > src > compilado > ≡ codigo_intermedio.txt
1 (0) EQ, i, 0, (null)
2 (1) ASSIGN, v, , minimo
3 (2) EQ, i, 1, (null)
4 (3) LT, v, minimo, (null)
5 (4) ASSIGN, v, , minimo
6 (5) JUMP_IF_FALSE, (null), , L0
7 (6) JUMP, , , L1
8 (7) OP_LABEL, L0, ,
9 (8) OP_LABEL, L1, ,
10
```

Figura 39. Terminal UCRT64: Ejemplo verificación generación de código intermedio funciona y se guarda en fichero de forma correcta

Una vez se ha generado el código intermedio, ahora solo queda generar el código objeto. El código objeto se basa en la compilación de un programa fuente, es una representación de bajo nivel del programa. Generalmente, el código objeto contiene instrucciones en lenguaje de máquina.

Para desarrollar esta última parte del compilador, se necesita desarrollar un traductor de código intermedio a código objeto que interprete las cuádruplas del código fuente y genere el código objeto. Para desarrollar el traductor a código objeto, se utiliza la clase `codegen.c` definido anteriormente para el código intermedio, y se añade una función `generate_object_code` que realiza esta traducción:

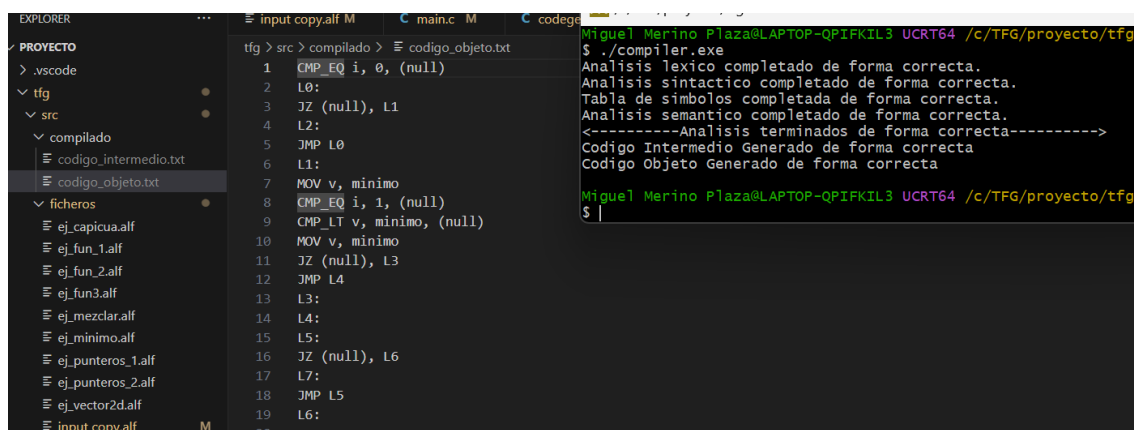
```
void generate_object_code(FILE *objfile) {
    for (int i = 0; i < quad_index; i++) {
        Quadruple q = quads[i];
        switch (q.op) {
            case OP_ADD:
                fprintf(objfile, "ADD %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_SUB:
                fprintf(objfile, "SUB %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_MUL:
                fprintf(objfile, "MUL %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_DIV:
                fprintf(objfile, "DIV %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_ASSIGN:
                fprintf(objfile, "MOV %s, %s\n", q.arg1, q.result);
                break;
            case OP_JUMP:
                fprintf(objfile, "JMP %s\n", q.result);
                break;
            case OP_JUMP_IF_FALSE:
                fprintf(objfile, "JZ %s, %s\n", q.arg1, q.result);
                break;
            case OP_LABEL:
                fprintf(objfile, "%s:\n", q.arg1);
                break;
            case OP_LT:
                fprintf(objfile, "CMP_LT %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_LE:
                fprintf(objfile, "CMP_LE %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_GT:
                fprintf(objfile, "CMP_GT %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_GE:
                fprintf(objfile, "CMP_GE %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_EQ:
                fprintf(objfile, "CMP_EQ %s, %s, %s\n", q.arg1, q.arg2, q.result);
                break;
            case OP_NE:
                fprintf(objfile, "CMP_NE %s, %s, %s\n", q.arg1, q.arg2, q.result);
```

```

        break;
    case OP_AND:
        fprintf(objfile, "AND %s, %s, %s\n", q.arg1, q.arg2, q.result);
        break;
    case OP_OR:
        fprintf(objfile, "OR %s, %s, %s\n", q.arg1, q.arg2, q.result);
        break;
    default:
        fprintf(objfile, "UNKNOWN %s, %s, %s, %s\n", operation_to_string(q.op), q.arg1, q.arg2,
q.result);
        break;
    }
}
}
}

```

Una vez se tiene esto, se realiza una llamada desde el main para que la salida se escriba y guarde en un fichero de la carpeta /src/compilado como se puede observar en la siguiente imagen:



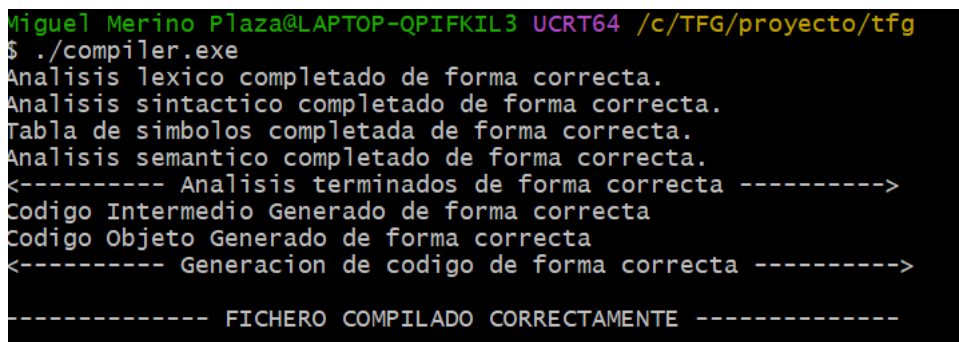
```

Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Analisis lexico completado de forma correcta.
Analisis sintactico completado de forma correcta.
Tabla de simbolos completada de forma correcta.
Analisis semantico completado de forma correcta.
<-----Analisis terminados de forma correcta----->
Codigo Intermedio Generado de forma correcta
Codigo Objeto Generado de forma correcta
Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$

```

Figura 40. Terminal UCRT64: Ejemplo verificación generación de código objeto funciona y se guarda en fichero de forma correcta

Una vez todo ha funcionado de forma correcta se puede observar lo siguiente:



```

Miguel Merino Plaza@LAPTOP-QPIFKIL3 UCRT64 /c/TFG/proyecto/tfg
$ ./compiler.exe
Analisis lexico completado de forma correcta.
Analisis sintactico completado de forma correcta.
Tabla de simbolos completada de forma correcta.
Analisis semantico completado de forma correcta.
<----- Analisis terminados de forma correcta ----->
Codigo Intermedio Generado de forma correcta
Codigo Objeto Generado de forma correcta
<----- Generacion de codigo de forma correcta ----->
----- FICHERO COMPILADO CORRECTAMENTE -----

```

Figura 41. Terminal UCRT64: Ejemplo verificación compilador funciona de forma correcta

5.7 Interfaz de usuario

Para hacer más accesible el uso del compilador se ha incorporado una interfaz de usuario simple de tal forma que al ejecutar el compilador se nos abra una ventana para seleccionar un fichero:

`./compiler.exe`

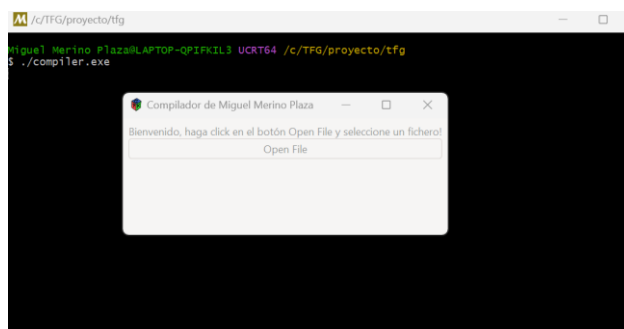


Figura 42. Terminal UCRT64: Ejemplo verificación de interfaz de usuario

Una vez se abre la ventana, se abre la carpeta `src/ficheros` del proyecto donde se puede seleccionar un fichero `.alfa`, de los que se hayan guardado en ese directorio u otro fichero, que se quiera compilar.

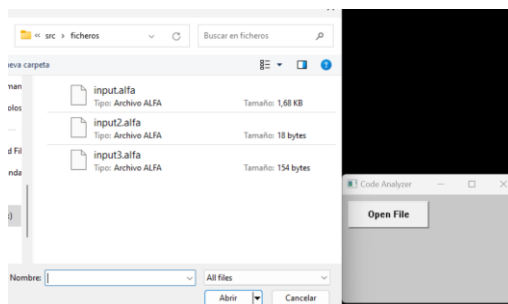


Figura 43. Terminal UCRT64: Ejemplo verificación de interfaz de usuario – Selección fichero `.alfa` a compilar

Una vez seleccionado se puede observar la salida del compilador por la terminal donde se puede ver si hay algún error (Ej: error “X” en línea “Y”), el tipo de error (Ej: error léxico..., error semántico... o error sintáctico...); o todo el código ha compilado de forma correcta.

En caso de que compile de forma correcta se genera un código intermedio y posteriormente un código objeto. Todo esto se ha podido ir viendo en las distintas capturas proporcionadas para explicar los distintos componentes del compilador.

Capítulo 6

Conclusiones y líneas futuras

6.1 Conclusiones

El desarrollo de un compilador es un proceso complejo que implica una serie de etapas fundamentales, desde la creación de una tabla de símbolos hasta la generación de código objeto. A lo largo de este trabajo, se ha explorado y desarrollado cada una de estas etapas, con el objetivo de comprender y aplicar los principios y técnicas involucrados en la construcción de un compilador.

Durante el proceso de desarrollo, se ha diseñado e implementado una arquitectura modular y escalable que permite realizar cada etapa del proceso de compilación de manera independiente y eficiente. Se han utilizado técnicas de análisis léxico, sintáctico y semántico para garantizar la corrección y coherencia del código fuente, se ha optimizado la generación del código intermedio generado para mejorar su eficiencia y rendimiento, y se ha realizado su transformación a código objeto.

A través de la realización de este trabajo, se ha adquirido un profundo conocimiento sobre los diferentes aspectos involucrados en el desarrollo de un compilador, así como una comprensión más amplia de los principios fundamentales de la ciencia de la computación y la ingeniería de software.

Por último, este conocimiento adquirido puede ser muy útil para los nuevos lenguajes y tecnologías que están surgiendo en la actualidad y que necesiten un compilador adecuado a su gramática, tokens...

6.2 Líneas futuras

A pesar de los avances realizados en este trabajo, aún existen diversas áreas que podrían explorarse en el futuro para mejorar y ampliar el compilador desarrollado:

Extensión del analizador léxico: nuevos tokens que puedan irse añadiendo a la gramática para incorporar nuevas reglas.

Extensión del analizador sintáctico: nuevas reglas gramáticas que hagan más eficaz el análisis sintáctico del fichero.

Extensión del analizador semántico: nuevas restricciones semánticas que hagan más eficaz y exacto el análisis semántico del fichero.

Optimización del código generado: Se podría investigar y aplicar técnicas más avanzadas de optimización de código para mejorar aún más el rendimiento del compilador y el código generado.

Soporte para características adicionales del lenguaje: Se podrían agregar nuevas características al compilador para admitir un conjunto más amplio de construcciones de lenguaje y mejorar su compatibilidad con estándares y especificaciones existentes.

Compatibilidad: Que el compilador pueda detectar distintos lenguajes de programación, y que funcione de la misma manera en las diferentes plataformas que existen (multiplataforma).

Mejoras en la interfaz de usuario y experiencia del desarrollador: Se podría trabajar en el diseño y la implementación de una interfaz de usuario más compleja que a la vez facilite el uso del compilador tanto por parte de los usuarios como de los desarrolladores.

Exploración de nuevas tecnologías y enfoques de desarrollo: Se podría investigar y evaluar el uso de nuevas tecnologías y metodologías de desarrollo que puedan mejorar la eficiencia y la calidad del compilador, como el uso de herramientas de compilación just-in-time (JIT) o el desarrollo basado en pruebas (TDD).

Bibliografía

- [1] Compiladores, principios técnicas y herramientas. Segunda edición. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Pearson - Addison Wesley, 2008
- [2] Modern Compiler Implementation in C/Java/ML - Andrew W. Appel
- [3] Engineering a Compiler - Keith D. Cooper, Linda Torczon
- [4] Flex & Bison - O'Reilly Media
- [5] Lex & Yacc - John R. Levine, Tony Mason, Doug Brown
- [6] OpenAI. (2024). ChatGPT (versión 3.5) [Modelo de lenguaje de gran tamaño]. <https://chat.openai.com/chat>
- [7] Qué es un compilador: para qué sirve y ejemplos. (2024, 23 abril). HubSpot. <https://blog.hubspot.es/website/que-es-compilador>
- [8] Calvo, J. (s. f.-b). ¿Que es un Compilador en programación? – Blog Europeanvalley. <https://www.europeanvalley.es/noticias/que-es-un-compilador-en-programacion/>
- [9] *Intro Flex y Bison*. (s. f.). <https://webdiis.unizar.es>. https://webdiis.unizar.es/asignaturas/LGA/material_2_004_2005/Intro_Flex_Bison.pdf
- [10] Compilador. (s. f.). *Repositorio Comillas*. <https://repositorio.comillas.edu/jspui/bitstream/11531/9589/1/PFC000282.pdf>
- [11] Desarrollo de un compilador para un lenguaje funcional con gestión explícita de la memoria. (s. f.). *Docta UCM*. <https://docta.ucm.es/rest/api/core/bitstreams/dab18a35-07c3-4f99-af5f-a4ab967980d6/content>
- [12] *Lexical Analysis With Flex, for Flex 2.6.2: Top*. (s. f.). <https://westes.github.io/flex/manual/>
- [13] *MSYS2*. (s. f.). *MSYS2*. <https://www.msys2.org/>
- [14] Colaboradores de Wikipedia. (2024, 25 abril). *GNU Bison*. Wikipedia, la Enciclopedia Libre. https://es.wikipedia.org/wiki/GNU_Bison
- [15] Compiladores. (s. f.). *Docencia*. http://arantxa.ii.uam.es/~mdlacruz/docencia/compiladores/2009_2010/gramatica%20de%20alfa.pdf
- [16] *Stack Overflow - where developers learn, share, & build careers*. (s. f.). Stack Overflow. <https://stackoverflow.com/>

Anexo I – Código

Anexo I.I – Código Makefile

```
CC = gcc
CFLAGS = -I. -g -Isrc/
FLEX = flex
BISON = bison
MKDIR_P = mkdir -p
OUT_DIR = src/generacion
EXEC = compiler

OBJS = $(OUT_DIR)/lex.yy.o $(OUT_DIR)/y.tab.o $(OUT_DIR)/symbol_table.o
$(OUT_DIR)/node.o $(OUT_DIR)/codegen.o $(OUT_DIR)/main.o

all: directories $(EXEC)

directories: ${OUT_DIR}

${OUT_DIR}:
    ${MKDIR_P} ${OUT_DIR}

$(EXEC): $(OBJS)
    $(CC) -o $@ $(OBJS) -lcomdlg32

$(OUT_DIR)/y.tab.o $(OUT_DIR)/y.tab.h: src/syntax.y
    $(BISON) -yd -o $(OUT_DIR)/y.tab.c src/syntax.y
    $(CC) $(CFLAGS) -c $(OUT_DIR)/y.tab.c -o $(OUT_DIR)/y.tab.o

$(OUT_DIR)/lex.yy.o: src/lex.l $(OUT_DIR)/y.tab.h
    $(FLEX) -o $(OUT_DIR)/lex.yy.c src/lex.l
    $(CC) $(CFLAGS) -c $(OUT_DIR)/lex.yy.c -o $@

$(OUT_DIR)/symbol_table.o: src/symbol_table.c src/symbol_table.h
    $(CC) $(CFLAGS) -c src/symbol_table.c -o $@

$(OUT_DIR)/codegen.o: src/codegen.c src/codegen.h
    $(CC) $(CFLAGS) -c src/codegen.c -o $@

$(OUT_DIR)/node.o: src/node.c src/node.h
    $(CC) $(CFLAGS) -c src/node.c -o $@

$(OUT_DIR)/main.o: src/main.c src/codegen.h
    $(CC) $(CFLAGS) -c src/main.c -o $@

clean:
    rm -f $(OUT_DIR)/*.o $(OUT_DIR)/*.c $(OUT_DIR)/*.h $(EXEC)
```

```
clean_docs:
    @echo "Eliminando la carpeta src/docs..."
    rm -rf src/docs

# Regla para generar la documentación
generate_docs:
    @echo "Generando la documentación con Doxygen..."
    doxygen src/Doxyfile $(EXEC)

# Regla para abrir la documentación
open_docs:
    @echo "Abriendo la documentación en el navegador..."
    start docs/html/index.html

# Regla principal para documentación que limpia, genera y abre la documentación
docs: clean_docs generate_docs open_docs

.PHONY: all clean directories
```

Anexo I.II – Código Tabla de Símbolos

```
#include "symbol_table.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

Symbol *symbolTable = NULL;

void insert_symbol(char *name, DataType type, int num_parameters) {
    Symbol *existing_symbol = find_symbol(name);
    if (existing_symbol) {
        fprintf(stderr, "Error semantico: Error identificador \"%s\" ya declarado previamente.\n", name);
        return;
    }
    Symbol *s = malloc(sizeof(Symbol));
    if (s == NULL) {
        fprintf(stderr, "Error al asignar memoria para el nuevo símbolo.\n");
        return;
    }
    s->name = strdup(name);
    s->type = type;
    s->next = symbolTable;
    s->num_parameters = num_parameters;
    symbolTable = s;
}

DataType convert_data_type(char* type_str) {
    if (strcmp(type_str, "int") == 0) return TYPE_INT;
    if (strcmp(type_str, "float") == 0) return TYPE_FLOAT;
    if (strcmp(type_str, "char") == 0) return TYPE_CHAR;
    if (strcmp(type_str, "string") == 0) return TYPE_STRING;
```

```
    if (strcmp(type_str, "boolean") == 0) return TYPE_BOOLEAN;
    return TYPE_INT;
}
```

```
Symbol *find_symbol(char *name) {
    for (Symbol *s = symbolTable; s; s = s->next) {
        if (strcmp(s->name, name) == 0) {
            return s;
        }
    }
    return NULL;
}
```

```
int count_parameters(ASTNode *parameter_list) {
    int count = 0;
    ASTNode *current = parameter_list;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```

```
int count_arguments(ASTNode *argument_list) {
    int count = 0;
    ASTNode *current = argument_list;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```

```
int print_symbol_table() {
    Symbol *s = symbolTable;
    if (s == NULL) {
        printf("La tabla de simbolos esta vacia.\n");
        return 0;
    }
    //printf("Tabla de Símbolos:\n");
    while (s != NULL) {
        //printf("Nombre: %s, Tipo: %d\n", s->name, s->type);
        s = s->next;
    }
    return 1; // Indica éxito
}
```

Anexo I.III – Código Analizador Léxico

```
%{
#include "y.tab.h"
#include "symbol_table.h"
#include "node.h"
```

```

int yywrap(void);
%}
%option yylineno
%%

"main" { /*printf("MAIN ");*/ return MAIN; }
"function" { /*printf("FUNCTION ");*/ return FUNCTION; }
"return" { /*printf("RETURN ");*/ return RETURN; }
"if" { /* printf("IF ");*/ return IF; }
"else" { /*printf("ELSE ");*/ return ELSE; }
"for" { /*printf("FOR ");*/ return FOR; }
"while" { /*printf("WHILE ");*/ return WHILE; }
"int" { /*printf("INT ");*/ return TOK_INT; }
"boolean" { /*printf("BOOLEAN ");*/ return TOK_BOOLEAN; }
"string" { /*printf("STRING ");*/ return TOK_STRING; }
"float" { /*printf("FLOAT ");*/ return TOK_FLOAT; }
"char" { /*printf("CHAR ");*/ return TOK_CHAR; }
"true" { /*printf("TRUE ");*/ return TOK_TRUE; }
"false" { /*printf("FALSE ");*/ return TOK_FALSE; }
"array" { /*printf("ARRAY ");*/ return ARRAY; }
"scanf" { /*printf("SCANF ");*/ return SCANF; }
"printf" { /*printf("PRINTF ");*/ return PRINTF; }
"free" { /*printf("FREE ");*/ return FREE; }
"malloc" { /*printf("MALLOC ");*/ return MALLOC; }
[ \t]+ { /* Ignorar espacios en blanco. */ }
\n { /*printf("NEWLINE\n");*/ return NEWLINE; }
[0-9]+ {
    yylval.ival = atoi(yytext);
    /*printf("NUMBER(%s) ", yytext);*/
    return NUMBER;
}
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.sval = strdup(yytext);
    /*printf("IDENTIFIER(%s) ", yytext);*/
    return IDENTIFIER;
}
"+" { /*printf("PLUS ");*/ return PLUS; }
"_" { /*printf("MINUS ");*/ return MINUS; }
"*" { /*printf("TIMES ");*/ return TIMES; }
"/" { /*printf("DIVIDE ");*/ return DIVIDE; }
"=" { /*printf("EQUAL ");*/ return EQUAL; }
";" { /*printf("SEMICOLON ");*/ return SEMICOLON; }
"&" { /*printf("AMPERSAND ");*/ return AMPERSAND; }
"," { /*printf("COMMA ");*/ return COMMA; }
"(" { /*printf("LPAREN ");*/ return LPAREN; }
")" { /*printf("RPAREN ");*/ return RPAREN; }
"{" { /*printf("LBRACE ");*/ return LBRACE; }
"}" { /*printf("RBRACE ");*/ return RBRACE; }
"[" { /*printf("LBRACKET ");*/ return LBRACKET; }
"]" { /*printf("RBRACKET ");*/ return RBRACKET; }
"&&" { /*printf("AND ");*/ return AND; }
"||" { /*printf("OR ");*/ return OR; }
"!" { /*printf("NOT ");*/ return NOT; }
"==" { /*printf("EQ ");*/ return EQ; }

```

```

"!=" { /*printf("NE ");*/ return NE; }
"<" { /*printf("LT ");*/ return LT; }
"<=" { /*printf("LE ");*/ return LE; }
">" { /*printf("GT ");*/ return GT; }
">=" { /*printf("GE ");*/ return GE; }
.|\n { printf("Error token: '%s' no reconocido en la linea %d\n", yytext, yylineno); return
ERROR_TOKEN; }

"//[^\n"]*" { /* Ignorar comentario de una línea */ }

%%
int yywrap(void) {
    return 1;
}

```

Anexo I.IV – Código Analizador Sintáctico

```

%{
#include "node.h"
#include <string.h>
#include <stdio.h>
#include "symbol_table.h"
#include "codegen.h"

extern int yylineno;
DataType current_function_type;
DataType data_type;
void yyerror(const char *s);
void yysemanticerror(const char *s);
int yylex(void);
%}

%token MAIN
%token FUNCTION RETURN FREE MALLOC
%token TOK_INT TOK_BOOLEAN TOK_CHAR TOK_FLOAT TOK_STRING TOK_TRUE TOK_FALSE
ERROR_TOKEN
%token PLUS MINUS TIMES DIVIDE EQUAL SEMICOLON LPAREN RPAREN LBRACE RBRACE COMMA
AMPERSAND AND OR NOT EQ NE LT LE GT GE IF ELSE FOR WHILE LBRACKET RBRACKET ARRAY SCANF
PRINTF
%token NEWLINE
%token <ival> NUMBER
%token <sval> IDENTIFIER

%type <node> declaration_statement assignment_statement if_statement while_statement
for_statement function_call printf_statement scanf_statement expression statement program
%type <sval> type
%type <node> program_body function_body if_body else_body while_body for_body pointer_type
pointer_assignment_statement free_statement array_access
%type <node> function_definition parameter_list parameter argument_list argument_for_list
argument_while_list return_statement array_declaration dimension_list expression_list
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

```



```
%nonassoc '(' ' ')
```

```
%left PLUS MINUS TIMES DIVIDE AND OR EQ NE LT LE GT GE '+' '-' '*' '/'
```

```
%right EQUAL
```

```
%union {
    int ival;
    char* sval;
    char* identifier;
    struct ASTNode* node;
    struct Declaration* declaration;
    struct IfExpr* ifExpr;
    struct WhileExpr* whileExpr;
    struct ForLoop* forLoop;
    struct FunctionCall* functionCall;
    struct {
        struct ASTNode* left;
        struct ASTNode* right;
        char* op;
    } binary;
}
```

```
%%
```

```
program:
```

```
    optional_newlines MAIN optional_newlines LBRACE program_body RBACE optional_newlines
optional_classes{
```

```
    //printf("Programa main procesado correctamente en la linea %d.\n", yylineno);
```

```
    $$ = $5;
```

```
}
```

```
| optional_newlines MAIN optional_newlines LBRACE program_body {
```

```
    printf("Error: falta RBACE para cerrar el main en la linea %d.\n", yylineno);
```

```
    yyerror("Falta RBACE para cerrar el main");
```

```
    yyerrok;
```

```
    $$ = NULL;
```

```
}
```

```
| optional_newlines MAIN optional_newlines {
```

```
    printf("Error: falta LBRACE para abrir el main en la linea %d.\n", yylineno);
```

```
    yyerror("Falta LBRACE abrir el main");
```

```
    $$ = NULL;
```

```
}
```

```
;
```

```
optional_newlines:
```

```
    | optional_newlines NEWLINE
```

```
;
```

```
class_definition:
```

```
    IDENTIFIER optional_newlines LBRACE program_body RBACE optional_newlines
```

```
{
```

```
    if (find_symbol($1) != NULL) {
```

```
        yysemanticerror("Error semantico: Identificador de clase previamente declarado");
```

```
    } else {
```

```
        insert_symbol($1,0,0);
```

```

        //printf("Definiendo clase '%s' en la línea %d.\n", $1, yylineno);
    }
}
;
optional_classes:
| optional_classes class_definition
;
program_body:
{ $$ = NULL; }
| program_body statement NEWLINE { $$ = combine_nodes($1, $2); }
| program_body NEWLINE { $$ = $1; }
;

statement:
function_definition
| declaration_statement
| printf_statement
| scanf_statement
| if_statement
| while_statement
| for_statement
| assignment_statement
| pointer_assignment_statement
| function_call
| return_statement
| free_statement
| array_declaration
| array_access
;

assignment_statement:
IDENTIFIER EQUAL expression SEMICOLON {
    Symbol *symbol = find_symbol($1);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $1);
    } else if ($3 == NULL) {
        yysemanticerror("Error semantico: Expresion no definida");
        printf("Error semantico en la línea %d: Expresion no definida para la asignacion a '%s'.\n",
yylineno, $1);
        $$ = NULL;
    } else {
        // printf("Debug: Tipo del simbolo '%s': %d, Tipo de la expresion: %d\n", $1, symbol->type, $3-
>data_type);
        if (symbol->type != $3->data_type) {
            yysemanticerror("Error semantico: Tipo de identificador incompatible en la asignacion.");
            printf("Error semantico en la línea %d: Tipo de identificador incompatible en la asignacion. Tipo
de '%s': %d, Tipo de expresion: %d\n", yylineno, $1, symbol->type, $3->data_type);
            $$ = NULL;
        } else {
            // printf("Asignando a '%s' el resultado de la expresion en la línea %d.\n", $1, yylineno);
            $$ = create_assignment_node($1, $3);
            generate_quad(OP_ASSIGN, extract_identifier($3), "", $1);
        }
    }
}

```

```

}
| IDENTIFIER EQUAL function_call SEMICOLON {
    Symbol *symbol = find_symbol($1);
    Symbol *func_symbol = find_symbol($3->data.functionCall.functionName);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico en la linea %d: Identificador \"%s\" no declarado previamente.\n",
yylineno, $1);
    } else if (func_symbol == NULL) {
        yysemanticerror("Error semantico: Funcion no declarada previamente");
        printf("Error semantico en la linea %d: Funcion \"%s\" no declarada previamente.\n", yylineno,
$3->data.functionCall.functionName);
    } else if ($3 == NULL) {
        yysemanticerror("Error semantico: Expresion no definida");
        printf("Error semantico en la linea %d: Expresion no definida para la asignacion a '%s'.\n",
yylineno, $1);
        $$ = NULL;
    } else if ($3->data_type == -1) {
        yysemanticerror("Error semantico: Tipo de expresion no definido");
        printf("Error semantico en la linea %d: Tipo de expresion no definido para la asignacion a '%s'.\n",
yylineno, $1);
        $$ = NULL;
    } else if (symbol->type != func_symbol->type) {
        yysemanticerror("Error semantico: Tipo de identificador incompatible en la asignacion.");
        printf("Error semantico en la linea %d: Tipo de identificador incompatible en la asignacion. Tipo
de '%s': %d, Tipo de funcion: %d\n", yylineno, $1, symbol->type, func_symbol->type);
        $$ = NULL;
    } else {
        // printf("Asignando a '%s' el resultado de la llamada a funcion en la linea %d.\n", $1, yylineno);
        $$ = create_assignment_node($1, $3);
        generate_quad(OP_ASSIGN, $3->data.functionCall.functionName, "", $1);
    }
}
}
| IDENTIFIER EQUAL array_access SEMICOLON {
    Symbol *symbol = find_symbol($1);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $1);
    } else {
        // printf("Asignando a '%s' el resultado de la llamada a array en la linea %d.\n", $1, yylineno);
        $$ = create_assignment_node($1, $3);
        generate_quad(OP_ASSIGN, extract_identifier($3), "", $1);
    }
}
}
| array_access EQUAL expression SEMICOLON {
    //printf("Asignando al array el resultado en la linea %d.\n", yylineno);
    $$ = create_assignment_node_array($1, $3);
    generate_quad(OP_ASSIGN, extract_identifier($3), "", extract_identifier($1));
}
}
| array_access EQUAL array_access SEMICOLON {
    // printf("Asignando al array el resultado en la linea %d.\n", yylineno);
    $$ = create_assignment_node_array($1, $3);
    generate_quad(OP_ASSIGN, extract_identifier($3), "", extract_identifier($1));
}
}
| IDENTIFIER EQUAL MALLOC SEMICOLON {

```

```

    if (find_symbol($1) == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $1);
    } else {
        //printf("Asignando al identifier malloc en la linea %d.\n", $1, yylineno);
        $$ = create_assignment_node($1, 0);
    }
}
| IDENTIFIER EQUAL expression {
    printf("Error: falta ';' despues de la asignacion de '%s' en la linea %d.\n", $1, yylineno);
    yyerror("Falta ';' despues de la asignacion");
    yyerrok;
    $$ = NULL;
}
| IDENTIFIER EQUAL function_call {
    printf("Error: falta ';' despues de la llamada a funcion asignada a '%s' en la linea %d.\n", $1,
yylineno);
    yyerror("Falta ';' despues de la llamada a funcion");
    yyerrok;
    $$ = NULL;
}
| IDENTIFIER EQUAL array_access {
    printf("Error: falta ';' despues del array en la linea %d.\n", $1, yylineno);
    yyerror("Falta ';' despues del array");
    yyerrok;
    $$ = NULL;
}
| array_access EQUAL expression {
    printf("Error: falta ';' despues de la asignacion en la linea %d.\n", $1, yylineno);
    yyerror("Falta ';' despues de la asignacion");
    yyerrok;
    $$ = NULL;
}
| IDENTIFIER EQUAL AMPERSAND IDENTIFIER SEMICOLON {
    Symbol *symbol1 = find_symbol($1);
    Symbol *symbol2 = find_symbol($4);
    if (symbol1 == NULL || symbol2 == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico: Error identificador \"%s\" o \"%s\" no declarado previamente.\n", $1,
$4);
    } else {
        printf("Asignando a '%s' la dirección de '%s' en la linea %d.\n", $1, $4, yylineno);
        $$ = create_assignment_node_ampersand($1, $4);
        generate_quad(OP_ASSIGN, $4, "", $1);
    }
}
;

pointer_type:
    TIMES { $$ = create_pointer_node(1); }
| pointer_type TIMES {
    $$ = increase_pointer_level($1);
    // printf("Annadiendo un nivel de puntero, total ahora es %d.\n", $$->data.pointer_level);
}
;

```

pointer_assignment_statement:

```
TIMES IDENTIFIER EQUAL expression SEMICOLON {
    Symbol *symbol = find_symbol($2);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $2);
    } else {
        //printf("Asignacion de puntero para '%s' en linea %d.\n", $2, yylineno);
        $$ = create_pointer_assignment_node($2, $4);
        generate_quad(OP_ASSIGN, extract_identifier($4), "", $2);
    }
}
| TIMES IDENTIFIER EQUAL expression {
    printf("Error: falta ';' despues de la asignacion de '%s' en la linea %d.\n", $2, yylineno);
    yyerror("Falta ';' despues de la asignacion");
    yyerrok;
    $$ = NULL;
}
;
```

declaration_statement:

```
type IDENTIFIER SEMICOLON {
    if (find_symbol($2) != NULL) {
        yysemanticerror("Error semantico: Identificador previamente declarado");
        printf("Error semantico: Error identificador \"%s\" ya declarado previamente.\n", $2);
    } else {
        insert_symbol($2, convert_data_type($1), 0);
        //printf("Declarando variable '%s' de tipo '%s' en la linea %d.\n", $2, $1, yylineno);
        $$ = create_declaration_node($1, $2, 0);
    }
}
| type IDENTIFIER {
    printf("Error: falta ';' despues de la declaracion de '%s' en la linea %d.\n", $2, yylineno);
    yyerror("Falta ';' despues de la declaracion");
    yyerrok;
    $$ = NULL;
}
| type pointer_type IDENTIFIER SEMICOLON {
    if (find_symbol($3) != NULL) {
        yysemanticerror("Error semantico: Identificador previamente declarado");
        printf("Error semantico: Error identificador \"%s\" ya declarado previamente.\n", $3);
    } else {
        insert_symbol($3, convert_data_type($1), 0);
        // printf("Declarando puntero '%s' de tipo '%s' con %d niveles de puntero en la linea %d.\n", $3,
$1, $2->data.pointer_level, yylineno);
        $$ = create_declaration_node($1, $3, $2->data.pointer_level);
    }
}
| type pointer_type IDENTIFIER {
    printf("Error: falta ';' despues de la declaracion del puntero de '%s' en la linea %d.\n", $3, yylineno);
    yyerror("Falta ';' despues de la declaracion del puntero");
    yyerrok;
    $$ = NULL;
}
;
```

```

| type AMPERSAND IDENTIFIER SEMICOLON {
  if (find_symbol($3) != NULL) {
    yysemanticerror("Error semantico: Identificador previamente declarado");
    printf("Error semantico: Error identificador \"%s\" ya declarado previamente.\n", $3);
  } else {
    insert_symbol($3, convert_data_type($1), 0);
    //printf("Declarando variable '%s' de tipo '%s' en la linea %d.\n", $3, $1, yylineno);
    $$ = create_declaration_node($1, $3, 0);
  }
}
| type AMPERSAND IDENTIFIER {
  printf("Error: falta ';' despues de la declaracion de '%s' en la linea %d.\n", $3, yylineno);
  yyerror("Falta ';' despues de la declaracion");
  yyerrok;
  $$ = NULL;
}
;

array_declaration:
  ARRAY type LBRACKET dimension_list RBRACKET IDENTIFIER SEMICOLON {
    if (find_symbol($6) != NULL) {
      yysemanticerror("Error semantico: Identificador previamente declarado");
      printf("Error semantico: Error identificador \"%s\" ya declarado previamente.\n", $6);
    } else {
      insert_symbol($6, convert_data_type($2), 0);
      $$ = create_array_node($2, $4, $6);
      // printf("Declaración de array de tipo '%s' con identificador '%s' en la linea %d.\n", $2, $6,
yylineno);
    }
  }
| ARRAY type LBRACKET dimension_list RBRACKET IDENTIFIER{
  printf("Error: falta ';' al final de la array de '%s' en la linea %d.\n", $2, yylineno);
  yyerror("Falta ';' al final de la array");
  yyerrok;
  $$ = NULL;
}
;

array_access:
  IDENTIFIER LBRACKET expression_list RBRACKET {
    Symbol *symbol = find_symbol($1);
    if (symbol == NULL) {
      yysemanticerror("Error semantico: Identificador no declarado previamente");
      printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $1);
    } else {
      $$ = create_array_access_node($1, $3);
      $$->data_type = symbol->type;
      //printf("Acceso a array '%s' de tipo '%d' en la linea %d.\n", $1, symbol->type, yylineno);
    }
  }
;

expression_list:
  expression
| expression_list COMMA expression {
  $$ = combine_expressions($1, $3);
}

```

```
}  
;  
  
dimension_list:  
  expression {  
    $$ = $1;  
  }  
  | dimension_list COMMA expression {  
    $$ = create_dimension_list($1, $3);  
  }  
  ;  
  
printf_statement:  
  PRINTF IDENTIFIER SEMICOLON {  
    if (find_symbol($2) == NULL) {  
      yysemanticerror("Error semantico: Identificador no declarado previamente");  
      printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $2);  
    } else {  
      //printf("Llamando a 'printf' con '%s' en la linea %d.\n", $2, yylineno);  
      $$ = create_printf_node($2);  
    }  
  }  
  | PRINTF IDENTIFIER {  
    printf("Error: falta ';' despues de un printf en la linea %d.\n", yylineno);  
    yyerror("Falta ';' despues de un printf");  
    yyerrok;  
    $$ = NULL;  
  }  
  | PRINTF pointer_type IDENTIFIER SEMICOLON {  
    if (find_symbol($3) == NULL) {  
      yysemanticerror("Error semantico: Identificador no declarado previamente");  
      printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $3);  
    } else {  
      // printf("Llamando a 'printf' con '%s' en la linea %d.\n", $3, yylineno);  
      $$ = create_printf_node($3);  
    }  
  }  
  | PRINTF pointer_type IDENTIFIER {  
    printf("Error: falta ';' despues de un printf en la linea %d.\n", yylineno);  
    yyerror("Falta ';' despues de un printf");  
    yyerrok;  
    $$ = NULL;  
  }  
  ;  
  
scanf_statement:  
  SCANF IDENTIFIER SEMICOLON {  
    if (find_symbol($2) == NULL) {  
      yysemanticerror("Error semantico: Identificador no declarado previamente");  
      printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $2);  
    } else {  
      //printf("Llamando a 'scanf' con '%s' en la linea %d.\n", $2, yylineno);  
      $$ = create_scanf_node($2);  
    }  
  }  
  ;
```

```

| SCANF array_access SEMICOLON {
    //printf("Llamando a 'scanf' con '%s' en la linea %d.\n", $2, yylineno);
    $$ = create_scanf_node_array($2);
}
| SCANF IDENTIFIER {
    printf("Error: falta ';' despues de un f scanf la linea %d.\n", yylineno);
    yyerror("Falta ';' despues de un scanf");
    yyerrok;
    $$ = NULL;
}
| SCANF array_access {
    printf("Error: falta ';' despues de un scanf en la linea %d.\n", yylineno);
    yyerror("Falta ';' despues de un scanf");
    yyerrok;
    $$ = NULL;
}
;

free_statement:
    FREE IDENTIFIER SEMICOLON {
        if (find_symbol($2) == NULL) {
            yysemanticerror("Error semantico: Identificador no declarado previamente");
            printf("Error semantico: Error identificador \"%s\" no declarado previamente.\n", $2);
        } else {
            // printf("Llamando a 'free' con '%s' en la linea %d.\n", $2, yylineno);
            $$ = create_free_node($2);
        }
    }
;

| FREE IDENTIFIER {
    printf("Error: falta ';' despues de un free en la linea %d.\n", yylineno);
    yyerror("Falta ';' despues de un free");
    yyerrok;
    $$ = NULL;
}
;

return_statement:
    RETURN IDENTIFIER SEMICOLON {
        Symbol *symbol = find_symbol($2);
        if (symbol == NULL) {
            yysemanticerror("Error semantico: Identificador no declarado previamente");
            printf("Error semantico: Identificador \"%s\" no declarado previamente.\n", $2);
        } else if (symbol->type != current_function_type) {
            yysemanticerror("Error semantico: Tipo de retorno incompatible.");
            printf("Error semantico: Tipo de retorno incompatible. Tipo de '%s': %d, Tipo esperado: %d\n",
$2, symbol->type, current_function_type);
            $$ = NULL;
        } else {
            // printf("Declaracion de retorno en la linea %d. Tipo de '%s': %d\n", yylineno, $2, symbol->type);
            $$ = create_return_node($2, NULL);
        }
    }
;

| RETURN IDENTIFIER {
    printf("Error: falta ';' despues de un return en la linea %d.\n", yylineno);
    yyerror("Falta ';' despues de un return");
}
;

```



```

yyerrok;
$$ = NULL;
}
| RETURN LPAREN function_call RPAREN SEMICOLON {
    Symbol *symbol = find_symbol($3->data.functionCall.functionName);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador de funcion no declarado previamente");
        $$ = NULL;
    } else if (symbol->type != current_function_type) {
        yysemanticerror("Error semantico: Tipo de retorno incompatible.");
        printf("Error semantico: Tipo de retorno incompatible. Tipo de funcion '%s': %d, Tipo esperado: %d\n", $3->data.functionCall.functionName, symbol->type, current_function_type);
        $$ = NULL;
    } else {
        // printf("Declaracion de retorno en la linea %d.\n", yylineno);
        $$ = create_return_node(NULL, $3);
    }
}
| RETURN NUMBER SEMICOLON {
    if (current_function_type != TYPE_INT) {
        yysemanticerror("Error semantico: Tipo de retorno incompatible.");
        printf("Error semantico: Tipo de retorno incompatible. Tipo de retorno: int, Tipo esperado: %d\n", current_function_type);
        $$ = NULL;
    } else {
        //printf("Declaracion de retorno en la linea %d.\n", yylineno);
        $$ = create_constant_node($2);
    }
}
| RETURN TOK_TRUE SEMICOLON {
    if (current_function_type != TYPE_BOOLEAN) {
        yysemanticerror("Error semantico: Tipo de retorno incompatible.");
        printf("Error semantico: Tipo de retorno incompatible. Tipo de retorno: boolean, Tipo esperado: %d\n", current_function_type);
        $$ = NULL;
    } else {
        $$ = create_constant_node(1);
        //printf("Valor booleano 'true' detectado en la linea %d.\n", yylineno);
    }
}
| RETURN TOK_FALSE SEMICOLON {
    if (current_function_type != TYPE_BOOLEAN) {
        yysemanticerror("Error semantico: Tipo de retorno incompatible.");
        printf("Error semantico: Tipo de retorno incompatible. Tipo de retorno: boolean, Tipo esperado: %d\n", current_function_type);
        $$ = NULL;
    } else {
        $$ = create_constant_node(0);
        // printf("Valor booleano 'false' detectado en la linea %d.\n", yylineno);
    }
}
;

```

function_definition:

FUNCTION type IDENTIFIER LPAREN parameter_list RPAREN optional_newlines LBRACE {

```

    if (find_symbol($3) != NULL) {
        yysemanticerror("Error semantico: Identificador de funcion previamente declarado");
    } else {
        int num_parameters = count_parameters($5);
        current_function_type = convert_data_type($2);
        insert_symbol($3, convert_data_type($2), num_parameters);
        // printf("Declaracion de funcion '%s' con tipo '%s' en la linea %d.\n", $3, $2, yylineno);
    }
}
function_body RBACE {
    //printf("Definicion de funcion '%s' con tipo '%s' y retorno asegurado en la linea %d.\n", $3, $2,
yylineno);
}
| FUNCTION IDENTIFIER LPAREN parameter_list RPAREN optional_newlines LBACE function_body
RBACE{
    printf("Error: falta definir el tipo de funcion en la linea %d.\n", yylineno);
    yyerror("Falta definir el tipo de funcion");
    yyerrok;
    $$ = NULL;
}
| FUNCTION type LPAREN parameter_list RPAREN optional_newlines LBACE function_body RBACE{
    printf("Error: falta definir el identificador de funcion en la linea %d.\n", yylineno);
    yyerror("Falta definir el identificador de function");
    yyerrok;
    $$ = NULL;
}
| FUNCTION type IDENTIFIER parameter_list RPAREN optional_newlines LBACE function_body
RBACE{
    printf("Error: falta LPAREN para definir parametros de la funcion en la linea %d.\n", yylineno);
    yyerror("Falta LPAREN para definir parametros de la funcion");
    yyerrok;
    $$ = NULL;
}
;

parameter_list:
    parameter
    | parameter_list SEMICOLON parameter {
        $$ = combine_parameter_nodes($1, $3);
    }
;

parameter:
    { $$ = NULL; }
    | type IDENTIFIER {
        if (find_symbol($2) != NULL) {
            yysemanticerror("Error semantico: Identificador previamente declarado");
            printf("Error semantico: Error identificador \"%s\" ya declarado previamente.\n", $2);
        } else {
            insert_symbol($2, convert_data_type($1), 0);
            $$ = create_parameter_node($1, $2);
        }
    }
;

function_body:

```

```
{ $$ = NULL; }  
| function_body statement NEWLINE { $$ = combine_nodes($1, $2); }  
| function_body NEWLINE  
;
```

function_call:

```
IDENTIFIER LPAREN argument_list RPAREN {  
    Symbol *symbol = find_symbol($1);  
    if (symbol == NULL) {  
        yysemanticerror("Error semantico: Identificador no declarado previamente");  
        printf("Error semantico: Identificador \"%s\" no declarado previamente.\n", $1);  
    } else {  
        int num_arguments = count_arguments($3);  
        if (num_arguments != symbol->num_parameters) {  
            yysemanticerror("Error semantico: Numero incorrecto de argumentos en la llamada a la  
funcion");  
            printf("Error semantico: Numero incorrecto de argumentos en la llamada a la funcion '%s'.  
Esperados %d, encontrados %d en la linea %d.\n", $1, symbol->num_parameters, num_arguments,  
yylineno);  
        } else {  
            // printf("Llamada a la funcion '%s' con argumentos en la linea %d.\n", $1, yylineno);  
            $$ = create_function_call_node($1, $3);  
        }  
    }  
}  
;
```

argument_list:

```
{ $$ = NULL; }  
| expression  
| argument_list COMMA expression {  
    $$ = combine_argument_nodes($1, $3);  
}  
;
```

type:

```
TOK_INT {  
    $$ = strdup("int");  
    // printf("Tipo detectado: int en la linea %d.\n", yylineno);  
}  
| TOK_BOOLEAN {  
    $$ = strdup("boolean");  
    //printf("Tipo detectado: boolean en la linea %d.\n", yylineno);  
}  
| TOK_FLOAT {  
    $$ = strdup("float");  
    // printf("Tipo detectado: boolean en la linea %d.\n", yylineno);  
}  
| TOK_CHAR {  
    $$ = strdup("char");  
    //printf("Tipo detectado: boolean en la linea %d.\n", yylineno);  
}  
| TOK_STRING {  
    $$ = strdup("string");  
}
```

```

    // printf("Tipo detectado: boolean en la linea %d.\n", yylineno);
}
;

if_statement:
    IF LPAREN expression RPAREN optional_newlines LBRACE if_body RBRACE optional_newlines ELSE
optional_newlines LBRACE else_body RBRACE {
    // printf("Entrando a la sentencia 'if-else' en linea %d.\n", yylineno);
    char label_else[20], label_end[20];
    sprintf(label_else, "L%d", new_label());
    sprintf(label_end, "L%d", new_label());
    generate_quad(OP_JUMP_IF_FALSE, extract_identifier($3), "", label_else);
    $$ = $7;
    generate_quad(OP_JUMP, "", "", label_end);
    generate_quad(OP_LABEL, label_else, "", "");
    $$ = $13;
    generate_quad(OP_LABEL, label_end, "", "");
}
| IF LPAREN expression RPAREN optional_newlines LBRACE if_body RBRACE optional_newlines
LBRACE else_body RBRACE {
    printf("Error: falta ELSE para definir parametros de la funcion en la linea %d.\n", yylineno);
    yyerror("Falta ELSE para definir parametros de la funcion");
    yyerrok;
    $$ = NULL;
}
;

if_body:
    { $$ = NULL; }
    | if_body statement NEWLINE { $$ = combine_nodes($1, $2); }
    | if_body NEWLINE { $$ = $1; }
;

else_body:
    { $$ = NULL; }
    | else_body statement NEWLINE { $$ = combine_nodes($1, $2); }
    | else_body NEWLINE { $$ = $1; }
;

while_statement:
    WHILE LPAREN argument_while_list RPAREN optional_newlines LBRACE while_body RBRACE {
        //printf("Procesando una sentencia 'while' en la linea %d.\n", yylineno);
        char label_start[20], label_end[20];
        sprintf(label_start, "L%d", new_label());
        sprintf(label_end, "L%d", new_label());
        generate_quad(OP_LABEL, label_start, "", "");
        generate_quad(OP_JUMP_IF_FALSE, extract_identifier($3), "", label_end);
        $$ = $7;
        generate_quad(OP_JUMP, "", "", label_start);
        generate_quad(OP_LABEL, label_end, "", "");
    }
;

argument_while_list:
    { $$ = NULL; }
    | expression

```

```

| argument_while_list SEMICOLON expression {
    $$ = combine_argument_nodes($1, $3);
}
;

while_body:
{ $$ = NULL; }
| while_body statement NEWLINE { $$ = combine_nodes($1, $2); }
| while_body NEWLINE
;

for_statement:
FOR LPAREN argument_for_list RPAREN optional_newlines LBRACE for_body RBRACE {
    // printf("Procesando una sentencia 'for' en la linea %d.\n", yylineno);
    char label_start[20], label_end[20], label_increment[20];
    sprintf(label_start, "L%d", new_label());
    sprintf(label_end, "L%d", new_label());
    sprintf(label_increment, "L%d", new_label());
    $$ = $3;
    generate_quad(OP_LABEL, label_start, "", "");
    generate_quad(OP_JUMP_IF_FALSE, extract_identifier($3), "", label_end);
    $$ = $7;
    generate_quad(OP_LABEL, label_increment, "", "");
    $$ = $3;
    generate_quad(OP_JUMP, "", "", label_start);
    generate_quad(OP_LABEL, label_end, "", "");
}
;

argument_for_list:
{ $$ = NULL; }
| expression
| argument_for_list SEMICOLON expression {
    $$ = combine_argument_nodes($1, $3);
}
;

for_body:
{ $$ = NULL; }
| for_body statement NEWLINE { $$ = combine_nodes($1, $2); }
| for_body NEWLINE
;

expression:
expression PLUS expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Operacion PLUS solo permitida entre enteros.");
        printf("Error semantico: Operacion PLUS solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Operacion PLUS en linea %d.\n", yylineno);
        $$ = create_binary_op_node("+", $1, $3);
    }
}
;

```

```

    $$->data_type = TYPE_INT;
    generate_quad(OP_ADD, extract_identifier($1), extract_identifier($3), extract_identifier($$));
}
}
| expression MINUS expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Operacion MINUS solo permitida entre enteros.");
        printf("Error semantico: Operacion MINUS solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Operacion MINUS en linea %d.\n", yylineno);
        $$ = create_binary_op_node("-", $1, $3);
        $$->data_type = TYPE_INT;
        generate_quad(OP_SUB, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression TIMES expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Operacion TIMES solo permitida entre enteros.");
        printf("Error semantico: Operacion TIMES solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Operacion TIMES en linea %d.\n", yylineno);
        $$ = create_binary_op_node("*", $1, $3);
        $$->data_type = TYPE_INT;
        generate_quad(OP_MUL, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression DIVIDE expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Operacion DIVIDE solo permitida entre enteros.");
        printf("Error semantico: Operacion DIVIDE solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Operacion DIVIDE en linea %d.\n", yylineno);
        $$ = create_binary_op_node("/", $1, $3);
        $$->data_type = TYPE_INT;
        generate_quad(OP_DIV, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression EQUAL expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
    }
}

```

```

    $$ = NULL;
} else if ($1->data_type != $3->data_type) {
    yysemanticerror("Error semantico: Comparacion '=' solo permitida entre operandos del mismo
tipo.");
    printf("Error semantico: Comparacion '=' solo permitida entre operandos del mismo tipo. Tipo de
operando 1: %d, Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
    $$ = NULL;
} else {
    // printf("Comparacion '=' en linea %d.\n", yylineno);
    $$ = create_binary_op_node("=", $1, $3);
    $$->data_type = TYPE_BOOLEAN;
    generate_quad(OP_EQ, extract_identifier($1), extract_identifier($3), extract_identifier($$));
}
}
| expression EQ expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != $3->data_type) {
        yysemanticerror("Error semantico: Comparacion '==' solo permitida entre operandos del mismo
tipo.");
        printf("Error semantico: Comparacion '==' solo permitida entre operandos del mismo tipo. Tipo de
operando 1: %d, Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Comparacion '==' en linea %d.\n", yylineno);
        $$ = create_binary_op_node("==", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_EQ, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression NE expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != $3->data_type) {
        yysemanticerror("Error semantico: Comparacion '!=' solo permitida entre operandos del mismo
tipo.");
        printf("Error semantico: Comparacion '!=' solo permitida entre operandos del mismo tipo. Tipo de
operando 1: %d, Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Comparacion '!=' en linea %d.\n", yylineno);
        $$ = create_binary_op_node("!=", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_NE, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression LT expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Comparacion '<' solo permitida entre enteros.");

```

```

        printf("Error semantico: Comparacion '<' solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Comparacion '<' en linea %d.\n", yylineno);
        $$ = create_binary_op_node("<", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_LT, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| array_access LT expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Comparacion '<' solo permitida entre enteros.");
        printf("Error semantico: Comparacion '<' solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Comparacion '<' en linea %d.\n", yylineno);
        $$ = create_binary_op_node("<", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_LT, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression LE expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Comparacion '<=' solo permitida entre enteros.");
        printf("Error semantico: Comparacion '<=' solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Comparacion '<=' en linea %d.\n", yylineno);
        $$ = create_binary_op_node("<=", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_LE, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression GT expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Comparacion '>' solo permitida entre enteros.");
        printf("Error semantico: Comparacion '>' solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Comparacion '>' en linea %d.\n", yylineno);
        $$ = create_binary_op_node(">", $1, $3);
        $$->data_type = TYPE_BOOLEAN;

```



```

        generate_quad(OP_GT, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression GE expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Comparacion inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_INT || $3->data_type != TYPE_INT) {
        yysemanticerror("Error semantico: Comparacion '>=' solo permitida entre enteros.");
        printf("Error semantico: Comparacion '>=' solo permitida entre enteros. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Comparacion '>=' en linea %d.\n", yylineno);
        $$ = create_binary_op_node(">=", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_GE, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression AND expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion lógica inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_BOOLEAN || $3->data_type != TYPE_BOOLEAN) {
        yysemanticerror("Error semantico: Operacion AND solo permitida entre booleanos.");
        printf("Error semantico: Operacion AND solo permitida entre booleanos. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        // printf("Operacion AND en linea %d.\n", yylineno);
        $$ = create_binary_op_node("&&", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_AND, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| expression OR expression {
    if ($1 == NULL || $3 == NULL) {
        yyerror("Operacion lógica inválida debido a operandos no definidos.");
        $$ = NULL;
    } else if ($1->data_type != TYPE_BOOLEAN || $3->data_type != TYPE_BOOLEAN) {
        yysemanticerror("Error semantico: Operacion OR solo permitida entre booleanos.");
        printf("Error semantico: Operacion OR solo permitida entre booleanos. Tipo de operando 1: %d,
Tipo de operando 2: %d\n", $1->data_type, $3->data_type);
        $$ = NULL;
    } else {
        //printf("Operacion OR en linea %d.\n", yylineno);
        $$ = create_binary_op_node("||", $1, $3);
        $$->data_type = TYPE_BOOLEAN;
        generate_quad(OP_OR, extract_identifier($1), extract_identifier($3), extract_identifier($$));
    }
}
| NUMBER {
    //printf("Numero detectado: %d en la linea %d.\n", $1, yylineno);
    $$ = create_constant_node($1);
    $$->data_type = TYPE_INT;
}

```

```
}
| IDENTIFIER {
    Symbol *symbol = find_symbol($1);
    if (symbol == NULL) {
        yysemanticerror("Error semantico: Identificador no declarado previamente");
        printf("Error semantico en la linea %d: Identificador \"%s\" no declarado previamente.\n",
yylineno, $1);
        $$ = NULL;
    } else {
        //printf("Identificador detectado: %s en la linea %d. Tipo: %d\n", $1, yylineno, symbol->type);
        $$ = create_identifier_node($1);
        $$->data_type = symbol->type; // Asegurar que el tipo se establece correctamente
    }
}
| TOK_TRUE {
    $$ = create_constant_node(1);
    $$->data_type = TYPE_BOOLEAN;
    // printf("Valor booleano 'true' detectado en la linea %d.\n", yylineno);
}
| TOK_FALSE {
    $$ = create_constant_node(0);
    $$->data_type = TYPE_BOOLEAN;
    //printf("Valor booleano 'false' detectado en la linea %d.\n", yylineno);
}
| LPAREN expression RPAREN {
    // printf("Evaluando expresion entre parentesis en la linea %d.\n", yylineno);
    $$ = $2;
}
| TIMES expression {
    // printf("Evaluando expresion con punteros en la linea %d.\n", yylineno);
    $$ = $2;
}
| error {
    yyerror("Error en la expresion");
}
;
%%
```

Anexo I.V – Código Analizador Semántico

En Anexo I.IV

Anexo I.VI – Código Generación de Código

En Anexo I.IV

Anexo II – Gramática Alfa

```

1  <programa>      ::=  main { <declaraciones> <funciones> <sentencias> }
2  <declaraciones> ::=  <declaracion>
3                      |  <declaracion> <declaraciones>
4  <declaracion>   ::=  <clase> <identificadores> ;
5  <clase>         ::=  <clase_escalar>
6                      |  <clase_puntero>
7                      |  <clase_vector>
8                      |  <clase_conjunto>
9  <clase_escalar> ::=  <tipo>
10 <tipo>          ::=  int
11                  |  boolean
12                  |  float
13 <clase_puntero> ::=  <tipo> *
14                  |  <clase_puntero> *
15 <clase_vector>  ::=  array <tipo> [ <constante_entera> ]
16                  |  array <tipo>[<constante_entera>, <constante_entera>]
17 <clase_conjunto> ::=  set of <constante_entera>
18 <identificadores> ::=  <identificador>
19 <funciones>     ::=  <funcion> <funciones>
20                  |  <funcion> ::= function <tipo> <identificador> (
21 <parametros_funcion> ) {<declaraciones_funcion> <sentencias> }
22 <parametros_funcion> ::=  <parametro_funcion>
23 <resto_parametros_funcion>
24 <resto_parametros_funcion> ::=  ; <parametro_funcion>
25 <resto_parametros_funcion>
26 <parametro_funcion> ::= <tipo> <identificador>
27 <declaraciones_funcion> ::= <declaraciones>
28 <sentencias> ::=  <sentencia>
29                  |  <sentencia> <sentencias>
30 <sentencia> ::=  <sentencia_simple> ;
31                  |  <bloque>
32 <sentencia_simple> ::=  <asignacion>
33                  |  <lectura>
34                  |  <escritura>
35                  |  <liberacion>
36                  |  <retorno_funcion>
37                  |  <operacion_conjunto>
38 <bloque> ::=  <condicional>
39                  |  <bucle>
40                  |  <seleccion>
41 <asignacion> ::=  <identificador> = <exp>

```

```

42      | <elemento_vector> = <exp>
43      | <acceso> = <exp>
44      | <identificador> = malloc
45      | <identificador> = & <identificador>
46  <elemento_vector> ::= <identificador> [ <exp> ]
47      | <identificador> [ <exp> , <exp> ]
48  <condicional> ::= if ( <exp> ) { <sentencias> }
49      | if ( <exp> ) { <sentencias> } else { <sentencias> }
50  <bucle> ::= while ( <exp> ) { <sentencias> }
51      | for ( <identificador> = <exp> ; <exp> ) { <sentencias> }
52  <lectura> ::= scanf <identificador>
53      | scanf <elemento_vector>
54  <escritura> ::= printf <exp>
55  <liberacion> ::= free <identificador>
56  <acceso> ::= * <identificador>
57      | * <acceso>
58  <retorno_funcion> ::= return <exp>
59  <exp> ::= <exp> + <exp>
60      | <exp> - <exp>
61      | <exp> / <exp>
62      | <exp> * <exp>
63      | - <exp>
64      | <exp> && <exp>
65      | <exp> || <exp>
66      | ! <exp>
67      | <identificador>
68      | <constante>
69      | ( <exp> )
70      | ( <comparacion> )
71      | <acceso>
72      | <elemento_vector>
73
74
75  <comparacion> ::= <exp> == <exp>
76      | <exp> != <exp>
77      | <exp> <= <exp>
78      | <exp> >= <exp>
79      | <exp> < <exp>
80      | <exp> > <exp>
81  <constante> ::= <constante_logica>
82      | <constante_entera>
83      | <constante_real>
84  <constante_logica> ::= true
85      | false
86  <constante_entera> ::= <numero>
87  <numero> ::= <digito>

```

```
88 | <numero> <digito>
89 <identificador> ::= <letra>
90 <letra> ::= a | b | ... | z | A | B | ... | Z
91 <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- El lenguaje permite incluir comentarios empezando por // (son comentarios de una sola línea).