



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**Diseño e implementación de
un lenguaje de programación
y su compilador**



Presentado por Adrián Zamora Sánchez
en Universidad de Burgos — 8 de enero
de 2026

Tutor: Dr. César Ignacio García Osorio



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. César Ignacio García Osorio, profesor del departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Adrián Zamora Sánchez, con DNI 48838775T, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado «Diseño e implementación de un lenguaje de programación y su compilador».

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 8 de enero de 2026

Vº. Bº. del Tutor: César Ignacio
García Osorio

Resumen

En este proyecto se aborda la creación de un lenguaje de programación, herramienta fundamental dentro del campo de la ingeniería informática. Para que este lenguaje sea utilizable, se implementará un compilador propio, capaz de llevar este lenguaje a ejecución. Este proyecto se limitará a algunas funcionalidades clave, puesto que querer abordar cuestiones, como OOP, podrían aumentar la complejidad del proyecto demasiado.

Para la realización del proyecto se tratan de utilizar las herramientas más modernas y extendidas en la creación de lenguajes y compiladores: ANTLR, LLVM, CMake y C++ como lenguaje de programación.

Descriptores

Compilador, lenguaje de programación, ANTLR, LLVM

Abstract

This project focuses on the creation of a programming language, a fundamental tool within the field of computer engineering. To make this language usable, a custom compiler will be implemented, capable of executing programs written in it. The scope of the project will be limited to a set of key functionalities, as tackling more advanced features, such as OOP, could increase the complexity beyond what is feasible.

For the development of the project, the most modern and widely adopted tools in language and compiler design will be used: ANTLR, LLVM, CMake, and C++ as the programming language.

Keywords

Compiler, programmin language, ANTLR, LLVM

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	vi
1. Introducción	1
2. Objetivos del proyecto	5
3. Conceptos teóricos	7
3.1. Introducción	7
3.2. El proceso de compilación	7
3.3. Infraestructura LLVM y modelo de compilación	11
4. Técnicas y herramientas	13
4.1. Metodología de desarrollo	13
4.2. Metodología de programación	14
4.3. Calidad del código	15
4.4. Elección de herramientas	15
5. Aspectos relevantes del proyecto	21
5.1. Evaluación del rendimiento	34
6. Trabajos relacionados	37
7. Conclusiones y líneas de trabajo futuras	39

Índice de figuras

4.1. Imagen con la visualización del tablero Kanban en ZenHub	14
5.1. Ejemplo de visualización de la expresión $2 + 3 * 5$ con la flag <code>--visualizeAST</code>	25
5.2. Ejemplo de tabla de símbolos generada por el analizador semántico	26
5.3. Gráfica comparativa del rendimiento en el cálculo de números primos	35

Índice de tablas

5.1. Correspondencia entre tipos del AST y tipos de LLVM IR	28
5.2. Tiempos de ejecución en el cálculo de números primos	34

1. Introducción

En la informática moderna estamos acostumbrados a disponer de numerosos lenguajes de programación, los cuales empleamos sin detenernos a analizar más allá de sus potenciales usos, el rendimiento que nos ofrecen o las librerías de las que dispone. Sin embargo, estas complejas herramientas cuentan con un gran trabajo detrás, el cual permite a los programadores abstraerse de complejos conceptos sobre lenguajes o computadores, pudiendo dedicar su tiempo únicamente a la tarea de la programación.

La creación de los compiladores es una ciencia que ha evolucionado mucho a lo largo de los años, no solo por el surgimiento de nuevos paradigmas de lenguajes y sus compiladores asociados, muchos lenguajes que llevan con nosotros desde los 70s han experimentado cambios significativos en sus sintaxis y compiladores, además de versiones alternativas, cada una enfocada en unos objetivos diferentes que se traduce a nuevos retos y formas de entender los lenguajes y sus procesos de compilación.

Ejemplos de proyectos que comenzaron siendo pequeños lenguajes de programación o lenguajes de dominio concreto que han crecido hasta ser herramientas ampliamente utilizadas son:

- **HTML + CSS:** HTML fue creado por Tim Berners-Lee a principios de los años 90 como el lenguaje estándar para estructurar documentos en la World Wide Web, permitiendo hipervínculos y texto marcado; posteriormente, CSS fue propuesto por Håkon Wium Lie y estandarizado en 1996 por W3C con el objetivo de separar claramente el contenido de su presentación, enriqueciendo enormemente el diseño visual de las páginas web. Hoy día disponen de importantes *frameworks* y su impacto ha inspirado llegado a otros lenguajes y herramientas fuera del entorno web. [27, 12]

- **MATLAB:** Diseñado en los 80 por Jack Little y Cleve Moler para ser un lenguaje específico de álgebra lineal [26]. Hoy día ha crecido como una herramienta ampliamente utilizada en ingeniería, robótica, procesamiento de señales, visión por computadora, etc.
- **SQL:** Comenzó en los 70 como un lenguaje que IBM desarrollaría para hacer consultas en sus bases de datos relacionales [28], sin embargo, fue adoptado como un estándar ISO/IEC [8] en la consulta de datos, que lo ha llevado a estar presente en prácticamente todos los motores de bases de datos modernos.
- **JavaScript:** Lenguaje creado a mediados de los 90, cuando los navegadores y entornos web empiezan a despegar. Su primera versión la hace Brendan Eich [29] en tan solo diez días, su primer nombre es "Mocha" pero por cuestión de marketing deciden aprovechar la fama de Java y lo llaman JavaScript. Con el crecimiento de la web y los navegadores termina siendo estandarizado bajo el estándar ECMAScript [7].
- **Python:** Guido Van Rossum comenzó su desarrollo a finales de los 80, con el único objetivo de crear un lenguaje que "fácil de usar, incluso para programadores no profesionales-[23]. En su primera versión es adoptado por numerosas universidades como herramienta científica. El lenguaje ha crecido hasta su versión 3, siendo uno de los lenguajes más usados, con numerosos módulos que lo convierten en una herramienta extremadamente versátil.

Memoria

La memoria ofrece una visión completa del proyecto, incluyendo la definición de objetivos, el marco teórico necesario para entender el presente documento y sus anexos, las técnicas y herramientas utilizadas, así como los aspectos más relevantes del desarrollo. También se enmarca este trabajo con otros proyectos o trabajos relacionados que han servido como una base para facilitar el conocimiento acerca de este área de estudio. Finalmente, se presentan las conclusiones obtenidas, así como posibles líneas de trabajo futuras donde se sugieren mejoras o ampliaciones del trabajo realizado.

Anexos

En el apartado de anexos, se complementa la información de la memoria, proporcionando una documentación más detallada del proyecto, como el plan de proyecto, las especificación de requisitos, el diseño del sistema, manuales

de programación, manuales de uso para el usuario final y, finalmente, la justificación de su valor educativo y curricular.

2. Objetivos del proyecto

- **Diseñar un lenguaje de programación:** Definir nuevo lenguaje de programación, estableciendo sus estructuras básicas, tipos de datos, operadores y reglas gramaticales.
- **Desarrollar un compilador capaz de reconocer el lenguaje:** Implementar un compilador con un diseño modular y moderno, capaz de analizar y procesar el código fuente del lenguaje, abarcando las fases de análisis léxico, sintáctico, semántico, generación de una representación intermedia (IR por sus siglas en inglés), optimización y finalmente generación de un ejecutable.
- **Estructurar de forma modular el proceso de compilación:** Separar las diferentes fases de compilación manteniendo las responsabilidades bien definidas dentro de cada una de las fases, evitando así una sobrecarga de responsabilidades que dificulte realizar ampliaciones o modificaciones.
- **Generación y gestión de errores:** Incorporar un sistema de detección y reporte de errores léxicos, sintácticos y semánticos, proporcionando mensajes claros y detallados que faciliten la depuración del código.
- **Herramienta de análisis y depuración visual:** Desarrollar una utilidad visual que permita representar y explorar estructuras internas del compilador, como el árbol de sintaxis abstracta o la tabla de símbolos, facilitando la comprensión y depuración del lenguaje.
- **Generación de IR basada en el ecosistema LLVM:** Implementar la traducción del lenguaje diseñado a una representación intermedia

basada en LLVM IR [10], permitiendo la optimización y posterior generación de código máquina para distintas arquitecturas.

- **Implementación de una biblioteca estándar:** Desarrollar una pequeña librería estándar, es decir, un conjunto de funciones básicas que proporcionen utilidades estándar dentro del lenguaje diseñado.
- **Proporcionar al menos una funcionalidad específica del lenguaje:** Se trabajará en la implementación de un sistema de ejecución de código basado en eventos y tiempo. Este será un factor diferenciador y experimental frente a otros lenguajes de propósito general.
- **Proporcionar un runtime personalizado:** Para poder implementar las funcionalidades específicas del lenguaje e indagar en otro aspecto académico de los lenguajes de programación, se implementará un runtime diseñado específicamente para poder ejecutar el código generado.

3. Conceptos teóricos

3.1. Introducción

Para comprender adecuadamente el proceso de compilación y las decisiones de diseño tomadas en este proyecto, es necesario introducir una serie de conceptos teóricos relacionados con la construcción de compiladores. Dado que los miembros del tribunal no tienen por qué estar familiarizados con la implementación interna de estos sistemas, en este capítulo se describen de forma general las distintas fases que componen un compilador moderno, así como las infraestructuras y técnicas empleadas habitualmente en su desarrollo.

El objetivo de esta sección no es presentar un glosario de definiciones aisladas, sino ofrecer una visión global del compilador como sistema, explicando cómo fluye la información entre las distintas etapas y cómo estas colaboran para transformar un programa escrito en un lenguaje de alto nivel en código ejecutable.

A lo largo del capítulo se describen las fases del proceso de compilación, el uso de representaciones intermedias y el papel de la infraestructura LLVM en el proyecto. En el capítulo 5 se presentan ejemplos concretos de este proceso, ilustrados mediante fragmentos de código del lenguaje desarrollado y las estructuras internas generadas durante la compilación.

3.2. El proceso de compilación

Un compilador es un sistema software cuya función principal es traducir un programa escrito en un lenguaje fuente a una forma equivalente que pueda ser ejecutada por un computador. Para facilitar su diseño, manteni-

miento y extensión, la mayoría de compiladores modernos se estructuran como una secuencia de **fases** bien definidas, donde cada una cumple una responsabilidad concreta.

Análisis léxico

La primera etapa del proceso de compilación es el análisis léxico. En esta fase, el código fuente se procesa como una secuencia de caracteres que se agrupan en unidades significativas llamadas *tókenes* o *lexemas*. Cada token representa elementos básicos del lenguaje, como identificadores, palabras clave, literales u operadores.

El resultado del análisis léxico es una secuencia estructurada de tókenes que será utilizada por la fase de análisis sintáctico. Separar esta fase del resto permite simplificar el diseño del compilador y detectar errores léxicos de forma temprana.

Análisis sintáctico

A partir de la secuencia de tókenes generada por el análisis léxico, el análisis sintáctico se encarga de verificar que el programa cumple las reglas gramaticales del lenguaje. Para ello, los tókenes se agrupan siguiendo una gramática formal, produciendo una representación jerárquica del programa.

Construcción del AST

El Árbol de sintaxis abstracta o *Abstract Syntax Tree* (AST), es una estructura de datos arbórea donde cada nodo representa un lexema, además, almacena información importante sobre este contexto sintáctico, eliminando detalles sintácticos innecesarios y facilitando su posterior análisis y transformación. El resultado final de este proceso representa la estructura lógica completa del programa, en el capítulo 5 se puede encontrar un ejemplo visual 4.4 de un AST.

Análisis semántico

Una vez construido el AST, el compilador realiza el análisis semántico, cuya función es comprobar que el programa tiene sentido desde el punto de vista del lenguaje, más allá de su corrección sintáctica. En esta fase se verifican aspectos como el uso correcto de tipos, la declaración previa de identificadores o la coherencia de las expresiones.

Durante el análisis semántico se construye y mantiene la **tabla de símbolos**, una estructura de datos que asocia cada identificador del programa con información relevante como su tipo, categoría y ámbito de validez. Esta información será esencial en las fases posteriores de generación de código.

Tipado

El «tipado» en un lenguaje de programación se refiere a los tipos de datos soportados por el lenguaje. Los lenguajes se diferencian entre lenguajes de tipado dinámico y de tipado estático, los primeros permiten mayor libertad con respecto al uso de tipos, implementando un sistema de resolución de tipos en tiempo de ejecución, mientras que los segundos requieren conocer el uso de tipos en tiempo de compilación.

Para este proyecto se ha utilizado un sistema de tipos estático, apostando por la seguridad que ofrece esta arquitectura y evitando el gran tiempo que consume la creación de un sistema de tipos dinámico y seguro.

Alcance o *scope*

En el ámbito de los compiladores los alcances se refieren a la posibilidad de acceder a un dato o hacer uso de una función desde un bloque de código concreto. Por ejemplo, si una variable es declarada dentro del bloque de código de una función, esta misma variable no puede ser utilizada fuera de dicho bloque de código.

El control de los alcances es fundamental para que la lógica del lenguaje sea correcta, el correcto uso de los alcances debe ser registrado por una estructura de datos, normalmente una tabla de símbolos. La fase responsable de orquestar este sistema es la fase de análisis semántico, sin embargo, los datos de alcance son utilizados durante todas las fases posteriores al análisis semántico.

Shadowing u ocultación

La ocultación de identificador es una técnica de control de alcances que permite redefinir identificadores en diferentes *scopes*, utilizando los alcances como única medida de control de acceso para identificadores con el mismo nombre.

Generación de código intermedio

Tras validar el programa, el compilador transforma el AST en una representación intermedia (IR). Esta forma intermedia actúa como puente entre el lenguaje fuente y el código máquina, permitiendo aplicar optimizaciones y desacoplar las fases de análisis de la generación de código máquina en el compilador.

LLVM IR

En este proyecto se utiliza LLVM IR [2] como representación intermedia, en el capítulo 5 [4.4](#) se pueden apreciar ejemplos concretos de programas y su IR. El uso de este IR junto con otros módulos de LLVM facilita tanto la aplicación de optimizaciones automáticas como la generación de código para distintas arquitecturas.

Optimización

La fase de optimización tiene como objetivo mejorar el código generado sin alterar su comportamiento observable. Estas mejoras pueden centrarse en reducir el número de instrucciones, eliminar código innecesario o mejorar el uso de recursos.

Gracias a la infraestructura LLVM, este proyecto puede beneficiarse de un amplio conjunto de optimizaciones ya implementadas, como la propagación de constantes, la eliminación de código muerto o la simplificación de expresiones. En el capítulo 5 [4.4](#) se muestran ejemplos concretos de estas optimizaciones aplicadas al código generado.

Generación de código final y *runtime*

La última fase del proceso consiste en transformar la representación intermedia optimizada en código máquina ejecutable. Para ello, LLVM se encarga de generar código específico para la arquitectura destino.

El código generado se apoya en un *runtime*, encargado de proporcionar funcionalidades de soporte necesarias para la ejecución del programa, como la inicialización del entorno, la gestión de eventos o la interacción con bibliotecas externas.

3.3. Infraestructura LLVM y modelo de compilación

LLVM es una infraestructura de compilación moderna basada en un enfoque modular y por fases. A diferencia de compiladores monolíticos como GCC, donde las distintas etapas están fuertemente acopladas, LLVM permite separar la responsabilidad de las primeras fases de análisis de la representación y la generación de código ejecutable.

Este enfoque permite reutilizar componentes, aplicar optimizaciones independientes del lenguaje fuente y generar código ejecutable independiente de la arquitectura *host*. Estas características han sido determinantes para la elección de LLVM como un elemento central del desarrollo del compilador en este proyecto.

4. Técnicas y herramientas

Este capítulo se divide en dos bloques principales: en primer lugar, se describe la metodología seguida durante el desarrollo del proyecto, tanto a nivel organizativo como técnico; en segundo lugar, se presentan las herramientas y tecnologías empleadas.

4.1. Metodología de desarrollo

El desarrollo del proyecto se ha organizado siguiendo una metodología ágil inspirada en el marco de trabajo **SCRUM**, apoyado en la herramienta **ZenHub** para su gestión y seguimiento. La elección de una metodología ágil se justifica por su *flexibilidad* y *adaptabilidad*, características que son especialmente valoradas en un contexto tan experimental como lo es el diseño e implementación de un lenguaje de programación.

El flujo de trabajo se estructura mediante un tablero **Kanban** en ZenHub, el cual permite visualizar de manera clara el estado actual del proyecto. En dicho tablero se gestionan las *issues* y los *milestones* de GitHub, representando las distintas tareas, etapas o funcionalidades en desarrollo. Cada elemento del tablero incluye información relevante como la prioridad, la duración estimada, el tipo de tarea y su relación con otras, lo que facilita la planificación y el control del progreso de manera visual y dinámica.

Esta metodología fomenta un desarrollo incremental y una mejora continua del código, permitiendo incorporar nuevas funcionalidades, corregir errores y ajustar los objetivos a medida que avanza el trabajo, garantizando así un resultado más coherente y alineado con los objetivos técnicos del proyecto.

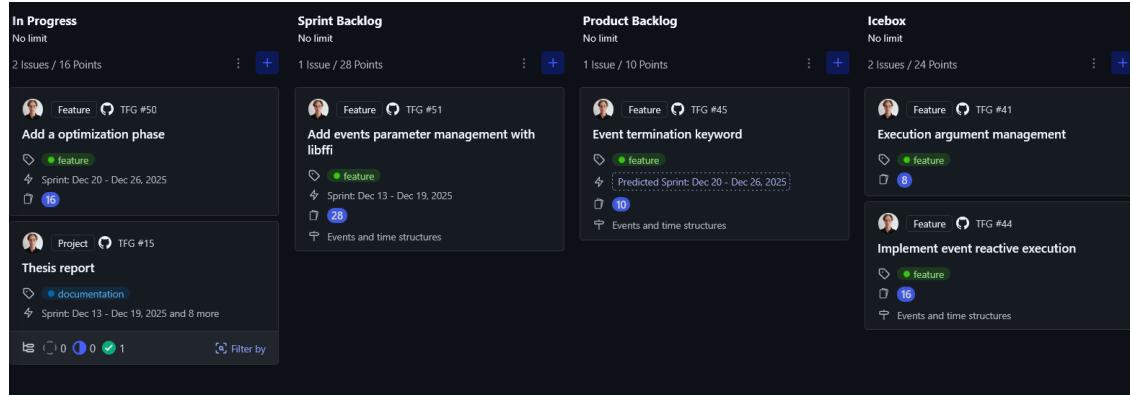


Figura 4.1: Imagen con la visualización del tablero Kanban en ZenHub

4.2. Metodología de programación

El desarrollo del compilador y de las herramientas asociadas se ha abordado desde el paradigma de la **programación orientada a objetos** (OOP, por sus siglas en inglés), reflejándose directamente en la estructura del código y en la organización de los distintos módulos del compilador. De forma más concreta, esta metodología permite la incorporación del patrón **visitor**, el cual puede encontrar detallado en el anexo de diseño.

La OOP favorece la reutilización, extensibilidad y mantenibilidad del software, permitiendo aislar los distintos módulos del compilador y facilitar su evolución futura. Además, este paradigma se integra de forma natural con las herramientas empleadas, como **ANTLR** [20] y **LLVM** [10], ambas diseñadas con arquitecturas orientadas a objetos.

Otro motivo determinante en la elección de este enfoque es el uso de **C++** [25] como lenguaje de implementación. C++ ofrece un control preciso sobre el uso de memoria y las abstracciones de bajo nivel, aspectos especialmente relevantes en la implementación de compiladores y runtimes personalizados. La compatibilidad de la metodología OOP entre C++ y las herramientas ANTLR y LLVM refuerza aún más la idoneidad de esta metodología para el proyecto.

La trazabilidad y el control de versiones del código se han gestionado mediante la herramienta **Git** y el repositorio remoto ya mencionado en **GitHub**, lo que permite registrar de forma precisa la evolución del proyecto y consultar en todo momento los cambios realizados desde su inicio.

Dado que el desarrollo del proyecto ha sido llevado a cabo de forma individual y siguiendo un flujo de trabajo mayoritariamente lineal, se ha optado por un modelo de trabajo simplificado basado en una única rama principal. Esta decisión permite mantener un historial de cambios claro y fácilmente interpretable, sin introducir la complejidad adicional asociada a la gestión de múltiples ramas, la cual resulta especialmente útil en entornos colaborativos pero menos relevante en un contexto de desarrollo individual y controlado como el presente.

4.3. Calidad del código

Para el análisis estático del código fuente escrito en C++, se ha utilizado la herramienta **Cppcheck**. Esta herramienta permite detectar errores comunes de programación, posibles accesos inválidos a memoria, usos indebidos de punteros, fugas de memoria, etc.

La elección de *Cppcheck* frente a otras soluciones más completas, como SonarQube, responde principalmente a criterios de accesibilidad y adecuación al contexto del proyecto. Aunque otras plataformas ofrecen análisis avanzados, en la práctica el soporte completo para C++ suele estar restringido a versiones comerciales de pago. Dado el carácter académico del proyecto, se ha considerado poco razonable asumir estos costes adicionales únicamente para el análisis estático del código, optando por una herramienta libre, ampliamente utilizada y suficientemente robusta para los objetivos del proyecto.

De forma complementaria, se ha seguido el estándar de estilos de C++ propuesto por el proyecto LLVM [14]. Este estándar proporciona un conjunto de convenciones claras y coherentes sobre nomenclatura, organización del código, formato y buenas prácticas. La adopción de este estilo resulta especialmente apropiada dado que el proyecto hace uso intensivo de la infraestructura LLVM, alineando así tanto el diseño interno como las herramientas empleadas con los estándares del propio ecosistema.

4.4. Elección de herramientas

A continuación se presentan las principales herramientas empleadas en el desarrollo del proyecto, así como la justificación de su elección frente a otras alternativas disponibles. Las herramientas seleccionadas se han escogido teniendo en cuenta su madurez, soporte, documentación, integración con C++ y adecuación a los objetivos técnicos del proyecto.

ANTLR

ANTLR (*Another Tool for Language Recognition*) es una herramienta ampliamente utilizada para la generación automática de analizadores léxicos y sintácticos a partir de gramáticas formales. Su principal fortaleza radica en la facilidad con la que permite definir y mantener gramáticas complejas mediante una sintaxis clara y expresiva, generando código eficiente y legible para distintos lenguajes de programación, incluido C++.

Entre sus características más destacables se encuentran:

- Soporte nativo para gramáticas LL(*) [21] que permiten manejar ambigüedades complejas.
- Generación automática de analizadores léxicos y sintácticos a partir de un único archivo de gramática.
- Aunque fue diseñado para usarse principalmente en el entorno Java, actualmente cuenta con compatibilidad para muchos lenguajes de destino, entre ellos C++.

Alternativas consideradas:

- **Bison** [4] y **Flex** [22]: Son las herramientas clásicas en la generación de analizadores. Sin embargo, en el contexto elegido ANTLR cuenta con una estructura orientada a objetos más integrable con otras herramientas.
- **PEG (Parsing Expression Grammar)** [5]: Ofrecen gran expresividad y flexibilidad, pero carecen del soporte multiplataforma y de la madurez de ANTLR, además de no integrarse de forma fácil con C++.
- **JavaCC** [9]: Alternativas orientadas a Java, no adecuadas para este proyecto al estar centrado en C++.

Justificación de la elección: ANTLR se considera la opción más equilibrada entre facilidad de uso, potencia y soporte técnico. Permite definir la gramática del lenguaje de manera declarativa y coherente, integrándose perfectamente con el entorno C++ mediante un runtime específico para este lenguaje. Además, ANTLR permite generar automáticamente componentes como los contextos, *visitors* y *listeners*, lo que facilita enormemente la construcción del AST.

LLVM

LLVM (*Low Level Virtual Machine*) [10, 15, 11] es una infraestructura modular y extensible para el desarrollo de compiladores, enlazadores y optimizadores de código. En este proyecto, LLVM se utiliza principalmente como una frontera intermedia estable entre las primeras fases de análisis y la generación de código máquina. El uso de LLVM IR permite validar la corrección del código generado, aplicar optimizaciones estándar y delegar la generación de código específico de arquitectura a la infraestructura de LLVM, reduciendo la complejidad de implementación y aportando soporte multiplataforma.

Entre sus principales ventajas destacan:

- **Representación intermedia (IR):** Permite una separación clara entre la fase de análisis y la de generación de código, haciendo posible optimizaciones y traducciones a múltiples arquitecturas.
- **Portabilidad:** LLVM IR puede compilarse a diferentes arquitecturas (x86, ARM, RISC-V, etc.) sin modificar el compilador fuente.
- **Integración con C++:** Al estar implementado en C++, se integra de manera natural con el entorno de desarrollo del proyecto.
- **Extensibilidad:** Su diseño modular permite hacer cambios como optimizaciones, cambio de backend o cambios en herramientas de análisis de forma sencilla.

Alternativas consideradas:

- **GCC (*GNU Compiler Collection*)** [6]: Ofrece una infraestructura madura, pero su API interna no está diseñada para un uso externo, lo que dificulta su integración en proyectos ajenos al entorno GNU.
- **QBE Compiler Infrastructure** [19]: Es más ligera que LLVM, pero carece del ecosistema, documentación y soporte de comunidad que caracterizan a LLVM.
- **Cranelift** [3]: Un *backend* moderno y rápido, pero enfocado en compilación JIT (*just-in-time*) y no tan versátil para la generación de IR o compiladores tradicionales.

Justificación de la elección: LLVM ofrece el equilibrio ideal entre potencial, documentación y flexibilidad, su diseño modular permite construir un compilador completamente funcional sin preocuparse por la dependencia de la arquitectura de destino. Además, su integración con C++ simplifica enormemente la generación de IR y la gestión de optimizaciones.

Entorno WSL

Justificación de la elección: El entorno Linux (conseguido mediante WSL) es especialmente sencillo para trabajar con las dependencias de este proyecto sin requerir capas de compatibilización adicionales requeridas por Windows (como MSYS2 [18] o MinGW [17]).

Sus principales ventajas frente a Windows:

- **Simplicidad:** Con un entorno únicamente de CLI permite un gran rendimiento y mantiene un entorno coherente con el desarrollo de un compilador, evitando elementos innecesarios que puedan interferir con el desarrollo del proyecto.
- **Shell scripting:** Permite crear rutinas de ejecución sencillas que ahorran repetir trabajo a la hora de compilar y probar código.
- **Integración:** Todos los paquetes están preparados para su uso en Unix-like, no se requiere ningún parche para compatibilizar las herramientas entre si. Gran parte del software viene preinstalado en muchas distribuciones de Linux.

Visual Studio Code

Justificación de la elección: La elección de Visual Studio Code como IDE se da por su simpleza y conocimiento previo, evitando tener que aprender sobre cómo usar un IDE más específico como Visual Studio, pero a su vez potente gracias a su sistema de extensiones, que permite modificarlo según las necesidades del desarrollo.

En mi caso lo he personalizado con las siguientes extensiones:

- **WSL:** Permite la integración con WSL como si se tratase del sistema host.
- **CMake y CMake Tools:** Permite ejecutar las rutinas de CMake creadas en el proyecto.

- **L^AT_EX Workshop:** Integración con L^AT_EX y comodidad de compilación al guardar los archivos para ver los cambios inmediatamente.
- **C/C++ Extension pack:** Agrega IntelliSense, linting, herramientas de debugging y code browsing para C y C++.
- **Clang-Format:** Herramienta configurable que permite establecer reglas de estilo consistentes en todos los archivos de código, en mi caso utilizo el estándar de LLVM.

Despliegue con Docker

Justificación de la elección: Con el objetivo de facilitar el acceso y la prueba del compilador desarrollado, se ha optado por proporcionar un entorno de ejecución reproducible mediante el uso de Docker. Esta decisión permite desacoplar el proceso de evaluación del proyecto de la configuración concreta del sistema anfitrión, evitando problemas derivados de dependencias, versiones de las herramientas o problemas derivados del sistemas operativo anfitrión.

Docker se utiliza en este proyecto como un **entorno de despliegue y ejecución**, no como entorno de desarrollo. El contenedor incluye únicamente lo necesario para la ejecución del compilador (binario del compilador y objetos del *runtime*), junto con las dependencias dinámicas requeridas, evitando la inclusión de herramientas de desarrollo o ficheros auxiliares generados durante la compilación.

Las principales ventajas de Docker son:

- **Reproducibilidad:** Cualquier usuario puede ejecutar el compilador en un entorno idéntico al utilizado durante el desarrollo, independientemente del sistema anfitrión.
- **Aislamiento de dependencias:** Se evita la necesidad de instalar manualmente bibliotecas complejas como LLVM o ANTLR.
- **Portabilidad:** El contenedor puede ejecutarse en sistemas anfitrión Linux, Windows o macOS mediante Docker, manteniendo un comportamiento consistente.
- **Simplicidad de evaluación:** El evaluador puede acceder a un entorno interactivo desde el cual compilar y ejecutar programas escritos en el lenguaje desarrollado sin configurar nada.

Alternativas consideradas:

- **Ejecución en el sistema anfitrión:** Esta opción requeriría la instalación manual de dependencias complejas como LLVM, ANTLR y el resto de bibliotecas de tiempo de ejecución, además de garantizar compatibilidad de versiones entre todas estas herramientas. Esta propuesta lleva con facilidad a problemas de reproducibilidad.
- **Distribución mediante máquina virtual:** El uso de una máquina virtual permitiría encapsular completamente el entorno de ejecución, pero implica un mayor consumo de recursos y menor comodidad. Por otro lado, Docker ofrece un aislamiento equivalente con un coste significativamente menor.

5. Aspectos relevantes del proyecto

El comienzo del proyecto parte del objetivo de conseguir una versión MVP (*Minimal Viable Product*) que pudiese aportar una base sólida y fácil de ampliar. En este caso el MVP se enfocó en la creación de un micro compilador capaz de procesar expresiones aritméticas y lógicas, puesto que esta estructura es básica que está presente en todos los lenguajes de programación.

En fases posteriores del proyecto se trabaja tomando como base este MVP para añadir características adicionales al lenguaje, como las estructuras de control condicionales, bucles, funciones y eventos.

Es importante como paso previo a la creación de un lenguaje, la elección de un nombre para el mismo. Puesto que el objetivo inicial del proyecto implicaba experimentar con estructuras lógicas dependientes del tiempo y teniendo en cuenta la presencia de un tipo temporal en el lenguaje, se ha elegido el nombre «T» para el lenguaje y «Tcompiler» para el compilador, la sencillez de este nombre viene inspirada por el lenguaje C.

Antes de compilar

El compilador antes de comenzar el proceso de compilación debe recibir en su ejecución datos que hagan referencia al archivo que se debe compilar, que archivo de salida se quiere generar y de que forma se debe realizar el proceso, permitiendo así al programador ajustar este proceso a sus necesidades.

Dentro del compilador se esperan algunos argumentos de llamada como:

- **<nombre de la entrada>**: argumento obligatorio, indica el nombre del fichero .T de entrada.
- **-o <nombre archivo>**: argumento obligatorio que especificar el fichero ejecutable de salida.
- **--debug**: permite ver información de la salida de cada una de las fases del compilador. Este argumento también activa el efecto del argumento **--visualizeAST**;
- **--visualizeAST**: genera un fichero AST.pfd con una visualización del AST en el directorio donde se ha ejecutado el compilador.
- **--IR <nombre archivo>**: genera un fichero .ll con el LLVM IR generado por el compilador.
- **--basic**: excluye la fase de optimización del proceso de compilación.
- **--help / -h**: muestra la ayuda del compilador, indicando como debe ser utilizado y sus diferentes argumentos.
- **--version / -v**: muestra la versión actual del compilador.

Para conocer los datos necesarios para ejecutar los comandos anteriores en el compilador se almacenan flags de compilado que permiten modificar el comportamiento de cada fase.

Proceso de compilación

Como el compilador se ha planificado de forma modular, a continuación se recorrerá el proceso de compilación completo, utilizando del contenido del MVP como ejemplo.

Análisis léxico

El analizador léxico lo generaremos mediante la herramienta ANTLR4, la cual nos permite definir los lexemas del lenguaje de forma independiente de la producciones gramaticales donde se van a utilizar. En este proyecto, el archivo que contiene esta información se encuentra en `src/grammar/Tlexer.g4`.

En este momento se definen las palabras clave asociadas a los tipos básicos del lenguaje:

```

TYPE_INT      : 'int'      ;
TYPE_FLOAT    : 'float'    ;
TYPE_CHAR     : 'char'     ;
TYPE_STRING   : 'string'   ;
TYPE_BOOLEAN  : 'bool'    ;
TYPE_VOID     : 'void'    ;
TYPE_PTR      : 'ptr'     ;

```

Se definen los operadores aritméticos y lógicos:

```

PLUS   : '+'   ;
MINUS  : '-'   ;
MUL   : '*'   ;
DIV   : '/'   ;
MOD   : '%'   ;

INC   : '++'  ;
DEC   : '--'  ;

EQ    : '=='  ;
NE    : '!=';
LT    : '<'  ;
LE    : '<=' ;
GT    : '>'  ;
GE    : '>=' ;

```

Análisis sintáctico

Una vez obtenido el analizador léxico el siguiente paso es construir un analizador sintáctico capaz de reconocer producciones gramaticales, al igual que en la fase anterior, se realiza con ANTLR4 y se define el archivo `src/grammar/Tparser.g4`.

Los operadores y las operaciones se definen en el lenguaje ANTLR4 de la siguiente forma:

```

expr
: expr op=(MUL|DIV) expr      # arithmeticExpr
| expr op=(PLUS|MINUS) expr    # arithmeticExpr
| expr comparisonOperator expr # logicalExpr
| operand                      # operandExpr

```

```

| LPAREN expr RPAREN           # parenExpr
;

comparisonOperator
: EQ
| NE
| LT
| LE
| GT
| GE
;

operand
: literal
;

```

Cabe destacar el uso de etiquetas (como `# parenExpr`) que facilitan posteriormente acceder a cada tipo de expresión gracias a la separación en contextos que realiza ANTLR4. En este ejemplo en concreto, la etiqueta `# arithmeticExpr` sirve para separar la precedencia, permitiendo empaquetar en el mismo contexto ambas producciones, facilitando el análisis en fases posteriores.

Generación del AST

Para formar el AST se empleará un *visitor pattern* (patrón de visita), objeto que recibe el nombre de `ASTBuilder`, el cual se encarga de visitar cada contexto generado de forma automática por el parser de ANTLR, dando por resultado una estructura de nodos enlazados, el AST.

Adicionalmente, si la flag `--visualizeAST` está activada, el compilador puede generar una representación visual del árbol tras su construcción, para conseguir esto se escriben cada nodo en el formato de la biblioteca ‘forest’ de `LATEX`, permitiendo el análisis de la estructura del programa así como datos relacionados con los componentes, como los tipos u operadores.

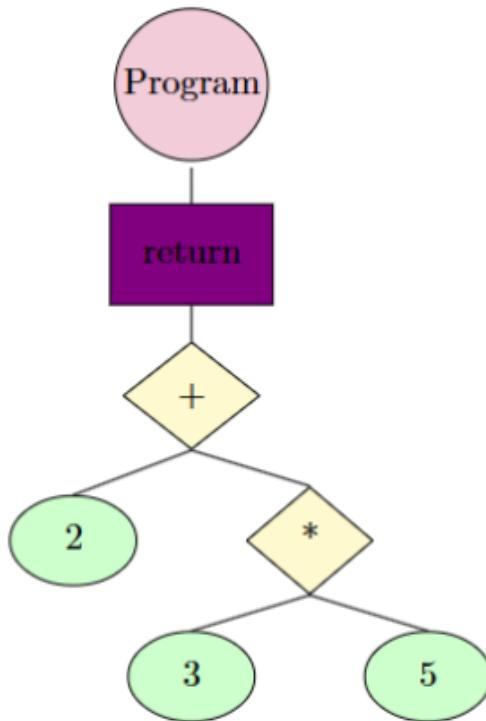


Figura 5.1: Ejemplo de visualización de la expresión $2 + 3 * 5$ con la flag `--visualizeAST`

Análisis semántico

Durante la fase de análisis semántico se examina principalmente el uso de los identificadores, como los nombres de variables y funciones. Entre las principales tareas de esta fase se encuentran:

- La creación de un *scope* para cada bloque de código.
- La inserción de los identificadores en su *scope* correspondiente.
- La verificación del alcance en cada acceso a un identificador.
- Las comprobaciones de tipos en asignaciones, operaciones y sentencias de retorno.

La tabla de símbolos está estructurada como una lista de *scopes*. Para los distintos bloques anidados se establecen referencias entre cada *scope* hijo

y su correspondiente *scope* padre, lo que permite verificar fácilmente los alcances y resolver nombres de forma correcta.

Scope #0 (level 0)		
Key	Category	Type
foo	FUNCTION	int
toString	FUNCTION	string
strlen	FUNCTION	int
print	FUNCTION	void

Scope #1 (level 1)		
Key	Category	Type
x	PARAMETER	int

Scope #2 (level 2)		
Key	Category	Type
b	VARIABLE	int

Figura 5.2: Ejemplo de tabla de símbolos generada por el analizador semántico

Para poder comprobar la corrección del uso de tipos, el análisis semántico debe ser capaz de entender los tipos, tanto de los literales como de las variables y funciones, para ello se implementa un sistema de representación de tipos en una enumeración llamada **SupportedTypes** que da soporte a los tipos:

- TYPE_INT
- TYPE_FLOAT
- TYPE_CHAR

- TYPE_STRING
- TYPE_BOOLEAN
- TYPE_VOID
- TYPE_PTR

Librería estándar

Para permitir mayor flexibilidad al programador, todas las funciones de la librería estándar de C son enlazadas con el programa final generado. Adicionalmente, se agregan o adaptan algunas funciones para crear una pequeña librería propia, que demuestra la capacidad del compilador de incluir funciones preinstaladas o *built-in functions*.

- `char *toString(int x)`
- `void print(const char *first, ...)`

La función antes mencionada,

```
char *toString(int x)
```

, aprovecha internamente funciones estándar de C como

```
int sprintf(char *str, const char *format, ...)
```

para devolver un *buffer* de tipo

```
char *
```

sin tener que reinventar este mecanismo y a su vez abstrayendo al programador del uso de estas funciones y el manejo de memoria.

Generación de LLVM IR

Para la generación de IR es emplea un *struct* que almacena los datos necesarios para que LLVM pueda generar el IR de forma automática, para ello se inicializan dentro del `CodegenContext` los siguientes tres elementos:

- Un módulo de LLVM.
- Un contexto de LLVM.
- Un builder de LLVM.

Durante el proceso de generación de código IR, se vuelve a recorrer el AST con un *visitor pattern*, el cual emplea principalmente el *builder* para generar las instrucciones a partir de los datos contenidos en los nodos del AST.

Algunas de las consideraciones necesarias para compatibilizar el AST y el generación de IR serían los tipos que se manejan en cada uno, para poder hacer una conversión correcta se implementa una función capaz de transformar:

Tipo AST	Tipo LLVM
TYPE_INT	i32
TYPE_FLOAT	float
TYPE_CHAR	i8
TYPE_STRING	i8*
TYPE_BOOLEAN	i1
TYPE_VOID	void
TYPE_PTR	ptrType

Tabla 5.1: Correspondencia entre tipos del AST y tipos de LLVM IR

Una vez completado el proceso obtendremos un programa equivalente en formato LLVM IR, el cual para este ejemplo concreto es:

```
; ModuleID = 'program'
source_filename = "program"

declare ptr @toString(i32)
```

```
declare i32 @printf(ptr, ...)

declare i32 @strlen(ptr)

declare void @registerEventData(ptr, float, ptr, i32, ptr, i32)

declare void @scheduleEvent(ptr, ptr)

define i32 @mainLLVM() {
entry:
    ret i32 17
}
```

En este caso vemos una optimización automática de LLVM que se explicará en el siguiente apartado.

Optimizaciones

Para asegurar la calidad del código final, se aplican sobre el LLVM IR algunas optimizaciones incluidas en la API[?] de LLVM, las cuales aplican transformaciones de código o eliminaciones mediante pasadas que recorren todo el IR generado.

Algunas de las optimizaciones más importantes son:

- **Propagación y plegado de constantes (*Constant propagation & Constant folding*):** Evalúa expresiones cuyos operandos son constantes en tiempo de compilación, sustituyéndolas por su valor resultante y eliminando cálculos innecesarios en tiempo de ejecución.
- **Eliminación de código muerto (*Dead code elimination*):** Suprime instrucciones, variables y bloques de código que no producen efectos observables sobre el programa, reduciendo el tamaño del código generado y mejorando su eficiencia.
- **Simplificación del flujo de control (*Control flow graph simplification*):** Simplifica la estructura del grafo de control eliminando ramas inalcanzables, saltos redundantes y bloques básicos innecesarios.
- **Optimización de bucles (*Loop optimizations*):** Incluye transformaciones como la extracción de código invariante fuera de los bucles y la simplificación de bucles con límites conocidos para reducir el coste de iteraciones repetidas.

- **Optimización interprocedural (IPO):** Analiza las relaciones entre funciones para aplicar optimizaciones como la eliminación de funciones no utilizadas y la propagación de información entre llamadas.
- **Inlining de funciones:** Sustituye llamadas a funciones pequeñas por el cuerpo de la función llamada cuando resulta beneficioso, reduciendo el coste de llamada y habilitando nuevas optimizaciones posteriores.
- **Optimización de accesos a memoria:** Reduce cargas y almacenamientos redundantes mediante análisis de alias y de flujo de datos, mejorando el uso de registros y la localidad de memoria.
- **Optimización de expresiones comunes (*Common subexpression elimination*):** Detecta cálculos idénticos que se repiten dentro de una función o bloque y los reutiliza, evitando evaluaciones duplicadas.

En el ejemplo anterior, podemos observar un ejemplo de *constant folding*, donde la expresión original `return 2+3*5;` puede ser traducida a `return 2+15;` y finalmente `return 17;`, obteniendo de esta manera un único *return statement* exento de cálculos.

El uso de estas combinaciones junto a las comprobaciones de calidad mínimas para compilar sin errores, generan un código de una calidad similar al código optimizado con `-O2` de compiladores como GCC, Rustc o Swiftc.

Este ejemplo ilustra cómo las optimizaciones permiten simplificar significativamente el código generado, reduciendo tanto el número de instrucciones como la complejidad del flujo de control, sin alterar el comportamiento del programa.

Código fuente original:

```
if (2 < 3) {
    return 1;
}

return 2;
```

LLVM IR generado sin optimizaciones:

En una primera fase, el generador de IR traduce directamente la estructura del programa fuente, manteniendo el flujo de control explícito mediante saltos condicionales.

```
define i32 @mainLLVM() {
entry:
  br i1 true, label %then, label %endif
then:
  ret i32 1
endif:
  ret i32 2
}
```

Puede observarse que la condición $2 < 3$ se evalúa en tiempo de compilación mediante una optimización de **constant folding**, sustituyendo la condición por un valor constante (**true**).

LLVM IR tras la fase de optimización:

Durante la fase de optimización, el compilador analiza el flujo de control del programa y detecta que todas las rutas de ejecución alcanzables conducen al mismo valor de retorno. Como consecuencia, el bloque condicional completo puede eliminarse, reduciendo la función a una única instrucción de retorno.

```
define noundef i32 @mainLLVM() local_unnamed_addr #0 {
entry:
  ret i32 1
}
```

Generación del ejecutable

La generación de un programa requiere conocer el «*target triplet*», es decir, la familia de CPU en la que se ejecuta el sistema. La API del sistema `llvm::sys` [16] de LLVM contiene funciones que nos permiten conocer el *triplet* y así poder establecer un data layout correcto para el sistema en el que estamos trabajando.

Una vez hemos generado el código objeto del programa, el compilador ejecuta un enlace utilizando la herramienta clang++, de forma que el archivo generado se enlaza con la librería estándar del compilador, la librería estándar de C y el *runtime*.

Runtime

El runtime es una parte necesaria para la ejecución de cualquier programa, normalmente aporta un punto de entrada valido mediante un `system call` en ensamblador.

La sencilla rutina de inicialización en ensamblador:

```
_start:
    call main

    mov %rax, %rdi
    mov $60, %rax

    syscall
```

Además aporta un entorno de ejecución correcto y personalizado para poder llevar a cabo acciones concretas, en el caso de este lenguaje, es el *runtime* el responsable de la gestión de los eventos, para su planificación y ejecución.

Gestión de eventos

Dado que los eventos representan unidades de ejecución independientes que pueden activarse de forma diferida o periódica, su correcta gestión resulta clave para garantizar la estabilidad, el control y la robustez del sistema.

Cada evento se ejecuta en un hilo independiente, separado del hilo principal del programa (la función *main*). Esta decisión de diseño permite aislar la ejecución de los eventos del flujo principal, de modo que un fallo o comportamiento inesperado en un evento no compromete la ejecución global del programa. De esta forma, el sistema mejora su tolerancia a errores y evita que una excepción o bloqueo en un evento provoque la finalización del proceso principal.

La creación, ejecución y finalización de los eventos se encuentra centralizada en el *runtime*, que actúa como gestor de eventos. Aunque los eventos se ejecutan en hilos independientes, el hilo principal mantiene en todo momento el control sobre su estado, pudiendo conocer qué eventos se encuentran activos, cuáles han finalizado y cuáles están pendientes de ejecución. Este enfoque permite combinar concurrencia con supervisión centralizada, evitando la pérdida de control sobre los hilos lanzados dinámicamente.

Los eventos son *autogestionados* una vez lanzados, se encargan de ejecutar su lógica asociada, control de sus limitaciones de ejecución y periodicidad. Esto reduce el acoplamiento entre el código del usuario y la infraestructura interna del runtime, simplificando tanto el modelo de programación como el mantenimiento del sistema.

Este modelo de ejecución concurrente controlada permite implementar mecanismos temporales y reactivos de forma segura, manteniendo un equilibrio entre flexibilidad, aislamiento y control. La gestión explícita de los eventos como entidades independientes refuerza la arquitectura del sistema y facilita futuras extensiones relacionadas con planificación, cancelación o monitorización avanzada de eventos.

Paso de parámetros

Uno de los aspectos más complejos del desarrollo del *runtime* ha sido la implementación del paso de parámetros hacia las funciones asociadas a los eventos. Dado que los eventos se ejecutan dinámicamente y en hilos independientes, resulta necesario disponer de un mecanismo que permita invocar funciones con firmas conocidas únicamente en tiempo de ejecución.

Para este propósito se exploró el uso de *libffi*, una biblioteca que permite realizar llamadas a funciones de forma dinámica, especificando en tiempo de ejecución la firma correcta conforme al *application binary interface* (ABI), tanto en la función a invocar como los tipos de sus parámetros. Aunque esta aproximación resulta adecuada para el paso de parámetros por valor, introduce una complejidad significativa cuando se trata de pasar punteros.

El principal problema radica en la gestión de memoria asociada a los parámetros por referencia. El uso de punteros implica garantizar que los datos apuntados permanezcan válidos durante toda la ejecución del evento, lo que requiere una gestión explícita de memoria en el *heap*, así como mecanismos claros de propiedad, ciclo de vida y liberación de recursos. Este tipo de gestión introduce riesgos adicionales, como accesos a memoria inválida, fugas de memoria o condiciones de carrera, especialmente en un entorno concurrente basado en múltiples hilos.

Dado que el objetivo principal del proyecto es el diseño e implementación del compilador y su sistema de ejecución, y no la construcción de un gestor de memoria completo, se decidió acotar el alcance de esta funcionalidad. En la implementación actual, el sistema de paso de parámetros se limita a valores por copia, evitando el uso de punteros y referencias complejas en las llamadas dinámicas realizadas mediante *libffi*.

Esta limitación se considera aceptable dentro del contexto del proyecto y se deja abierta y documentada en la sección 7 [5.1](#) como una línea de trabajo futura.

5.1. Evaluación del rendimiento

Con el objetivo de evaluar el rendimiento del compilador desarrollado, para esta tarea se han creado entre tres implementaciones equivalentes de un mismo algoritmo: una escrita en el lenguaje diseñado en este proyecto, su equivalente en lenguaje C compilado con optimizaciones `-O2`, y una versión implementada en `python3`.

El algoritmo seleccionado consiste en el cálculo de números primos hasta un valor n , utilizando exclusivamente estructuras de control básicas (condicionales y bucles) y operaciones aritméticas. No se emplean estructuras de datos complejas ni bibliotecas externas, con el fin de centrar la comparación en el coste del flujo de control y del modelo de ejecución de cada lenguaje.

Las mediciones se han realizado utilizando la herramienta `hyperfine`, una sencilla herramienta diseñada para medir el rendimiento de programas de consola. Para las pruebas se ha ejecutando cada programa de forma independiente y variando progresivamente el valor de n .

n	Lenguaje T (s)	C -O2 (s)	Python 3 (s)
50 000	0.059	0.046	1.388
100 000	0.156	0.126	4.263
150 000	0.279	0.218	8.332
300 000	0.774	0.636	24.553
500 000	1.579	1.509	55.395
1 000 000	4.464	4.168	152.328

Tabla 5.2: Tiempos de ejecución en el cálculo de números primos

En la siguiente gráfica podemos observar una comparativa entre las tres implementaciones:

Los resultados obtenidos muestran que el código generado por el compilador desarrollado presenta un rendimiento muy cercano al del código C optimizado, con diferencias aproximadas de TODO. En contraste, tanto el programa en C como el generado por el compilador superan ampliamente a Python, siendo los ejecutables compilados alrededor de un TODO más rápidos que el interpretado.

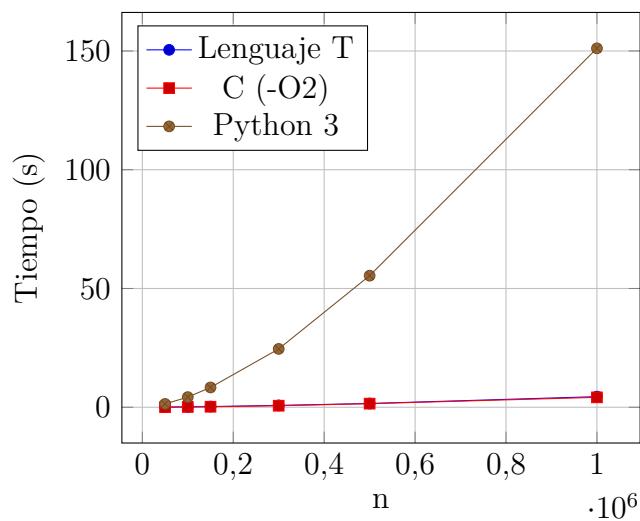


Figura 5.3: Gráfica comparativa del rendimiento en el cálculo de números primos

Estos resultados se explican por el hecho de que tanto el lenguaje desarrollado como C generan código nativo que se ejecuta directamente sobre la arquitectura subyacente, sin la sobrecarga propia de los lenguajes interpretados. Asimismo, el uso de una representación intermedia eficiente y la delegación de las optimizaciones de bajo nivel en la infraestructura LLVM permiten obtener un rendimiento competitivo pese a tratarse de un compilador desarrollado en el contexto de un Trabajo de Fin de Grado.

6. Trabajos relacionados

En el aspecto más teórico y académico destaca el «Libro del Dragón» [1] de Aho, Sethi & Ullman, el cual es un referente en cuanto a la teoría de lenguajes, así como de todos los componentes y técnicas que podemos encontrar en lexers, parsers, construcciones de AST, analizadores semánticos, generación de IR y optimizaciones. Aunque es un material muy completo, su extensión me ha llevado a consultar apartados concretos y resúmenes enfocados en ciertos aspectos relevantes para mi proyecto.

Destaca como proyecto de alcance y complejidad similar el tutorial/lenguaje didáctico llamado Kaleidoscope [13] el cual es parte del tutorial oficial de LLVM sobre compiladores. El proyecto Kaleidoscope permite comprender de forma práctica y ordenada los conceptos de *lexing*, *parsing*, generación de AST y, especialmente, la generación de código intermedio mediante LLVM, sirviendo como una referencia clara para la construcción de compiladores modernos.

El compilador de Rust [24] pese a su extensión y complejidad está excelentemente documentado, lo cual permite comprender sin demasiado esfuerzo como está estructurado y su funcionamiento interno. Este recurso me ha permitido contemplar como funciona un verdadero compilador del más alto nivel.

7. Conclusiones y líneas de trabajo futuras

Conclusiones

Tras realizar el proyecto, he conseguido una comprensión profunda sobre como funcionan los lenguajes de programación así como los procesos de compilación y en menor medida el proceso de interpretación. Este trabajo me ha hecho lanzar una mirada crítica a herramientas y lenguajes que llevo usando desde hace muchos años pero nunca me había preguntado realmente como operaban fuera de la mirada del programador.

Estudiar, analizar y comprender el trabajo tan extenso y complejo que supone crear un compilador de la calidad de GCC o rustc, también me ha hecho que me sienta especialmente agradecido de disponer de acceso a tecnologías tan punteras y sofisticadas de forma totalmente gratuita, estas herramientas nos permiten desarrollar de forma cómoda nuestra labor en el desarrollo de software sin pedir nada a cambio.

Líneas de trabajo futuras

Puesto que el proyecto está extremadamente acotado por su complejidad y el tiempo disponible, considero que hay numerosas líneas de trabajo con las que me gustaría continuar próximamente, algunas de las más destacables:

- Migrar de ANTLR a un *lexer* y *parsers* propios. Pese a que ANTLR4 genera analizadores de una calidad comparable a cualquier compilador profesional, la mayoría de grandes compiladores actuales proporcionan sus propias herramientas de análisis, puesto que son mucho más

optimizables y personalizables. Además, evitan la inclusión de clases, contextos y símbolos innecesarios que aumentan en algunos casos innecesariamente la complejidad de los compiladores finales.

- Ampliar el lenguaje para soportar la metodología OOP, una de las metodologías de la programación más populares y de las que mejor encajan con un lenguaje que pretende modelar estructuras tan complejas como eventos.
- Un *runtime* más sofisticado, con mayor precisión en el control del tiempo y rutinas más optimizadas hechas directamente en ensamblador.
- Implementar captura de variables para mejorar las funciones, esta característica que implementan la mayoría de compiladores modernos, permiten el uso de variables de alcances superiores dentro de la propia función sin pasar explícitamente los valores a la función. Por ejemplo, el siguiente pseudo código sería incorrecto si los compiladores modernos no implementaran captura de variables:

```
function makeCounter():
    a = 0

function counter():
    a = a + 1 // Ocurre captura de la variable a
    return a

return counter
```

En el ejemplo anterior, a no existen el alcance de **counter()**, tampoco se pasa como un parámetro, sino que a accesible porque es capturada. Para ello el compilador tomaría toda la información del entorno donde se define la función **counter()**.

```
struct Env_makeCounter {
    a
}
```

Y posteriormente sustituye la llamada por la misma función reescrita con el entorno siendo accesible mediante paso por parámetro:

```
function counter(env):
    env.a = env.a + 1 // Se reescribe el uso de la variable a
    return env.a
```

- Más pruebas y tiempo de depuración para poder detectar errores complejos en casos de uso muy específicos, para poder analizarlos y subsanarlos.
- Inclusión de bibliotecas/módulos, que permitan a los usuarios escribir sus propios códigos en archivos que posteriormente puedan importar y utilizar, permitiendo una mejor organización del código fuente, así como la reutilización del código fuente y la colaboración entre proyectos.
- Sistema de gestión de memoria (*garbage collector*) capaz de hacer reservas y liberaciones de memoria en *heap*, permitiendo un control más amplio de punteros/referencias, especialmente en el contexto de los programas en ejecución.

Bibliografía

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2nd edition, 2007.
- [2] Ayman Alheraki. *LLVM IR Quick Reference*. simplifycpp.org, 2025. 455 pages.
- [3] Bytecode Alliance. Cranelift code generator. <https://cranelift.dev/>, 2025. Accedido: diciembre 2025.
- [4] Charles Donnelly. Bison the yacc-compatible parser generator. *Technical report, Free Software Foundation*, 1988.
- [5] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, 2004.
- [6] GCC Development Team. GNU compiler collection (GCC). <https://gcc.gnu.org/>, 2025. Accedido: diciembre 2025.
- [7] ECMA International. EcmaScript standardization. <https://ecma-international.org/publications-and-standards/standards/ecma-262/>, 1997. [Internet; consultado diciembre-2015].
- [8] ISO/IEC. Information technology—database languages—sql. Technical Report ISO/IEC 9075:2016, International Organization for Standardization, 2016.
- [9] Java Community. JavaCC: Java compiler compiler. <https://javacc.github.io/javacc/>, 2025. Accedido: diciembre 2025.

- [10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [11] Chris Lattner and Vikram Adve. The LLVM compiler framework and infrastructure tutorial. *Languages and Compilers for High Performance Computing*, pages 15–16, 2008.
- [12] Håkon Wium Lie and Bert Bos. Cascading style sheets, level 1. Technical report, W3C Recommendation, 1996.
- [13] LLVM Project. Kaleidoscope introduction. <https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>, 2025. Accedido: diciembre 2025.
- [14] LLVM Project. Llvm coding standards. <https://llvm.org/docs/CodingStandards.html>, 2025. Accedido: Diciembre de 2025.
- [15] LLVM Project. The LLVM compiler infrastructure. <https://llvm.org>, 2025. Accedido: Diciembre de 2025.
- [16] LLVM Project. *LLVM’s Target Independant Code Generation*, 2025. Accedido: Diciembre de 2025.
- [17] MinGW Project. MinGW: Minimalist GNU for windows. <https://nuwen.net/mingw.html>, 2025. Accedido: diciembre 2025.
- [18] MSYS2 Contributors. MSYS2: Software distribution and building platform for windows. <https://www.msys2.org/>, 2025. Accedido: diciembre 2025.
- [19] Quentin Mével. QBE: The quick backend. <https://c9x.me/compile/>, 2025. Accedido: diciembre 2025.
- [20] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013.
- [21] Terence Parr and Kathleen Fisher. LL(*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices*, 46(6):425–436, 2011.
- [22] Vern Paxson, Will Estes, and John Millaway. *Flex: The Fast Lexical Analyzer Generator*. Free Software Foundation, 1995. Version 2.5.

- [23] Guido van Rossum Python Software Foundation. Foreword for "programming python". <https://www.python.org/doc/essays/foreword/>, 2025. [Internet; consultado diciembre-2015].
- [24] Rustc Lang Org. Rustc compiler development guide. <https://rustc-dev-guide.rust-lang.org/overview.html>, 2025. Accedido: diciembre 2025.
- [25] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Upper Saddle River, NJ, 4th edition, 2013.
- [26] The MathWorks Inc. Matlab. <https://www.mathworks.com/products/matlab.html>, 2025. [Internet; consultado diciembre-2015].
- [27] WHATWG. HTML living standard. <https://html.spec.whatwg.org/>, 2025. [Internet; consultado diciembre-2015].
- [28] Wikipedia. Sql — wikipedia. <https://en.wikipedia.org/wiki/SQL>, 1970s. Accedido: diciembre 2025.
- [29] Wikipedia. Javascript — wikipedia, la enciclopedia libre, 2005. JavaScrip fue creado en 10 días para Netscape.