



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  
Documentación Técnica**



Presentado por Adrián Zamora Sánchez  
en Universidad de Burgos — 20 de diciembre  
de 2025

Tutor: César Ignacio García Osorio



---

# Índice general

---

<b>Índice general</b>	i
<b>Índice de figuras</b>	iii
<b>Índice de tablas</b>	iv
<b>Apéndice A Plan de Proyecto Software</b>	1
A.1. Introducción . . . . .	1
A.2. Planificación temporal . . . . .	1
A.3. Estudio de viabilidad . . . . .	3
<b>Apéndice B Especificación de Requisitos</b>	7
B.1. Introducción . . . . .	7
B.2. Requisitos funcionales . . . . .	7
B.3. Casos de uso . . . . .	9
<b>Apéndice C Especificación de diseño</b>	17
C.1. Introducción . . . . .	17
C.2. Diseño de datos . . . . .	17
C.3. Diseño arquitectónico . . . . .	17
C.4. Diseño procedimental . . . . .	18
<b>Apéndice D Documentación técnica de programación</b>	19
D.1. Introducción . . . . .	19
D.2. Estructura de directorios . . . . .	19
D.3. Manual del programador . . . . .	20
D.4. Compilación, instalación y ejecución del proyecto . . . . .	21

D.5. Pruebas del sistema . . . . .	21
<b>Apéndice E Documentación de usuario</b>	<b>23</b>
E.1. Introducción . . . . .	23
E.2. Requisitos de usuarios . . . . .	23
E.3. Instalación . . . . .	24
E.4. Manual del usuario . . . . .	24
<b>Apéndice F Anexo de sostenibilización curricular</b>	<b>25</b>
F.1. Introducción . . . . .	25
F.2. Principios de sostenibilidad del proyecto . . . . .	25
F.3. Competencias transversales en sostenibilidad . . . . .	26
F.4. Impacto social y ambiental del proyecto . . . . .	27
F.5. Conclusión . . . . .	27
<b>Bibliografía</b>	<b>29</b>

---

# Índice de figuras

---

C.1. Diagrama fases del compilador . . . . .	18
D.1. Diagrama de directorios . . . . .	19

---

# Índice de tablas

---

A.1. Costes totales . . . . .	4
A.2. Licencias . . . . .	4
B.1. CU-1, Traducción a código ejecutable . . . . .	10
B.2. CU-2, Visualización del estado de compilación . . . . .	11
B.3. CU-3, Traducción a código intermedio (IR) . . . . .	12
B.4. CU-4 Comprobación de la validez léxica de un código . . . . .	13
B.5. CU-5, Compilación sin optimizaciones . . . . .	14
B.6. CU-6, Generación de una visualización del AST en formato PDF	15

## *Apéndice A*

---

# **Plan de Proyecto Software**

---

## **A.1. Introducción**

En este apartado se tiene como objetivo definir y estructurar el plan de proyecto para el desarrollo de un lenguaje de programación y el compilador del mismo, de forma que queden satisfechas sus necesidades y requerimientos técnicos, garantizando la calidad, eficiencia y cumplimiento de unos plazos establecidos. Este plan proporciona un marco de referencia para la gestión del proyecto, incluyendo la planificación temporal así como un estudio de la viabilidad tanto económica como legal.

## **A.2. Planificación temporal**

En este apartado se explica de que forma se aplicará la metodología ágil por excelencia, Scrum, para el control temporal de este proyecto, cabe destacar que al ser un trabajo unipersonal con fin académico puede no reflejar a la perfección el trabajo habitual de esta metodología.

A continuación se muestra una planificación organizada por sprints, donde se prioriza la mejora de un producto mínimo viable (MVP por sus siglas en inglés), de forma que en los primeros sprints se obtiene este MVP y en los próximos sprints se trabaja en ampliar las funcionalidades incrementalmente con cada sprint. Cada dos sprints se realiza una reunión con el tutor, para evaluar el trabajo realizado y planificar los siguientes sprints, teniendo en cuenta lo aprendido anteriormente y las posibles dificultades encontradas.

Así mismo, para mantener un orden durante el desarrollo, se integrará en Github la herramienta ZenHub, la cual ayudará a mantener la planificación

en un tablero estilo Kanban, el cual se integra con las issues y milestones definidas.

A continuación se citan los sprints realizados:

## 1º Milestone: MVP

**Sprint 0 (15/09/2025 - 04/10/2025):** Se investiga y sobre herramientas y conceptos relacionados con los lenguajes y compiladores.

**Sprint 1 (05/10/2025 - 11/10/2025):** Se inicia el repositorio y se trabaja en las primeras dos fases (análisis léxico y sintáctico) del compilador.

**Sprint 2 (12/10/2025 - 18/10/2025):** Primera reunión con el tutor. Se trabaja en la fase de generación de código intermedio del compilador.

**Sprint 3 (19/10/2025 - 24/10/2025):** Se añade una funcionalidad para visualización del AST, además, se incluyen mejoras y se madura el trabajo realizado en los sprints anteriores.

## 2º Milestone: Lenguaje completo

**Sprint 4 (25/09/2025 - 01/11/2025):** Primera ampliación del MVP, se añaden tipos, declaraciones y asignaciones de variables.

**Sprint 5 (02/11/2025 - 08/11/2025):** Segunda ampliación del lenguaje, se añaden las funciones y estructura de control if-else.

**Sprint 6 (09/11/2025 - 14/11/2025):** Tercera ampliación del lenguaje, se añaden bucles while y for. Se añade la fase del compilador responsable de llevar el LLVM IR a código máquina específico de la arquitectura host.

**Sprint 7 (15/11/2025 - 21/11/2025):** Se añade un runtime mínimo personalizado, se integran funciones de la librería estándar de C como printf, strlen e implementa una librería estándar propia con una función toString. Se añade el tipo que modelará el tiempo.

## 3º Milestone: Eventos y tiempo

**Sprint 8 (22/11/2025 - 28/11/2025):** Se añade la estructura event, que modela la principal característica propia del lenguaje, además se trabaja en refactorizaciones de calidad y seguridad de la fase de análisis semántico del compilador.

**Sprint 9 (29/11/2025 - 05/12/2025):** Se realizan importantes mejoras de calidad del software. Se añade shadowing de variables y se comienza a hacer pruebas sobre

**Sprint 10 (06/12/2025 - 12/12/2025):** Se completa una primera versión funcional del runtime, se añaden operadores unitarios ++ y -. Se añade una función print propia del lenguaje.

**Sprint 11 (13/12/2025 - 19/12/2025):** Se mejora la gestión de argumentos con la librería y de errores del compilador mediante registro de información adicional y el uso de una estructura de datos específica para los errores.

**Sprint 12 (20/12/2025 - 26/12/2025):** Se completa el paso de parámetros mediante libffi entre runtime y código compilado. Se añade la fase de compilación.

**Sprint 13 (27/12/2025 - 03/01/2025):** Refactorizaciones para mejorar la calidad del código. Redacción exhaustiva y revisiones de la memoria y anexos.

**Sprint 14 (04/01/2025 - 15/01/2025):** El sprint final.

## A.3. Estudio de viabilidad

En este apartado analizaremos la viabilidad vista desde el punto de vista de recursos económicos y requisitos para llevarlo a cabo dentro del marco legal. Este análisis puede parecer poco necesario en un proyecto académico, sin embargo, sería vital en un proyecto privado real.

### Viabilidad económica

A continuación analizaremos que costes deben ser tenidos en cuenta para el desarrollo de este proyecto, así como una determinación aproximada de los costes específicos y totales.

**Coste humano:** es el coste del salario que se requiere para el equipo que programará durante el proyecto. Puesto que es un proyecto con un único desarrollador, se estima un sueldo mensual de unos 1500€ brutos.

**Coste hardware:** este coste representa todo el material que requiere el equipo para funcionar correctamente, este comprende materiales como ordenadores, teclados, ratones, etc. Dando por hecho que en los 5 meses de

proyecto solo se alquila el material necesario, podría estimarse un alquiler mensual de 200€ por hardware.

**Coste software:** estos son costos asociados comúnmente a desarrollos software, sin embargo, con la elección de software libre vamos a evitar este gasto tan común.

## Coste total

En este caso de un proyecto de 5 meses, los costes los podemos aproximar de la siguiente forma:

Tipo de coste	Precio aproximado
Coste humano	7500€
Coste hardware	1000€
Coste software	0€
<b>Coste total</b>	<b>8500€</b>

Tabla A.1: Costes totales

## Viabilidad legal

A continuación se citan todos las herramientas y librerías utilizadas junto con las licencias asociadas a las mismas:

Herramienta	Licencia
Ubuntu	GPL
ANTLR	BDS
LLVM	UIUC
GCC	GPLv3
CMake	BSD
LaTex	LPPL
Doxygen	GPL
Gtest	BDS
argparse lib	MIT
spdlog lib	MIT

Tabla A.2: Licencias

Como podemos observar, todas las licencias de estas herramientas permiten el uso **gratuito** de las mismas, sin restricciones adicionales que afecten a este proyecto.



## *Apéndice B*

---

# Especificación de Requisitos

---

## B.1. Introducción

En este anexo se especifican los requisitos que debe cumplir el proyecto para considerarse completado, así como los casos de uso que se espera que los usuarios hagan de este sistema.

## B.2. Requisitos funcionales

A continuación se listan los RF (requisitos funcionales) necesarios para completar el proyecto, utilizando para la redacción de estos requisitos el formato *Easy Approach to Requirements Syntax (EARS)*:

**RF-1 Análisis del texto fuente:** El sistema deberá aceptar como entrada programas escritos en el lenguaje fuente definido.

**RF-1.1 Gestión de errores léxicos:** Cuando el sistema detecte errores léxicos durante el análisis del texto de entrada, deberá informar de dichos errores indicando su localización en el código fuente.

**RF-1.2 Información de depuración léxica:** Cuando el usuario habilite el modo de depuración mediante un argumento de compilación, el sistema deberá mostrar la lista de *tokens* generados durante el análisis léxico.

**RF-2 Análisis sintáctico del programa:** El sistema deberá analizar sintácticamente el texto recibido conforme a las reglas de producción definidas para el lenguaje.

**RF-2.1 Gestión de errores sintácticos:** Cuando el sistema detecte un error sintáctico, deberá mostrar un mensaje de error indicando la causa y la posición del error en el código fuente.

**RF-2.2 Información de depuración sintáctica:** Cuando el modo de depuración esté habilitado, el sistema deberá mostrar información relativa al proceso de análisis sintáctico.

**RF-3 Creación del AST:** Tras completar correctamente el análisis léxico y sintáctico, el sistema deberá generar un Árbol de Sintaxis Abstracta (AST) que represente la estructura del programa.

**RF-3.1 Recorrido con patrón *visitor*:** El sistema deberá implementar un patrón *visitor* que permita recorrer de forma ordenada los nodos del AST.

**RF-3.2 Recolecta de información estructural:** Durante el recorrido del AST, el sistema deberá almacenar en cada nodo la información necesaria para las fases posteriores del compilador.

**RF-3.3 Visualización del AST:** Cuando el usuario lo solicite mediante un argumento de compilación, el sistema deberá generar una representación visual del AST.

**RF-4 Análisis semántico:** El sistema deberá realizar un análisis semántico del programa, construyendo la tabla de símbolos y verificando la corrección de tipos.

**RF-4.1 Asociación de símbolos y sus datos:** El sistema deberá asociar cada símbolo declarado con la información relevante para su uso posterior, incluyendo su tipo y su referencia en memoria.

**RF-4.2 Gestión de ámbitos (*scopes*):** El sistema deberá gestionar correctamente los distintos ámbitos del programa, garantizando que el acceso a los símbolos respete las reglas de visibilidad.

**RF-4.3 Comprobaciones de tipos:** Durante el análisis semántico, el sistema deberá comprobar la compatibilidad de tipos en declaraciones, asignaciones y operaciones, detectando usos incorrectos.

**RF-5 Generación de código intermedio:** Una vez superado el análisis semántico, el sistema deberá generar una representación intermedia del programa en LLVM IR.

**RF-5.1 Recorrido del AST para generación de IR:** El sistema deberá recorrer el AST utilizando un patrón *Visitor* para generar el código LLVM IR correspondiente a cada estructura del lenguaje.

**RF-5.2 Uso de identificadores propios:** Durante la generación del código intermedio, el sistema deberá aplicar a los nombres de símbolos una serie de etiquetas como *ptrparapunteroso*

**RF-6 Optimizaciones:** El sistema deberá permitir la optimización del código LLVM IR generado.

**RF-6.1 Control mediante opciones de compilación:** Cuando el usuario lo especifique mediante una argumento de compilación, el sistema deberá generar el código con o sin optimizaciones.

**RF-7 Generación de un ejecutable:** Cuando el programa de entrada sea correcto, el sistema deberá realizar el proceso completo, transformando todo el resultado final en un programa ejecutable para la arquitectura donde se ha ejecutando.

## B.3. Casos de uso

Los casos de uso se describen siguiendo un formato estructurado, basado en la propuesta de Cockburn, que permite detallar de forma clara la interacción entre el usuario y el sistema, así como los distintos flujos de ejecución y los errores que puedan ocurrir durante el proceso.

### Listado de casos de uso

<b>CU-1</b>	<b>Traducción a código ejecutable</b>
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-7
<b>Descripción</b>	El sistema compila un programa escrito en el lenguaje fuente y genera un programa ejecutable.
<b>Precondición</b>	El fichero de entrada existe y es accesible por el sistema.
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. El usuario proporciona un fichero con código fuente.</li> <li>2. El sistema realiza el análisis léxico del programa.</li> <li>3. El sistema realiza el análisis sintáctico y genera el AST.</li> <li>4. El sistema ejecuta el análisis semántico.</li> <li>5. El sistema genera el código intermedio (LLVM IR).</li> <li>6. El sistema genera y enlaza el código objeto.</li> <li>7. El sistema produce un programa ejecutable.</li> </ol>
<b>Postcondición</b>	Se genera un programa ejecutable o se muestran los errores de compilación junto con su información asociada.
<b>Excepciones</b>	El fichero con el código fuente no existe o el código fuente es incorrecto.
<b>Prioridad</b>	Alta

Tabla B.1: CU-1, Traducción a código ejecutable

<b>CU-2</b>	<b>Visualización del estado de compilación</b>
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-1, RF-2, RF-3, RF-4, RF-5
<b>Descripción</b>	El sistema compila un programa fuente y muestra información detallada sobre las distintas fases del compilador, permitiendo analizar su funcionamiento interno o visualizar la estructura del AST.
<b>Precondición</b>	El fichero de entrada existe y es accesible por el sistema.
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. El usuario proporciona un fichero con código fuente.</li> <li>2. El usuario ejecuta el compilador activando el modo de depuración o visualización.</li> <li>3. El sistema realiza las distintas fases de compilación.</li> <li>4. El sistema muestra información detallada del proceso o genera una visualización del AST.</li> </ol>
<b>Postcondición</b>	Se genera un ejecutable junto con información de depuración o se muestran errores detallados de compilación.
<b>Excepciones</b>	El fichero con el código fuente no existe o el código fuente es incorrecto.
<b>Prioridad</b>	Alta

Tabla B.2: CU-2, Visualización del estado de compilación

<b>CU-3</b>	<b>Traducción a código intermedio (IR)</b>
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-5
<b>Descripción</b>	El sistema procesa un programa escrito en el lenguaje fuente y genera un fichero con la representación intermedia del código en LLVM IR.
<b>Precondición</b>	El fichero de entrada existe y es accesible por el sistema.
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. El usuario proporciona un fichero con código fuente.</li> <li>2. El usuario solicita la generación de código IR.</li> <li>3. El sistema realiza las fases de análisis léxico, sintáctico y semántico.</li> <li>4. El sistema genera el fichero con código LLVM IR.</li> </ol>
<b>Postcondición</b>	Se genera un fichero con código IR o se muestran errores de compilación.
<b>Excepciones</b>	El fichero con el código fuente no existe o el código fuente es incorrecto.
<b>Importancia</b>	Alta

Tabla B.3: CU-3, Traducción a código intermedio (IR)

CU-4	Ayuda de la herramienta
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-5
<b>Descripción</b>	El usuario requiere de instrucciones para utilizar el compilador
<b>Precondición</b>	El usuario conoce el formato de ayuda mediante argumentos <code>-h</code> o <code>-help</code>
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. Ejecutar el compilador con el argumento <code>-h</code> o <code>-help</code>.</li> <li>2. Recibir un texto en pantalla con las instrucciones de uso del compilador.</li> <li>3. Utilizar esta información para el resto de casos de uso.</li> </ol>
<b>Postcondición</b>	Una lista de lexemas o un error durante el análisis léxico y sus detalles
<b>Excepciones</b>	El fichero con el código de entrada no existe
<b>Importancia</b>	Media

Tabla B.4: CU-4 Comprobación de la validez léxica de un código

<b>CU-5</b>	<b>Compilación sin optimizaciones</b>
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-6.1
<b>Descripción</b>	El programa parte de un código fuente y realiza el proceso completo a excepción de las optimizaciones finales, el cual sirve para poder comparar con uno optimizado y analizar este proceso
<b>Precondición</b>	El fichero de entrada existe y es correcto a nivel léxico y sintáctico.
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. Escribir un código fuente</li> <li>2. Ejecutar el compilador</li> <li>3. Utilizar el argumento <b>-basic</b></li> <li>4. Recibir errores (en este caso hay que volver al paso nº1) o una salida no optimizada</li> </ol>
<b>Postcondición</b>	El compilador realiza sus funciones habituales con o sin optimizaciones, según la configuración seleccionada.
<b>Excepciones</b>	El fichero con el código fuente no existe o contiene errores
<b>Importancia</b>	Media

Tabla B.5: CU-5, Compilación sin optimizaciones

<b>CU-6 Generación de una visualización del AST en formato PDF</b>	
<b>Actor:</b>	Usuario
<b>Requisitos asociados</b>	RF-3.3
<b>Descripción</b>	El programa parte de un código fuente y realiza el proceso completo, generando un archivo en formato PDF con una representación visual del AST.
<b>Precondición</b>	El fichero de entrada existe y es correcto a nivel léxico y sintáctico.
<b>Acciones</b>	<ol style="list-style-type: none"> <li>1. Escribir un código fuente.</li> <li>2. Ejecutar el compilador.</li> <li>3. Utilizar el argumento <code>-visualizeAST</code></li> <li>4. Recibir errores (en este caso hay que volver al paso nº1) o un archivo llamado <code>AST.pdf</code></li> </ol>
<b>Postcondición</b>	Un programa ejecutable y un archivo con la visualización del AST asociado al programa.
<b>Excepciones</b>	El fichero con el código de entrada no existe o contiene errores.
<b>Importancia</b>	Media

Tabla B.6: CU-6, Generación de una visualización del AST en formato PDF



## *Apéndice C*

---

# **Especificación de diseño**

---

## **C.1. Introducción**

En esta sección se describe cómo se construirá el sistema desde el punto de vista del diseño. El objetivo es definir la estructura general de la aplicación, los modelos de datos que la sustentan y los procedimientos que regirán su funcionamiento. De esta manera se establece un puente entre los requisitos previamente definidos y la posterior implementación, garantizando que el desarrollo siga una organización clara y coherente.

## **C.2. Diseño de datos**

### **Estructuras de datos utilizadas**

AST

Tabla de símbolos

### **Diagramas UML de los objetos del sistema**

### **Tipos de datos del lenguaje**

## **C.3. Diseño arquitectónico**

A continuación se lista los módulos principales:

Analizador léxico o lexer

Analizador sintáctico o parser

Analizador semántico

Generador de código IR

El siguiente diagrama muestra linealmente una ejecución sin errores, donde se puede visualizar el estado de los datos en cada capa:

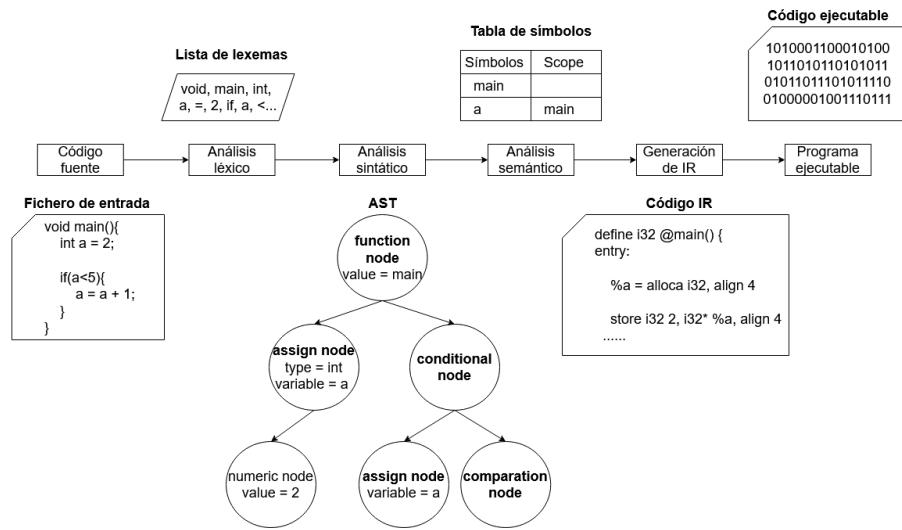


Figura C.1: Diagrama fases del compilador

## C.4. Diseño procedimental

A continuación se explican de qué formas realizan sus tareas los elementos del diseño arquitectónico visto anteriormente:

Algoritmo del lexer (NFA → DFA)

Algoritmo parser (Adaptive LL(\*) / ALL(\*)) que primero intenta decisiones con bajo lookahead (SLL), y si falla, usa contexto completo (modo LL)

Algoritmo para generar la AST (visitor pattern sobre contextos ANTLR)

También visitor pattern para recorrer AST en análisis semántico y generación de IR.

Algoritmo de scheduling de eventos

## *Apéndice D*

---

# Documentación técnica de programación

---

## D.1. Introducción

En esta sección se analizará la estructura del trabajo de programación, cómo se estructuran los ficheros de código, cómo se instalan los requisitos, cómo compilar el proyecto, etc.

## D.2. Estructura de directorios

A continuación se muestra un diagrama con los principales directorios y ficheros presentes en el proyecto.

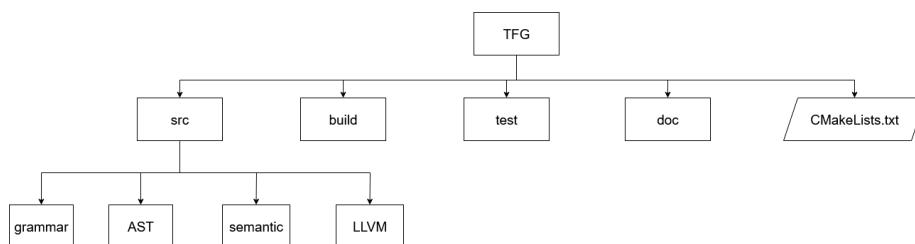


Figura D.1: Diagrama de directorios

## D.3. Manual del programador

El objetivo de este manual es ofrecer una visión técnica del código fuente, de manera que cualquier desarrollador que necesite mantener, extender o comprender el sistema pueda hacerlo.

### Requisitos previos

Instalación de ANTLR y LLVM

### Lenguaje de programación y estilo

El compilador está desarrollado en C++17, empleando un estilo de programación modular y orientado a objetos. Se sigue la convención de nombres en *camelCase* para métodos y variables, y como es estándar de C++, *PascalCase* para clases. El código está documentado con comentarios en formato Doxygen para la generación automática de documentación.

### Convenciones internas

- Los ficheros de cabecera se ubican en el directorio `include/`, y las implementaciones en `src/`.
- La definición del lenguaje de ANTLR se encuentra en el directorio `src/grammar/`.
- El AST y las clases de nodos se implementan en `src/ast/`.
- El análisis semántico se implementa en `src/semantic/`.
- El generador de código intermedio se localiza en `src/LLVM/`.
- El sistema de tests se encuentra en el directorio `test/`.
- La documentación del proyecto se encuentra en `doc/`.

## **D.4. Compilación, instalación y ejecución del proyecto**

### **Compilación**

El fichero CMakeLists.txt será el responsable de compilar el proyecto, mediante CMake, herramienta la cual tiene muy buena integración con C++.

### **Instalación**

### **Ejecución**

## **D.5. Pruebas del sistema**

Para un desarrollo de calidad, debemos asegurarnos que cuando introducimos cambios en el sistema, las características anteriores se mantengan funcionales. Para ello se aportan tests unitarios mediante GoogleTest, un framework diseñado por Google para hacer tests unitarios sobre código C++.

Los tests se encuentran en el directorio "testz se encargan de comprobar la correcta aceptación de código fuente, generación de AST, tablas de símbolos y algunos casos de generación de código IR.



## *Apéndice E*

---

# **Documentación de usuario**

---

## **E.1. Introducción**

Este anexo tiene el propósito de guiar a un usuario final que desee instalar, configurar y usar el compilador. Se describen los requisitos que debe cumplir el usuario, el proceso de Instalación y las instrucciones fundamentales para poder utilizar el compilador de forma correcta.

## **E.2. Requisitos de usuarios**

Antes de comenzar a utilizar el sistema, el usuario debe contar con los siguientes requisitos:

### **Conocimientos**

- Conocimientos básicos de programación
- Familiaridad con un editor de código
- Conocimiento de uso de programas CLI

### **Requisitos de hardware**

- CPU: 1GHz
- RAM: 1GB
- Espacio en disco: 500MB

## Requisitos de software

- Sistema operativo Windows, Linux o macOS.
- Entorno C++

## E.3. Instalación

La descarga se realizará desde el repositorio de código en este enlace.

Una completada la descargada se puede colocar el compilador en la ruta que se desee y utilizar esa misma ruta para ejecutar el compilador, o bien introducir la ruta al compilador como variable de entorno en sistemas Windows o como un alias en sistemas Linux, de forma que podamos ejecutar más comodamente el compilador.

### Variables de entorno (Windows)

### Alias (Linux)

## E.4. Manual del usuario

Para compilar, requerimos de tener un fichero con código y podremos ejecutar el compilador con las siguientes opciones:

- **Compilación básica:**
- **Comprobación de validez léxica:**
- **Comprobación de validez sintáctica:**
- **Compilación a IR:**

## *Apéndice F*

---

# **Anexo de sostenibilización curricular**

---

## **F.1. Introducción**

En este anexo se abordan los conceptos de sostenibilidad, es decir, la búsqueda de un equilibrio entre el desarrollo económico, la protección del medio ambiente y la justicia social, con el objetivo de garantizar un futuro digno para las próximas generaciones.

En el marco de este Trabajo de Fin de Grado, centrado en el desarrollo de un lenguaje de programación y la construcción de su compilador, resulta complejo reflexionar sobre como herramientas como esta, la cual aparentemente queda fuera de estos debates sociales y ambientales, puede también contribuir a los objetivos de sostenibilidad, sin embargo, hay algunos puntos en los que podemos abordar este tema.

## **F.2. Principios de sostenibilidad del proyecto**

A continuación se tratan los principios de sostenibilidad desde el punto de vista de este proyecto: Principio ético: la programación de un compilador debe orientarse a respetar la transparencia, la seguridad de los datos y la accesibilidad del software. La forma más accesible de presentar este software es con una licencia que lo haga gratuito e impulse a los usuarios a utilizarlo en todo tipo de proyectos.

Principio holístico: el compilador no debe entenderse como un simple traductor de lenguaje humano a lenguaje máquina, sino como una pieza de software que tiene un posible impacto en la eficiencia energética de un sistema informático y que puede afectar a la productividad de los desarrolladores.

Principio de complejidad: un software moderno como este debe ser pensado en dimensiones sociales y ambientales, asegurando que el compilador sea eficiente reduciendo el consumo eléctrico así como evitar que el usuario necesite contar con un hardware de altas prestaciones para su ejecución, lo que a largo plazo puede tener un impacto ambiental.

Principio de glocalización: en el marco de la creación de software, podemos incluir este principio tratando de hacer llegar este software a los usuarios en sus lenguas locales y con documentación adaptada a todos los niveles de conocimiento técnico posibles, asegurando sus accesibilidad.

Principio de transversalidad: el criterio de la sostenibilidad no es un factor aislado a algunos de los objetivos del proyecto, sino que debe estar presente durante todo ciclo de diseño así como en el proceso de desarrollo, asegurando todos los principios cubiertos anteriormente.

### **F.3. Competencias transversales en sostenibilidad**

Siguiendo lo indicado en las directrices de la CRUE, se pueden identificar las siguientes competencias transversales vinculadas al desarrollo del compilador:

SOS1: Contextualización crítica del conocimiento. El compilador se ha diseñado no solo como un ejercicio académico, sino también como una herramienta que puede integrarse en contextos reales de desarrollo de software. Esta visión permite relacionar el trabajo con problemáticas más amplias, como la eficiencia energética de los centros de datos o la necesidad de software más accesible.

SOS2: Uso sostenible de recursos. Un compilador eficiente contribuye a que los programas generados consuman menos recursos. Esto tiene un impacto directo en el consumo energético de los dispositivos, lo cual es especialmente relevante en un contexto donde los centros de procesamiento de datos representan una parte significativa del gasto energético global.

SOS3: Participación comunitaria. El software desarrollado puede ponerse a disposición de la comunidad universitaria o profesional, fomentando la

colaboración abierta y el aprendizaje compartido mediante licencias como la MIT, lo que promueve el acceso igualitario al conocimiento.

SOS4: Principios éticos. En la construcción del compilador se ha procurado mantener buenas prácticas de programación, así como transparencia en el funcionamiento de la herramienta, evitando prácticas que puedan considerarse opacas o poco éticas.

## F.4. Impacto social y ambiental del proyecto

El impacto de un compilador sobre la sostenibilidad puede analizarse en dos niveles:

Impacto ambiental: la optimización del código y la reducción del consumo de recursos computacionales contribuyen indirectamente a disminuir el gasto energético. Aunque este efecto pueda parecer limitado en un solo proyecto, los programas generados por los compiladores pueden influir directamente en el consumo de energía de millones de dispositivos.

Impacto social: un compilador bien diseñado puede facilitar el acceso a la programación a estudiantes y profesionales de distintas procedencias. La posibilidad de que la herramienta se difunda como recurso abierto fomenta la democratización del conocimiento, reduciendo barreras de entrada en el aprendizaje de lenguajes de programación.

## F.5. Conclusión

Aunque a primera vista este proyecto pueda parecer desconectado de las cuestiones de sostenibilidad, un análisis más detallado demuestra que es posible establecer relaciones significativas. La eficiencia computacional, la ética en el diseño de software, la accesibilidad y la potencial contribución al aprendizaje colectivo son aspectos que reflejan cómo la informática puede integrarse en un marco más amplio de desarrollo humano sostenible.

De esta manera, el presente trabajo no solo cumple una función académica y técnica, sino que también se alinea con los compromisos institucionales de la universidad en materia de sostenibilidad, reforzando la idea de que toda actividad de investigación y desarrollo debe contemplar su impacto social, económico y ambiental.

[1]

---

## 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, 2006.