



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 — 15 de enero
de 2026

Tutor: César Ignacio García Osorio

Índice general

Índice general	i
Índice de figuras	iii
Índice de tablas	iv
Apéndice A Plan de proyecto software	1
A.1. Introducción	1
A.2. Planificación temporal	1
A.3. Estudio de viabilidad	3
Apéndice B Especificación de requisitos	7
B.1. Introducción	7
B.2. Requisitos funcionales	7
B.3. Casos de uso	9
Apéndice C Especificación de diseño	17
C.1. Introducción	17
C.2. Diseño de datos	17
C.3. Diseño arquitectónico	19
C.4. Diseño procedimental	20
Apéndice D Documentación técnica de programación	23
D.1. Introducción	23
D.2. Manual del programador	23
D.3. Compilación, instalación y ejecución del proyecto	26
D.4. Pruebas del sistema	27

Apéndice E Documentación de usuario	29
E.1. Introducción	29
E.2. Requisitos de usuarios	29
E.3. Instalación	30
E.4. Manual del usuario	31
Apéndice F Anexo de sostenibilización curricular	33
F.1. Introducción	33
F.2. Principios de sostenibilidad del proyecto	33
F.3. Competencias transversales en sostenibilidad	35
F.4. Impacto social y ambiental del proyecto	35
F.5. Conclusión	36
Apéndice G Manual del lenguaje T	37
G.1. Introducción	37
G.2. Estructura de un programa	37
G.3. Comentarios y formato	38
G.4. Tipos de datos	38
G.5. Literales	39
G.6. Variables	40
G.7. Expresiones	41
G.8. Operadores de incremento y decremento	42
G.9. Estructuras de control	42
G.10. Funciones	43
G.11. Eventos	43
G.12. Limitaciones actuales del lenguaje	44
Bibliografía	47

Índice de figuras

C.1. Diagrama de fases del compilador	20
D.1. Diagrama de directorios del proyecto	25

Índice de tablas

A.1. Costes totales.	4
A.2. Licencias.	5
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. 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 y una breve descripción de los mismos.

1º Milestone: MVP

Sprint 0 (15/09/2025 - 04/10/2025): Se investiga 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-the-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 anfitrión.

Sprint 7 (15/11/2025 - 21/11/2025): Se añade un *runtime* mínimo personalizado, se integran funciones de la biblioteca 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 la integración del runtime.

Sprint 10 (06/12/2025 - 12/12/2025): Se completa una primera versión funcional del *runtime*, se añaden operadores unarios `++` 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 optimización del compilador.

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): Realización de otros materiales como diapositivas de presentación, video de demostración y video defensa del proyecto.

Los datos de trazabilidad de los cambios exactos incluidos en cada *milestone*, *sprint* o *issue* puede ser encontrado en GitHub o de igual manera en ZenHub.

A.3. Estudio de viabilidad

En este apartado analizaremos la viabilidad 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 programara 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, se estima un alquiler mensual de 200€ [3] 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€
Coste total	8500€

Tabla A.1: Costes totales.

Viabilidad legal

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

Herramienta/Biblioteca	Licencia
Ubuntu	GPL
ANTLR	BSD
LLVM	UIUC
GCC	GPLv3
CMake	BSD
LaTex	LPPL
Doxygen	GPL
Gtest	BSD
argparse	MIT
spdlog	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.

Para este proyecto, se ha elegido una licencia **GPL**, para garantizar que el código se mantenga accesible, sin posibilidad de privatizarse. Esta decisión se hace siguiendo la filosofía [2] de grandes compiladores como *GNU GCC* que han crecido y mejorado gracias al esfuerzo comunitario que permiten estas licencias.

Apéndice B

Especificación de requisitos

B.1. Introducción

En este anexo se especifican los requisitos que debe cumplir el proyecto para considerarse completado, así las interacciones esperadas entre los usuarios y el 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)* [4]:

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 tókenes 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 `ptr_` para punteros o `_val` para valores.

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 el programa de entrada completo en un ejecutable para la arquitectura donde se ha ejecutado.

B.3. Casos de uso

Los casos de uso se describen siguiendo un formato estructurado, basado en la propuesta de Cockburn [1], 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

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

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

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

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

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

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

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

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

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

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

CU-6	Generación de una visualización del AST en formato PDF
Actor:	Usuario.
Requisitos asociados	RF-3.3.
Descripción	El programa parte de un código fuente y realiza el proceso completo, generando un archivo en formato PDF con una representación visual del AST.
Precondición	El fichero de entrada existe y es correcto a nivel léxico y sintáctico.
Acciones	<ol style="list-style-type: none"> 1. Escribir un código fuente. 2. Ejecutar el compilador. 3. Utilizar el argumento <code>--visualizeAST</code> 4. Recibir errores (en este caso hay que volver al paso nº1) o un archivo llamado <code>AST.pdf</code>.
Postcondición	Un programa ejecutable y un archivo con la visualización del AST asociado al programa.
Excepciones	El fichero con el código de entrada no existe o contiene errores.
Importancia	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 el diseño del sistema desde un punto de vista arquitectónico y conceptual. El objetivo es definir la estructura general del compilador, los modelos de datos que lo sustentan y los procedimientos que rigen su funcionamiento interno.

Este anexo actúa como un puente entre los requisitos funcionales definidos previamente y la implementación final, permitiendo comprender las decisiones de diseño adoptadas y facilitando futuras tareas de mantenimiento, ampliación o refactorización del sistema.

C.2. Diseño de datos

Estructuras de datos utilizadas

El compilador se apoya en dos estructuras de datos principales: el Árbol de Sintaxis Abstracta (AST) y la tabla de símbolos. Ambas estructuras son fundamentales para separar correctamente las distintas fases del proceso de compilación.

Árbol de Sintaxis Abstracta (AST)

El AST representa la estructura semántica del programa fuente, eliminando detalles sintácticos innecesarios y conservando únicamente la información relevante para el análisis y la generación de código.

Cada nodo del AST representa una construcción del lenguaje (expresiones, declaraciones, sentencias, funciones, eventos, etc.). Todos los nodos heredan de una clase base común, lo que permite recorrer el árbol mediante el patrón *visitor* sin acoplar la lógica de procesamiento a la estructura concreta de los nodos.

Esta representación intermedia facilita:

- La realización del análisis semántico.
- El almacenamiento estructurado y organización de los datos.
- La detección temprana de errores.
- Separación de responsabilidades de representación y transformación.
- La generación de código intermedio de forma estructurada.

Tabla de símbolos

La tabla de símbolos almacena la información asociada a los identificadores del programa, como variables, funciones, parámetros y eventos. Está organizada mediante una estructura jerárquica de ámbitos (*scopes*), lo que permite modelar correctamente la visibilidad y el sombreado de identificadores.

Cada ámbito contiene un conjunto de símbolos y mantiene una referencia a su ámbito padre. Este diseño permite:

- Búsqueda eficiente de identificadores.
- Gestión correcta de ámbitos anidados.
- Diferenciación entre bloques y ámbitos de función.

Diagramas del sistema

Debido a la complejidad y tamaño del sistema desarrollado, los diagramas de clases, dependencias, espacios de nombres y relaciones de llamada resultan demasiado extensos para su inclusión directa en el presente documento. No obstante, dichos diagramas pueden ser generados automáticamente a partir del código fuente mediante la herramienta **Doxxygen**.

Los comandos necesarios para generar dicha documentación son los siguientes:

```
doxygen Doxyfile

# Generación de la documentación en formato PDF (LaTeX)
cd doc/doxygen/latex
make
# El documento resultante se genera como refman.pdf
```

Esta aproximación garantiza que los diagramas reflejan fielmente la implementación real del sistema, evitando inconsistencias entre la documentación y el código fuente. Además permiten la posterior ampliación de la documentación mediante la adaptación a diferentes idiomas como se menciona en el anexo F.2

Tipos de datos del lenguaje

El lenguaje implementado soporta un conjunto reducido pero expresivo de tipos de datos, entre los que se incluyen:

- Tipos primitivos: enteros, booleanos, caracteres y valores numéricos en coma flotante.
- Tipo cadena, representado internamente mediante punteros.
- Tipo temporal, utilizado para la gestión de eventos basados en tiempo.
- Tipos referencia, que permiten el paso por referencia y la manipulación indirecta de valores.

Cada tipo del lenguaje se mapea de forma explícita a su correspondiente tipo en LLVM IR, garantizando una generación de código coherente y segura.

C.3. Diseño arquitectónico

El compilador sigue una arquitectura por capas, donde cada módulo corresponde a una fase concreta del proceso de compilación. Esta separación permite aislar responsabilidades y facilita la extensibilidad del sistema.

Los módulos principales del compilador son:

- Analizador léxico (*lexer*).
- Analizador sintáctico (*parser*).

- Constructor del AST
- Analizador semántico.
- Generador de código intermedio.
- Optimizador de código intermedio.
- Generador de código ejecutable.

El siguiente diagrama muestra una ejecución lineal del compilador en ausencia de errores, ilustrando el flujo de datos entre las distintas capas del sistema:

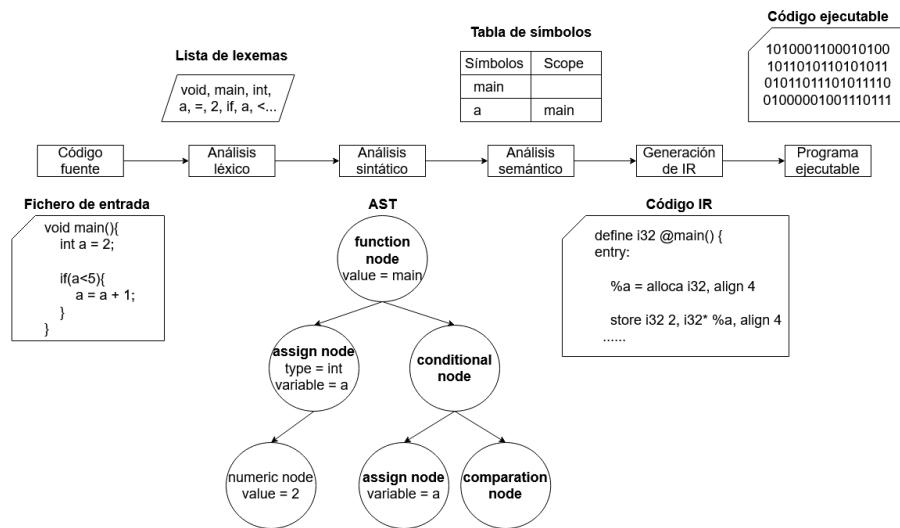


Figura C.1: Diagrama de fases del compilador

Esta arquitectura garantiza que cada fase trabaje únicamente sobre representaciones válidas generadas por la fase anterior.

C.4. Diseño procedimental

A continuación se describen los principales algoritmos y procedimientos empleados en cada uno de los módulos del sistema.

Algoritmo del analizador léxico

El analizador léxico se basa en autómatas finitos, siguiendo el esquema clásico de transformación de una expresión regular en un autómata no determinista (NFA), y posteriormente en un autómata determinista (DFA).

Este proceso permite reconocer de forma eficiente los tókenes del lenguaje, que serán consumidos por el analizador sintáctico.

Algoritmo del analizador sintáctico

El analizador sintáctico utiliza el algoritmo *Adaptive LL(*)* [5] proporcionado por ANTLR. Este enfoque combina dos modos de análisis:

- Modo SLL, con bajo *lookahead* para un análisis rápido.
- Modo LL completo, que utiliza contexto completo cuando el modo SLL falla.

Este mecanismo permite obtener un equilibrio entre eficiencia y expresividad gramatical.

Generación del AST

La construcción del AST se realiza mediante el patrón *visitor*, recorriendo los contextos generados por ANTLR y creando los nodos correspondientes del árbol abstracto.

Este diseño desacopla completamente la gramática del lenguaje de la representación interna del programa.

Análisis semántico y generación de código

Tanto el análisis semántico como la generación de código LLVM IR utilizan el patrón *visitor* para recorrer el AST.

Durante el análisis semántico se validan los tipos, la correcta declaración de identificadores y el uso de ámbitos, generando en este proceso la tabla de símbolos. En la fase de generación de código, la AST se traduce a instrucciones LLVM IR con la ayuda de los datos de la tabla de símbolos, garantizando que se respete la estructura y uso de identificadores del programa fuente.

Algoritmo de *scheduling* de eventos

El sistema de eventos se gestiona en tiempo de ejecución mediante un mecanismo de planificación basado en hilos. Cada evento se registra función la cual será ejecutada según unos parámetros de ejecución, tales como el periodo de ejecución, el límite de ejecuciones y los parámetros de entrada.

El *runtime* se encarga de:

- Iniciar la ejecución del programa.
- Registrar los eventos definidos en el programa.
- Registrar y gestionar los parámetros que se le pasan al evento (en caso de que los haya).
- Lanzar los eventos, permitiendo su autogestión.
- Gestionar el ciclo de vida de los eventos, así como su correcta terminación.

Este enfoque permite implementar un modelo de ejecución reactivo y temporal, integrando de forma transparente la semántica de eventos dentro del lenguaje.

Apéndice D

Documentación técnica de programación

D.1. Introducción

Este anexo tiene como objetivo servir de guía técnica para desarrolladores que deseen comprender, mantener, extender o colaborar en el proyecto. Se describe la estructura del código fuente, las decisiones de diseño más relevantes, los requisitos técnicos, el proceso de compilación y ejecución, así como el sistema de pruebas existente.

La documentación está orientada a programadores con conocimientos previos de C++ y compiladores, y pretende facilitar la continuidad del proyecto sin necesidad de conocer en profundidad su desarrollo original.

D.2. 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 de forma progresiva y estructurada.

Requisitos previos

Para poder compilar y trabajar con el proyecto es necesario disponer de los siguientes elementos:

- Compilador C++ compatible con el estándar C++17.

- LLVM instalado en el sistema (Versión 18).
- ANTLR4:
 - *Java Runtime Environment* (JRE) para la herramienta ANTLR.
 - *Runtime* de ANTLR4 para C++ instalado en el sistema.
- CMake (versión mínima 3.10).
- GoogleTest para la ejecución de las pruebas unitarias.

Lenguaje de programación y estilo

El compilador está desarrollado íntegramente en **C++17**, empleando un estilo de programación modular y orientado a objetos. Este enfoque permite una clara separación de responsabilidades entre los distintos componentes del sistema (análisis léxico, sintáctico, semántico y generación de código).

Como se ha explicado en el capítulo 4 de la memoria, se ha utilizado el estándar de estilos propuesto por LLVM, siendo algunas de sus más apreciables reglas:

Se siguen las siguientes convenciones de estilo:

- *PascalCase* para nombres de clases.
- *camelCase* para métodos y variables.
- Uso explícito de `const` siempre que es posible.
- Preferencia por `std::unique_ptr` y `std::shared_ptr` para la gestión segura y automática de memoria.

El código está documentado mediante comentarios en formato **Doxygen**, lo que permite generar documentación automática a partir del código fuente.

Convenciones internas y organización del código

El proyecto sigue una estructura basada en capas, donde cada módulo corresponde a una fase concreta del compilador:

- **src/grammar/**: Contiene la gramática del lenguaje definida mediante ANTLR4, así como el lexer y parser generados.

- **src/AST/**: Implementa la estructura del Árbol de Sintaxis Abstracta (AST) y el visitante encargado de construirlo a partir del árbol sintáctico.
- **src/semantic/**: Contiene el análisis semántico, incluyendo la tabla de símbolos, la gestión de ámbitos (*scopes*) y la detección de errores semánticos.
- **src/LLVM/**: Implementa el generador de código intermedio LLVM IR mediante el patrón *visitor*, así como la integración con el backend de LLVM.
- **src/compiler/**: Orquesta el flujo completo del compilador, desde la lectura del archivo fuente hasta la generación del ejecutable final.
- **src/runtime/**: Contiene el sistema de ejecución en tiempo de ejecución (*runtime*), incluyendo la gestión de eventos, temporización y funciones auxiliares.
- **tests/**: Incluye los tests unitarios implementados con GoogleTest.
- **tests/input**: Incluye ejemplos de diferentes códigos fuente. Estos son utilizados por los tests unitarios.
- **doc/**: Contiene la documentación del proyecto, incluida la memoria del TFG y sus anexos.

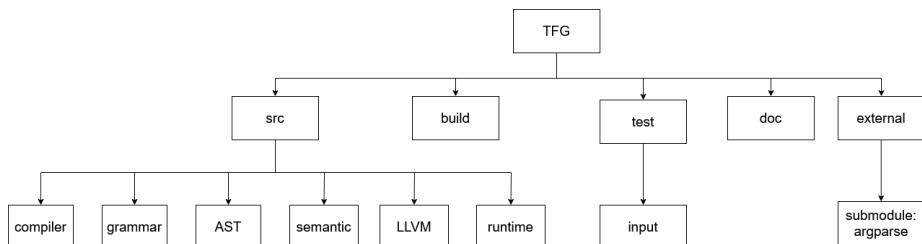


Figura D.1: Diagrama de directorios del proyecto

Esta organización permite que un desarrollador pueda trabajar en una fase concreta del compilador sin necesidad de modificar el resto del sistema.

D.3. Compilación, instalación y ejecución del proyecto

Compilación

El proyecto utiliza **CMake** como sistema de construcción. El fichero `CMakeLists.txt` define todas las dependencias, objetivos y opciones de compilación necesarias.

El proceso de compilación estándar es el siguiente:

```
mkdir build
cd build
cmake ..
make
```

Este proceso genera el ejecutable del compilador, así como los objetos necesarios del *runtime*.

Instalación

El proyecto no requiere un proceso de instalación formal. Una vez compilado mediante el script `./build.sh` o el proceso de `make`, el ejecutable puede copiarse a cualquier directorio del sistema.

De forma opcional, el desarrollador puede añadir la ruta del ejecutable a las variables de entorno o definir un alias para facilitar su uso durante el desarrollo.

Ejecución

El compilador se ejecuta desde línea de comandos indicando como argumento el archivo fuente a compilar. Opcionalmente, se pueden añadir distintas banderas para depuración, visualización del AST o generación de LLVM IR. Estas vienen indicadas en la ayuda del programa de la siguiente forma:

```
-h, --help      shows help message and exits
-v, --version   prints version information and exits
-o, --output    Output object file [nargs=0..1] [default: "out"]
--visualizeAST Generates a AST visualization -pdf file
```

```
--debug      Shows debug data about all the compiler phases.  
--basic      Skips the optimization phase over the LLVM IR module.  
-IR          Generates a LLVM IR file. [nargs=0..1] [default: "ir"]
```

Durante el desarrollo, se recomienda ejecutar el compilador en modo `--debug` para facilitar el análisis del flujo interno de datos del sistema.

D.4. Pruebas del sistema

Para garantizar la estabilidad del sistema ante modificaciones o extensiones, el proyecto incluye un conjunto de pruebas unitarias implementadas mediante **GoogleTest**.

Las pruebas pueden ejecutarse tras la compilación mediante los objetivos generados por CMake, permitiendo validar que los cambios introducidos no rompen el comportamiento previo del compilador.

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 algún editor de código o entorno de desarrollo.
- Conocimiento básico del uso de programas mediante línea de comandos (CLI).

Requisitos de hardware

- CPU: Procesador de al menos 1 GHz.
- RAM: 1 GB de memoria RAM.

- Espacio en disco: 500 MB libres.
- Acceso a internet para descargar la herramienta.

Requisitos de software

- Sistema operativo Windows o Linux.
- Compilador C++ compatible con el estándar C++17.
- LLVM instalado en el sistema.

E.3. Instalación

La descarga del compilador se realiza desde el repositorio de código del proyecto. Una vez descargado, el usuario puede compilar el proyecto siguiendo las instrucciones proporcionadas en el repositorio.

Tras completar la compilación, el ejecutable del compilador puede colocarse en cualquier directorio del sistema. Para facilitar su uso, se recomienda añadir la ruta del ejecutable a las variables de entorno del sistema o definir un alias, según el sistema operativo utilizado.

Variables de entorno (Windows)

En sistemas Windows, se puede añadir la carpeta que contiene el ejecutable del compilador a la variable de entorno PATH siguiendo estos pasos:

- Abrir el panel de *Configuración del sistema*.
- Acceder a *Variables de entorno*.
- Editar la variable PATH del sistema o del usuario.
- Añadir la ruta donde se encuentra el ejecutable del compilador.

Una vez realizado este proceso, el compilador podrá ejecutarse desde cualquier terminal de comandos sin necesidad de indicar su ruta completa.

Alias (Linux)

En sistemas Linux o macOS, se puede crear un alias para facilitar la ejecución del compilador. Para ello, basta con añadir la siguiente línea al archivo `~/.bashrc` o `~/.zshrc`:

```
alias Tcompiler="/ruta/al/ejecutable/TCompiler"
```

Tras recargar la configuración del terminal, el compilador podrá ejecutarse escribiendo simplemente `Tcompiler`.

E.4. Manual del usuario

Para compilar un programa escrito en el lenguaje, es necesario disponer de un fichero fuente con el código a compilar. El compilador se ejecuta desde la línea de comandos utilizando la siguiente sintaxis general:

```
Tcompiler <archivo_fuente> [opciones]
```

A continuación se describen las principales opciones disponibles:

- `Tcompiler -h / --help`: Muestra un mensaje de ayuda con la descripción de las opciones disponibles.
- `Tcompiler <archivo_fuente>`: Compila el archivo fuente especificado y genera un archivo objeto como salida.
- `Tcompiler <archivo_fuente> --debug`: Activa el modo de depuración, mostrando información adicional como la lista de tokens, la tabla de símbolos y el código LLVM IR generado.
- `Tcompiler <archivo_fuente> --visualizeAST`: Genera una representación visual del Árbol de Sintaxis Abstracta (AST) en formato PDF.
- `Tcompiler <archivo_fuente> -IR <archivo_IR>`: Exporta el código LLVM IR generado al archivo especificado.
- `Tcompiler <archivo_fuente> -o <nombre_programa>`: Permite especificar el nombre del archivo objeto o ejecutable generado por el compilador.

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

En esta sección se analizan los principios de sostenibilidad aplicables al presente proyecto, describiendo su relación con las decisiones de diseño y con el enfoque adoptado en el desarrollo del software asociado.

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 usuario en sus lenguas locales y con documentación adaptada a todos los niveles de conocimiento técnico posibles, asegurando sus accesibilidad. Por motivos de tiempo, esto no ha sido posible realizarlo y se deja registrado en el apartado de trabajos futuros.

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.

Apéndice G

Manual del lenguaje T

G.1. Introducción

El lenguaje T es un lenguaje de programación imperativo y de tipado estático, con una sintaxis inspirada en lenguajes de la familia C. Su principal rasgo distintivo es la incorporación de *eventos temporales* y *eventos condicionados*, que permiten programar comportamientos reactivos basados en el tiempo o en condiciones lógicas del programa.

Un programa en T se compone de declaraciones de variables, expresiones, estructuras de control, definiciones de funciones y definiciones de eventos, los cuales pueden ejecutarse de forma automática según las reglas temporales o condicionales especificadas.

Este capítulo describe el lenguaje desde el punto de vista del programador, detallando su sintaxis y semántica básica, sin entrar en aspectos internos del compilador.

G.2. Estructura de un programa

Un programa en T consiste en una secuencia de sentencias que se ejecutan en el orden en el que aparecen. No existe una función `main` explícita; el bloque principal del programa actúa como punto de entrada.

```
int a;  
a = 10;
```

```
print(a);
return 0;
```

La instrucción `return` puede aparecer en el bloque principal, indicando el valor de salida del programa.

G.3. Comentarios y formato

Comentarios

T admite comentarios de una sola línea, iniciados por `//`. Todo el texto desde este símbolo hasta el final de la línea es ignorado por el compilador.

```
// Esto es un comentario
int a;
```

Espacios en blanco

Los espacios, tabuladores y saltos de línea no tienen significado semántico y se utilizan únicamente para mejorar la legibilidad del código.

Identificadores

Los identificadores representan nombres de variables, funciones y eventos. Deben comenzar por una letra y pueden contener letras y dígitos.

Ejemplos válidos:

- a
- contador1
- miFuncion2

G.4. Tipos de datos

El lenguaje T define los siguientes tipos primitivos:

- int

- float
- char
- string
- bool
- void
- time

Además, existe el modificador `ref`, utilizado exclusivamente para indicar paso de parámetros por referencia en funciones.

G.5. Literales

Literales numéricicos

Los literales enteros y reales se escriben de la forma habitual:

10
3.14

Literales de cadena

Las cadenas de texto se escriben entre comillas dobles:

"Ejemplo de string"

Literales booleanos

El lenguaje define dos literales booleanos:

`true`
`false`

Literales temporales

El tipo `time` representa valores temporales expresados mediante una cantidad numérica seguida de una unidad. Las unidades disponibles son:

- `tick`
- `sec`
- `min`
- `hr`

Ejemplos:

```
5 sec
2.5 min
10 tick
1 hr
```

Durante el proceso de compilación todos los valores son transformados a *tick*, cada tick equivale a 100 ms

G.6. Variables

Declaración de variables

Las variables se declaran especificando su tipo y su identificador:

```
int a;
char a;
float x;
string name;
bool ok;
time t;
```

Asignación

Una variable puede recibir un valor mediante el operador de asignación `=`:

```
a = 10;
x = 2.718;
name = "Adrian";
ok = true;
```

También es posible declarar e inicializar una variable en una única sentencia:

```
int a = 10;
float x = 3.14;
```

G.7. Expresiones

Expresiones aritméticas

T soporta los operadores aritméticos básicos: suma `+`, resta `-`, multiplicación `*`, división `/` y módulo `%`.

```
int a = 2 + 3 * 4;
float b = (1.0 + 2.0) / 3.0;
```

Los paréntesis permiten modificar la precedencia de evaluación.

Expresiones relacionales

Se admiten las comparaciones:

`==` `!=` `<` `<=` `>` `>=`

```
if (a < 10) {
    print(a);
}
```

G.8. Operadores de incremento y decrecimiento

El lenguaje admite operadores de incremento y decremento tanto en forma prefija como postfija:

```
a++;
--a;
```

Estos operadores pueden formar parte de expresiones más complejas.

G.9. Estructuras de control

Condicionales

La estructura condicional se define mediante `if` y `else`:

```
if (a < 0) {
    print("Negativo");
} else if (a < 10) {
    print("Pequeño");
} else {
    print("Grande");
}
```

Bucles

Bucle `while`

```
while (a < 10) {
    a++;
}
```

Bucle `for`

El bucle `for` tiene la forma:

```
for (int i = 0; i < 10; i = i + 1) {
    print(i);
}
```

Control de bucle

Se permiten las sentencias de control en bucles:

- **break**: Termina el bucle
- **continue**: Vuelve a evaluar la condición

G.10. Funciones

Declaración y definición

Una función puede declararse sin cuerpo:

```
int function suma(int a, int b);
```

O definirse completamente:

```
int function suma(int a, int b) {
    return a + b;
}
```

Parámetros por referencia

El modificador **ref** permite pasar parámetros por referencia:

```
void function inc(ref int a) {
    a = a + 1;
}
```

G.11. Eventos

Los eventos constituyen la característica diferenciadora del lenguaje T. Un evento define un bloque de código que se ejecuta automáticamente según una condición temporal o lógica.

Eventos temporales

Los eventos temporales se definen mediante los comandos `every`, `after` y `at`.

```
event tickLogger every 1 sec {
    print("tick");
}

event bomba after 5 sec {
    print("boom");
}
```

En el estado actual del proyecto, `every` se encuentra plenamente implementado y soportado por el compilador y el *runtime*. Por otro lado, las construcciones `after` y `at` han sido definidas y reconocidas por el lenguaje, pero su implementación completa no ha sido abordada debido a limitaciones de tiempo. No obstante, se han reservado como parte de la sintaxis del lenguaje con el objetivo de facilitar su incorporación en futuras ampliaciones.

Límite de ejecuciones

Un evento puede limitar su número de ejecuciones mediante `limit`:

```
event hello every 1 sec limit 3 {
    print("hola");
}
```

Finalización de eventos

Dentro de un evento puede utilizarse la instrucción `exit` seguido del identificador de un evento, finalizando así este evento en concreto:

```
exit tickLogger;
```

G.12. Limitaciones actuales del lenguaje

En su estado actual, el lenguaje presenta las siguientes limitaciones:

- No se soportan estructuras de datos complejas como arrays o **struct**.
- El tipo **ref** solo está disponible en parámetros de funciones. Puesto que el paso en eventos requiere de gestión avanzada de memoria.
- No existe mecanismos para la definición o inclusión de bibliotecas.
- Por cuestión de tiempo, los eventos no han podido enriquecerse con otros mecanismos de ejecución.

Bibliografía

- [1] Alistair Cockburn and Lord Cockburn. *Writing effective use cases*. Pearson Education India, 2008.
- [2] GNU. License. <https://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html>. Último acceso: diciembre-2025.
- [3] Alquiler Ordenadores Madrid. Alquiler de ordenadores portÁtiles en madrid. <https://alquilerordenadoresmadrid.es/>. Último acceso: diciembre-2025.
- [4] Alistair Mavin. Alistair Mavin EARS method description. <https://alistairmavin.com/ears/>. Último acceso: 2025-12-15.
- [5] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll (*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices*, 49(10):579–598, 2014.