uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería de Software

Trabajo Fin de Grado

"Desarrollo de un Compilador Genérico de Lenguaje Ensamblador para el Simulador CREATOR"

Autor

Álvaro Guerrero Espinosa

Tutor

Félix García Carballeira

Leganés, Madrid, España

Junio 2025

Cualquier ingenuo puede escribir código que un computador puede entender. Los buenos programadores escriben código que los humanos pueden entender

— Martin Fowler, ***Refactoring: Improving the Design of Existing Code***

RESUMEN

Este trabajo presenta un ensamblador genérico, robusto y flexible para su uso en el simulador CREATOR que permite mejorar las capacidades de esta herramienta. Está diseñado para ser utilizado por los alumnos de la asignatura *Estructura de computadores*.

Este ensamblador es capaz de utilizar múltiples arquitecturas distintas definidas mediante un fichero de configuración. En esta configuración se pueden definir muchas características distintas, como las instrucciones, registros, y directivas definidas. Una vez especificada una arquitectura, este ensamblador permite compilar programas en lenguaje ensamblador para dicha arquitectura, utilizando funcionalidades avanzadas como expresiones aritméticas o etiquetas como valores numéricos. Además, está diseñado para ser capaz de generar mensajes de error útiles que ayuden a los usuarios a aprender el lenguaje ensamblador.

Debido a estas características, con este ensamblador se busca mejorar la enseñanza del lenguaje ensamblador a los estudiantes de esta asignatura, y mejorar las capacidades de CREATOR para el desarrollo de nuevas arquitecturas.

Palabras clave: Ensamblador • CREATOR • Genérico • Compilador

DEDICATORIA

Querría comenzar agradeciendo a mis padres por haberme permitido llegar hasta aquí, al pagar la matrícula y permitirme centrarme en los estudios sin necesidad de trabajar a tiempo parcial.

En segundo lugar, quiero agradecer a mi tutor Félix por permitirme trabajar en este proyecto y proponerme la idea cuándo no sabía sobre qué hacer este trabajo. Este proyecto me ha permitido aplicar muchos de los conocimientos aprendidos en la carrera y aprender mucho.

Quiero darle las gracias a José Antonio por ayudarme con su conocimiento de ensamblador a decidir las funcionalidades a implementar en el sistema y a encontrar fallos en el mismo. Creó un muy buen caso de prueba para verificar el correcto funcionamiento y rendimiento del sistema completo. Me aguantó muchas tardes en la universidad mientras trabajaba en este proyecto y me permitió discutir con él problemas que estaba teniendo.

También quiero darle las gracias a Alejandro Calderón por ser uno de mis mejores profesores durante la carrera y darme consejos durante la realización de este proyecto.

También me gustaría darle las gracias a todos mis profesores por enseñarme los conocimientos necesarios para realizar este proyecto, y a mis amigos de la universidad por permitirme llegar hasta aquí y ayudarme cuando lo necesitaba.

Por último, me gustaría darle las gracias a la comunidad FOSS. Aunque no he interactuado mucho directamente, la mayoría de las herramientas que uso fueron creadas por ellos y he aprendido mucho leyendo mensajes en StackOverflow y GitHub. Las pocas veces que he contribuido a algún proyecto FOSS siempre he sido bienvenido, y admiro su esfuerzo por mejorar el mundo con *software* libre.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN

En este primer capítulo se va a presentar el proyecto. Primero se explicará su motivación (Sección 1.1, *Motivación*). Tras esto, los objetivos a lograr (Sección 1.2, *Objetivos*). Finalmente, se indicará la estructura de este documento (Sección 1.3, *Estructura del documento*).

1.1. Motivación

Este Trabajo Fin de Grado (TFG) se ha desarrollado como parte de una beca de colaboración con el Departamento de Informática financiada por el Ministerio de Educación, Formación Profesional y Deportes. Dentro de esta colaboración, el TFG se ha enmarcado dentro del contexto del proyecto investigación "Entorno de desarrollo integrado para formación e investigación en procesadores RISC-V" financiado por la Agencia Estatal de Investigación dentro de la convocatoria de proyectos "Prueba de Concepto" 2023-PERTE CHIP, dentro del cual se ha desarrollado el compilador de ensamblador desarrollado en este TFG.

La programación es el acto de componer secuencias de instrucciones que indiquen a un computador cómo realizar una tarea. Para realizar esto, hay que definir esta secuencia en un lenguaje de programación, creando un programa. Según el nivel de abstracción sobre el computador, los lenguajes de programación se pueden clasificar en un espectro, donde los lenguajes de bajo nivel utilizan pocas o ninguna abstracción mientras que los lenguajes de alto nivel utilizan muchas abstracciones.

Aunque estas abstracciones ayudan al desarrollo de programas, en todos los casos estos acaban siendo ejecutados en un computador. Es por esto que el funcionamiento del computador siempre tiene una influencia en todos los lenguajes, y determina las consecuencias del uso de las diferentes abstracciones ofrecidas por los lenguajes de alto nivel.

Debido a esto, para diseñar programas que sean capaces de utilizar eficientemente todos los recursos de un computador, es fundamental conocer su funcionamiento.

Un computador es una combinación de *hardware* y *software*, que forman la arquitectura del mismo. El *hardware* determina las capacidades del computador, mientras que el *software* indica cómo se utilizan estas capacidades para realizar una tarea. Conocer ambos componentes es fundamental para entender el funcionamiento del computador, qué es capaz de realizar, y cómo lo hace.

El *software* que puede ejecutar directamente un computador se escribe en una familia de lenguajes de bajo nivel conocidos como "lenguajes ensamblador". Para simplificar la electrónica y aumentar el rendimiento del computador, estos están formados únicamente por una secuencia de instrucciones atómicas simples que operan en unos pocos datos. Para ejecutar cualquier programa escrito en un lenguaje de mayor nivel es necesario convertirlos a uno de estos lenguajes ensamblador mediante un proceso conocido como "compilación".

Debido a todo esto, conocer el funcionamiento de los lenguajes ensamblador es fundamental para cualquier ingeniero informático, ya que permite entender el funcionamiento de un computador y utilizarlo de forma efectiva para llevar a cabo una tarea.

Actualmente, para enseñar estos lenguajes ensamblador se utilizan simuladores, que permiten al usuario visualizar el estado del computador a medida que ejecuta las instrucciones y pausar su ejecución en cualquier momento, de una forma similar a las herramientas de depuración desarrolladas para los lenguajes de alto nivel. Un componente importante de estos simuladores es su ensamblador (también conocido como compilador), encargado de convertir la representación textual del código en lenguaje ensamblador a una representación que pueda ser ejecutada en el simulador. Para permitir a los usuarios entender el funcionamiento del lenguaje, es necesario que este ensamblador detecte acciones no permitidas, y sea capaz de producir mensajes de error que ayuden a los usuarios a entender y solucionar estos problemas. Además, este ensamblador tiene que ser lo suficientemente flexible como para permitir realizar todas las acciones permitidas por un computador real.

Además, para que la enseñanza pueda seguir al rápido desarrollo de las arquitecturas de un computador, estos simuladores se han vuelto genéricos: permiten modificar muchos parámetros del computador simulado mediante un fichero de configuración, con pocos (o ningún) cambio en el código. Esto permite simular múltiples arquitecturas diferentes en una misma herramienta sin necesidad de desarrollar un simulador completo para cada una.

CREATOR [1] es un simulador genérico desarrollado por el grupo de investigación ARCOS de la Universidad Carlos III de Madrid. Se desarrolló con fines didácticos para ayudar a los estudiantes a aprender el lenguaje ensamblador en la asignatura *Estructura de Computadores*. En lo que respecta al simulador, CREATOR es muy flexible y tiene muchas funcionalidades. Sin embargo, se ve limitado por su ensamblador, que permite

únicamente funcionalidades básicas.

Por lo tanto, proponemos desarrollar un ensamblador genérico para CREATOR que sea robusto, lo suficientemente flexible como para permitir el uso de funcionalidades avanzadas, y esté centrado en producir buenos mensajes de error que ayuden a los usuarios a aprender el lenguaje ensamblador.

1.2. Objetivos

El principal objetivo de este proyecto es desarrollar un compilador de lenguaje ensamblador genérico que sustituya al actualmente utilizado por el simulador CREATOR, añadiendo nuevas funcionalidades y mejorando sus mensajes de error. Esto ayudará a los alumnos a aprender el lenguaje ensamblador.

A partir de este objetivo principal, se pueden definir objetivos secundarios:

- **O1:** Utilizar diferentes juegos de instrucciones.
- **O2:** Compilar cualquier programa ensamblador escrito con las instrucciones definidas.
- **O3:** Generar mensajes de error con mucha información que ayuden a los estudiantes comprender la causa del problema.
- **O4:** Ser compatible con las definiciones de arquitecturas actualmente desarrolladas para CREATOR.
- **O5:** Ser lo suficientemente flexible como para añadir nuevas funcionalidades, que permitan el uso de CREATOR en nuevas situaciones.

1.3. Estructura del documento

Este documento estará formado por los siguientes capítulos:

- Capítulo 1, *Introducción*, realiza una introducción al proyecto, explicando sus motivaciones y objetivos. Además, también contiene una descripción de los contenidos del documento.
- Capítulo 2, *Estado del arte*, analiza el estado de los simuladores y ensambladores existentes, las diferentes técnicas utilizadas para desarrollar analizadores sintácticos, los mensajes de error generados por diversos compiladores, y los lenguajes de programación utilizados para el desarrollo de estas herramientas.
- Capítulo 3, *Análisis*, presenta el proyecto y define sus requisitos y casos de uso.

- Capítulo 4, *Diseño*, explica el diseño y arquitectura del sistema creado, y sus componentes.
- Capítulo 5, *Implementación y Despliegue*, presenta los detalles de implementación del sistema, y explica cómo realizar su despliegue.
- Capítulo 6, *Validación, verificación, y evaluación*, indica cómo se ha verificado el correcto desarrollo del sistema desarrollando, detallando las pruebas y comparándolo con otras herramientas.
- Capítulo 7, *Plan del proyecto*, describe la planificación seguida en el proyecto y su presupuesto. También discute el marco regulador y entorno socioeconómico en el que se desarrolla el mismo.
- Capítulo 8, *Conclusiones y trabajos futuros*, expone las conclusiones obtenidas con la realización del proyecto, contribuciones adicionales realizadas durante el mismo, y los trabajos futuros que podrían mejorar el sistema.

CAPÍTULO 2

ESTADO DEL ARTE

En este capítulo se analizará el estado del arte, detallando el estado de las diferentes tecnologías relacionadas con el desarrollo del proyecto. Este capítulo estará dividido en cinco secciones. La primera (Sección 2.1, *Simuladores de ensamblador*), presentará los simuladores de lenguaje ensamblador. La segunda (Sección 2.2, *Ensambladores*), analizará los ensambladores utilizados en la actualidad. La tercera (Sección 2.3, *Técnicas de análisis sintáctico*), describirá las diferentes técnicas utilizadas para implementar analizadores sintácticos. Tras esto, la Sección 2.4, *Mensajes de error*, analizará la información contenida en los mensajes de error generados por diversos compiladores. Por último, la Sección 2.5, *Lenguajes de programación*, estudiará los lenguajes de programación utilizados en la actualidad para implementar las diversas herramientas utilizadas durante el desarrollo de *software*.

2.1. Simuladores de ensamblador

Un simulador de ensamblador es un programa que permite simular la ejecución de un programa escrito en ensamblador, simulando el comportamiento de un procesador que ejecuta el código. Suelen tener un propósito didáctico, para ayudar al usuario a aprender el funcionamiento del lenguaje y el procesador, aunque algunos también se pueden utilizar de forma profesional para desarrollar nuevas ISAs.

Estos simuladores están compuestos de dos componentes principales: un ensamblador y un ejecutor. El ensamblador se encarga de procesar el código ensamblador introducido por el usuario a un formato que pueda ser utilizado por el simulador, mientras que el ejecutor se encarga de ejecutar el código generado por el ensamblador en el procesador simulado. Los simuladores se pueden clasificar en específicos, si estos componentes solo permiten simular una ISA específica, o genéricos, si se pueden configurar para simular

varias ISAs distintas.

Como el objetivo del proyecto es la creación de un ensamblador para un simulador, en esta sección se van a analizar varios simuladores existentes en la actualidad.

2.1.1. Simuladores específicos

Un simulador específico está diseñado para simular una única ISA, como RISC-V o ARM, y no permite su modificación. Esto hace que sean más simples que los simuladores genéricos, y, debido a esto, la mayoría de los simuladores en la actualidad entran dentro de esta categoría. En esta sección se exponen dos ejemplos de este tipo de simuladores: Spike y Kite.

Spike

Spike [2] es el simulador de referencia para RISC-V, desarrollado por RISC-V International. Está escrito en C++, y es capaz de emular un sistema RISC-V completo. Está diseñado para servir como un punto de inicio para ejecutar código RISC-V. Debido a esto, permite ejecutar casi cualquier programa.

Spike cuenta con una interfaz por línea de comandos (CLI) (Figura 2.1), y permite seleccionar el modelo de memoria y las extensiones de la ISA a utilizar (contando con soporte para la mayoría de las extensiones estándar). Permite ejecutar código instrucción a instrucción, visualizar el contenido de los registros y la memoria, I/O, y ejecutar código en modo usuario y privilegiado. También cuenta con un modo básico de depuración interactivo, y permite utilizar `gdb` [3] en caso de necesitar un depurador más avanzado. Aunque es un simulador específico de RISC-V, permite añadir y probar nuevas instrucciones.

```
$ ./spike -d pk a.out
(spike) rs 100005
:
(spike) run 10
core   0: 0x0000000080004678 (0x80858593) addi    a1, a1, -2040
core   0: 0x0000000080004680 (0x016050ef) jal     pc + 0x5016
core   0: >>>>  strcmp
core   0: 0x0000000080009696 (0x00054783) lbu     a5, 0(a0)
core   0: 0x0000000080009698 (0x00000505) c.addi  a0, 1
core   0: 0x0000000080009699c (0x00000585) c.addi  a1, 1
core   0: 0x0000000080009696e (0xfff5c703) lbu     a4, -1(a1)
core   0: 0x00000000800096a2 (0x0000c799) c.beqz  a5, pc + 14
core   0: 0x00000000800096a4 (0xfee789e3) beq     a5, a4, pc - 14
core   0: >>>>  strcmp
core   0: 0x0000000080009696 (0x00054783) lbu     a5, 0(a0)
core   0: 0x0000000080009698 (0x00000505) c.addi  a0, 1
(spike) reg 0 a0
0x0000000000001559
(spike) pc 0
0x000000008000969c
(spike) mem 0 0x1000
0x0202859300000297
(spike) rs
Factorial calculator:
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

Fig. 2.1. Interfaz de Spike

Kite

Kite [4] [5] es un simulador que modela un procesador RISC-V con un *pipeline* de cinco etapas, implementado en C++. Su primera versión se desarrolló en 2019 con propósitos educativos para la asignatura *Arquitectura de Computadores*. Este simulador está basado en el modelo de procesador descrito en *Computer Organization and Design, RISC-V Edition: The Hardware and Software Interface* [6]. Su objetivo es ofrecer a los estudiantes un simulador fácil de usar con un modelo de temporización preciso que siga lo descrito en el libro.

Kite implementa la mayoría de las instrucciones básicas descritas en el libro. Este simulador ofrece funcionalidades avanzadas del modelo de *pipeline*, como comprobaciones de dependencias (riesgos de datos), *stalls*, memoria caché de datos, y opcionalmente envío adelantado y predicción de ramas. Además, también cuenta con una opción de depuración que permite ver el progreso detallado de la ejecución de instrucciones en el *pipeline*. Esto ayuda a los estudiantes a mejorar su comprensión del *pipeline* de un procesador.

Kite se utiliza mediante una CLI (Figura 2.2) que recibe el nombre del fichero con el código ensamblador a ejecutar. Además, utiliza dos ficheros adicionales, `reg_state` y `mem_state`, que deben contener el estado inicial de los registros y memoria a utilizar. El simulador ejecuta el programa dado, opcionalmente mostrando información sobre el progreso de la ejecución de instrucciones en el *pipeline*. Al terminar la ejecución del programa, muestra una serie de estadísticas sobre esta, como el número de ciclos de reloj utilizados, el número de ciclos perdidos durante *stalls*, los ciclos de reloj consumidos por cada instrucción de media, y estadísticas de la memoria caché. Por último, también muestra el estado final de los registros y las direcciones de memoria accedidas.

```
$ ./kite program_code
************************************************************
* Kite: Architecture Simulator for RISC-V Instruction Set  *
* Developed by William J. Song                             *
* Computer Architecture and Systems Lab, Yonsei University *
* Version: 1.12                                            *
************************************************************

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = 48
Total number of stalled cycles = 6
Total number of executed instructions = 17
Cycles per instruction = 2.824

Data cache stats:
    Number of loads = 0
    Number of stores = 1
    Number of writebacks = 0
    Miss rate = 1.000 (1/1)

Register state:
x0 = 0
x1 = 0
x2 = 4096
x3 = 0
x4 = 0
x5 = 0
x6 = 0
x7 = 0
x8 = 0
x9 = 0
x10 = 3
x11 = 0
x12 = 0
x13 = 0
x14 = 0
x15 = 0
x16 = 0
x17 = 0
x18 = 0
x19 = 0
x20 = 32
x21 = 400
x22 = 720
x23 = 0
x24 = 0
x25 = 0
x26 = 0
x27 = 0
x28 = 0
x29 = 0
x30 = 0
x31 = 0

Memory state (only accessed addresses):
(2000) = 3

======== [End of Pipeline Stats] =========
```

Fig. 2.2. Interfaz de Kite

2.1.2. Simuladores genéricos

Un simulador genérico permite simular varias ISAs distintas. Esto añade bastante complejidad a su diseño, ya que los componentes necesitan soportar muchas opciones de configuración distintas para permitir especificar la ISA. En esta sección se exponen dos ejemplos de este tipo de simuladores: Sail y CREATOR.

Sail

Sail [7] es un lenguaje para definir la ISA de un procesador desarrollado por la Universidad de Cambridge.

Esta herramienta permite validar la definición de la ISA y generar simuladores en C y OCaml. Además, también permite ejecutar pruebas de validación y generar versiones de

la ISA que se pueden ejecutar con un probador de teoremas para realizar demostraciones sobre su comportamiento. Actualmente, está en desarrollo un componente para generar un modelo de referencia en un lenguaje de descripción de *hardware*. Todas estas características lo hacen una muy buena opción para el uso profesional, aunque esto también lo vuelve una mala opción para el uso didáctico debido a su gran complejidad.

CREATOR

CREATOR (didaCtic and geneRic assEmbly progrAmming simulaTOR) [1] es un simulador didáctico desarrollado por el grupo de investigación ARCOS de la Universidad Carlos III de Madrid. Tiene tanto una versión web (Figura 2.3) como una versión de CLI, y se desarrolló para ayudar a los estudiantes a aprender el lenguaje ensamblador en la asignatura *Estructura de Computadores*.

CREATOR cuenta con un editor de código con resaltado de sintaxis para escribir el código ensamblador, y una interfaz que permite ver el estado del procesador (registros de control, de enteros, y de coma flotante) y la memoria (memoria de datos, memoria de texto, y *stack*), las instrucciones cargadas, y diversas estadísticas sobre la ejecución del programa. Tiene soporte para la ejecución instrucción a instrucción, I/O, llamadas al sistema, y *breakpoints*. Además, también permite verificar si un programa cumple la convención de paso de parámetros.

Con respecto a la modificación de la ISA, CREATOR es muy flexible. Permite modificar la arquitectura del procesador (registros definidos y su nombre, tipo, tamaño y diferentes propiedades), la organización de la memoria, y las instrucciones, pseudo-instrucciones, y directivas de ensamblador permitidas en el código. Con respecto a las instrucciones y pseudo-instrucciones, permite definir su nombre, sintaxis, argumentos y la codificación de estos en binario, tamaño, número de ciclos para su ejecución, y su definición, ya sea las acciones a realizar durante su ejecución para las instrucciones o la secuencia de instrucciones por la que reemplazarla para las pseudo-instrucciones. Todo esto se realiza mediante un fichero de configuración en formato JavaScript Object Notation (JSON), permitiendo el uso de JavaScript (JS) para ciertas opciones como la ejecución de una instrucción. Utilizando esto, CREATOR cuenta con definiciones para RISC-V y MIPS.

2.2. Ensambladores

Un ensamblador es un programa que traduce el código de otro programa escrito en lenguaje ensamblador a código máquina, que puede ser ejecutado en un procesador. Forma parte de la *toolchain* utilizada para compilar un programa. Aunque la gran mayoría se diseñan para una ISA concreta, algunos pueden procesar código para múltiples ISAs distintas.

Cada ensamblador utiliza una sintaxis propia, por lo que, en general, el código se escribe para un ensamblador concreto. En el caso de las instrucciones de x86, por ejemplo,

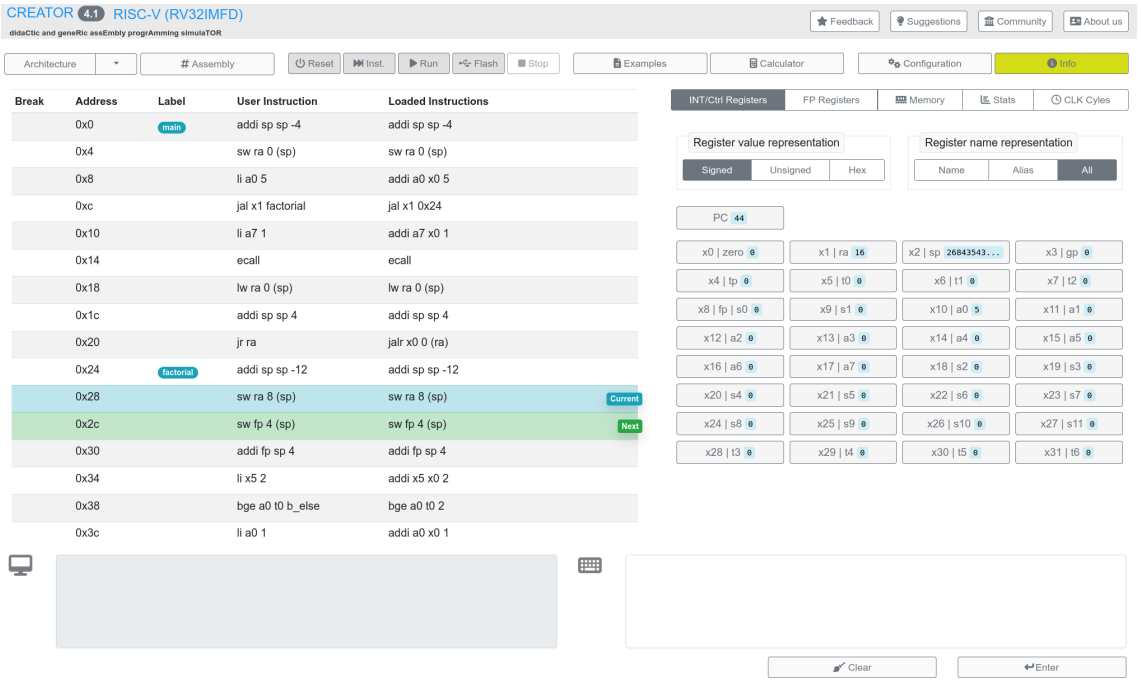

Fig. 2.3. Interfaz principal de CREATOR (Web)

existen dos grandes familias: la sintaxis de Intel y la sintaxis de AT&T. La sintaxis de AT&T se utiliza principalmente en ensambladores antiguos, mientras que los modernos utilizan la sintaxis de Intel. Algunas de las principales diferencias entre estas son: [8]

- **Orden de los argumentos:** en la sintaxis de AT&T el destino de la operación es el primer argumento de las instrucciones, mientras que en la sintaxis de Intel es el último.
- **Prefijo de los valores:** en la sintaxis de AT&T cada argumento de una instrucción requiere un símbolo como prefijo para indicar su tipo, mientras que en la sintaxis de Intel esto no es necesario, ya que el ensamblador deduce el tipo automáticamente.
- **Direcciones de memoria:** en la sintaxis de AT&T las direcciones de memoria se especifican con una sintaxis similar a una llamada a función, mientras que en la sintaxis de Intel se utilizan expresiones aritméticas.

Debido a que el objetivo principal del proyecto es desarrollar un ensamblador, en esta sección se exponen tres ejemplos de ensambladores ampliamente utilizados en la actualidad: GNU Assembler (GAS), TCCASM, y NASM. Además, también se estudiará el ensamblador actualmente utilizado por CREATOR. Tras esto, se realizará una comparativa entre estas herramientas y la solución propuesta.

2.2.1. GNU Assembler

GAS [9] es una familia de ensambladores desarrollados por el Proyecto GNU. Están escritos en C y se distribuyen bajo la licencia GNU GPLv3 [10]. Cada uno de estos en-

sambladores está diseñado para una ISA concreta, aunque comparten las características independientes de la máquina como la sintaxis, la mayoría de directivas de ensamblador, y el formato del fichero binario resultante [11]. GAS soporta más de 50 ISAs distintas, entre las que se incluyen RISC-V, MIPS, ARM, y x86. En la actualidad, GAS es uno de los ensamblador más utilizados [8]. GAS utiliza principalmente la sintaxis de AT&T, aunque también permite el uso de la sintaxis de Intel.

GAS está diseñado para ser utilizado principalmente como parte de la *toolchain* de GNU, siendo el *backend* por defecto de `gcc` [12]. Sin embargo, también se puede utilizar como un ensamblador independiente y es mayormente compatible con código escrito para otros ensambladores [11]. Debido a que está diseñado para ser utilizado por otras herramientas, sus mensajes de error son mejorables, ya que la prioridad es obtener un alto rendimiento y no generar buenos mensajes de error. Cabe destacar que GAS implementa estrategias de recuperación de errores para poder detectar múltiples errores en una misma compilación. Un ejemplo de esto se puede ver en la Figura 2.4. Para lograr este alto rendimiento, GAS procesa el código con una única pasada sobre el mismo.

```
$ riscv64-unknown-elf-as -march=rv32i -o test.o test.s
test.s: Assembler messages:
test.s:3: Error: illegal operands `addi t0,t0,t1'
test.s:5: Error: unrecognized opcode `unknown t0,t0,10'
```

Fig. 2.4. Mensaje de error del ensamblador de GNU

GAS tiene soporte para muchas funcionalidades avanzadas que lo vuelven un ensamblador muy flexible. Tiene soporte para etiquetas locales, evaluación de expresión aritméticas, cadenas en UTF-8 [13], compilación condicional, emisión de mensajes de error, definición de constantes, y macros variádicas (utilizando una cantidad variable de argumentos) y recursivas, entre otros. Cabe destacar que, aunque permite el uso de bigints, estos se convierten a enteros de 32 bits al usarlos en expresiones aritméticas. Además, no permite el uso de números decimales en las expresiones.

2.2.2. TCCASM

TCCASM es el ensamblador de TCC (Tiny C Compiler) [14]. TCC es un compilador de C diseñado para utilizar poca memoria y ser muy rápido. Es significativamente más rápido compilando que `gcc` [12], aunque también produce ficheros ejecutables más lentos [15]. Está escrito en C y se distribuye bajo la licencia GNU LGPLv2.1. TCC soporta principalmente x86 y x86-64, aunque también tiene soporte para ARM y RISC-V [16] [17] [18].

Para aumentar la velocidad de compilación, TCC genera directamente código máquina, sin pasar por ensamblador como hacen la mayoría de compiladores. TCC utiliza TCCASM únicamente para procesar ficheros en ensamblador y segmentos en ensamblador dentro del código C. Gracias a esto, TCC permite desactivar TCCASM para obtener

un ejecutable de TCC más pequeño. [16]

En cuanto al código ensamblador, TCCASM utiliza una sintaxis similar a GAS. Sin embargo, debido a su objetivo de utilizar poca memoria, soporta muchas menos funcionalidades que este. En particular, soporta únicamente la sintaxis de AT&T, enteros de 32 bits, muchas menos directivas de ensamblador, y no soporta compilación condicional, definición de constantes, ni macros. Además, sus expresiones aritméticas soportan menos operadores que GAS.

Con respecto a la detección de errores, TCCASM es más simple que GAS. TCCASM no implementa estrategias de recuperación de errores, por lo que aborta la compilación en el primer error detectado. Sus mensajes de error utilizan el mismo formato que los de GAS, aunque son un poco más simples. Por ejemplo, como se puede ver en la Figura 2.5, TCCASM no muestra los operandos de una instrucción cuando estos son incorrectos.

```
$ tcc test.s
test.s:5: error: bad operand with opcode 'addl'
```

Fig. 2.5. Mensaje de error de TCCASM

2.2.3. NASM

NASM (Netwide Assembler) [19] es un ensamblador multiplataforma de x86 y x86-64 diseñado para ser portable y modular. Está escrito en C, y se distribuye bajo la licencia BSD 2-Clause [20]. Tiene soporte para todas las extensiones de x86 actualmente conocidas, y permite utilizar muchos formatos distintos para el fichero binario resultante. Junto con GAS, es uno de los ensambladores más utilizados en Linux [8].

Con respecto al código ensamblador, NASM soporta etiquetas locales, evaluación de expresión aritméticas (con algunos operadores más que GAS) con números enteros, cadenas en UTF-8 [13], compilación condicional, definición de constantes, y uso de bucles para la generación de código. Además, utiliza la sintaxis de Intel. Cabe destacar su potente sistema de macros, que permite macros variádicas, sobrecarga (definición de múltiples macros con el mismo nombre) según el número de argumentos, etiquetas locales a la macro, y múltiples funcionalidades para procesar los argumentos.

NASM permite optimizar el código generado realizando múltiples pasadas sobre este. Sin embargo, para realizar esto, necesita conocer el tamaño de todos los datos e instrucciones durante la primera pasada. Debido a esto, no permite el uso de referencias hacia delante en situaciones en las que el valor de estas puede afectar al tamaño de los elementos generados. [21]

Con respecto a la detección de errores, NASM se encuentra en un punto intermedio entre GAS y TCCASM. Al igual que GAS, NASM implementa estrategias de recuperación de errores, lo que le permite detectar múltiples errores en una compilación. Sus

mensajes de error utilizan el mismo formato que los de GAS y TCCASM, pero tienen la misma simplicidad de los mensajes de TCCASM, como se puede ver en la Figura 2.6.

```
$ nasm test.s
test.s:5: error: invalid combination of opcode and operands
test.s:7: error: invalid combination of opcode and operands
```

Fig. 2.6. Mensaje de error de NASM

2.2.4. CREATOR

CREATOR [1] cuenta con un ensamblador propio para procesar el código ensamblador, convirtiéndolo a una representación interna que puede ejecutar. Este ensamblador está escrito en JS, y soporta únicamente funcionalidades básicas: instrucciones, etiquetas, y algunas directivas de ensamblador simples. Carece de muchas funcionalidades importantes como el uso de etiquetas en las directivas de datos. Esto es necesario, por ejemplo, para la creación del vector de interrupciones utilizado en un sistema operativo para gestionar los diversos tipos de interrupciones. Tampoco tiene soporte para evaluación de expresiones aritméticas, bigints, cadenas en UTF-8 [13], compilación condicional, definición de constantes, o macros.

Con respecto a sus mensajes de error, aunque generalmente son mejores que los de GAS, siguen siendo mejorables. Aunque incluyen la causa del error y la localización en el código que lo produjo (Figura 2.7), no incluyen información adicional relevante para el problema ni posibles soluciones del mismo. Estos mensajes también se podrían mejorar añadiendo algo de formato para resaltar las partes más importantes de los mismos. Cabe destacar que, en ciertos casos, estos mensajes no son capaces de mostrar el problema real debido a limitaciones del ensamblador.

```
Error at line 3 (10):
Register '10' not found

  2 main:
->3     add t0, t0, 10
  4

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

Fig. 2.7. Mensaje de error del ensamblador de CREATOR

El sistema a desarrollar se encargará de reemplazar a este ensamblador, añadiendo soporte para nuevas funcionalidades y mejorando sus mensajes de error.

2.2.5. Comparativa

Este proyecto tiene como objetivo combinar las características de los ensambladores analizados y adaptarlas para su uso en CREATOR. Para ello, se va a realizar una comparativa de las características de estos ensambladores, junto con la propuesta realizada.

Los resultados de la comparación se pueden ver en la Tabla 2.1. Las características a analizar son:

- **Lenguaje:** Lenguaje de programación en el que está escrito.
- **Licencia:** Licencia bajo la que se distribuye el *software*.
- **Ejecución web:** Capacidad de ser ejecutado en un navegador web.
- **Biblioteca:** Capacidad de ser embebido en otro programa y utilizado como una biblioteca.
- **Expresiones:** Capacidad de evaluar expresiones aritméticas durante el procesado del código.
- **Etiquetas como valores:** Capacidad de utilizar etiquetas como valores de las directivas de datos.
- **Bigints:** Capacidad de utilizar bigints en las expresiones aritméticas.
- **UTF-8:** Capacidad de utilizar cadenas de caracteres codificadas en UTF-8 [13].
- **Definición de constantes:** Capacidad de definir y utilizar constantes.
- **Macros:** Capacidad de definir y utilizar macros en el código.
- **Compilación condicional:** Capacidad utilizar expresiones para determinar si compilar un segmento del código o no.
- **Recuperación de errores:** Capacidad de recuperarse de un error en el código para continuar procesando el programa, potencialmente detectando múltiples errores en una misma compilación.

TABLA 2.1
COMPARATIVA DE LAS CARACTERÍSTICAS DE LOS ENSAMBLADORES

Ensamblador	GAS	TCCASM	NASM	CREATOR	Propuesta
Lenguaje	C	C	C	JS	Rust
Licencia	GPLv3	LGPLv2.1	BSD-2-Clause	LGPLv2.1	LGPLv2.1
Ejecución web				✓	✓
Biblioteca				✓	✓
Expresiones	✓*	✓*	✓*		✓
Etiquetas como valores	✓	✓	✓		✓
Bigints	✓				✓
UTF-8	✓	✓	✓		✓
Definición de constantes	✓		✓		✓**
Macros	✓		✓		✓**
Compilación condicional	✓		✓		✓**
Recuperación de errores	✓		✓		✓**

* Solo posible con números enteros de tamaño fijo

** Trabajo futuro

2.3. Técnicas de análisis sintáctico

Un compilador se compone principalmente de dos partes: análisis sintáctico y análisis semántico. El análisis sintáctico (*parsing*) se encarga de transformar el código a una representación intermedia, mientras que el análisis semántico se encarga de, utilizando la representación intermedia, verificar que el programa cumple la semántica del lenguaje y generar el código compilado. [22]

Al tratar con la semántica del lenguaje, no existen técnicas generales para realizar el análisis semántico. Sin embargo, para realizar el análisis sintáctico, existen múltiples técnicas basadas en el uso de gramáticas libres de contexto. Una gramática es una descripción formal de la sintaxis de un lenguaje, y está formada por: [22]

- Un conjunto de símbolos terminales (también conocidos como *tokens*), que representan los símbolos que pueden aparecer en las cadenas del lenguaje.
- Un conjunto de símbolos no terminales, que representan estructuras sintácticas del lenguaje (conjuntos de cadenas).
- Un símbolo no terminal denominado axioma, que representa todas las cadenas del lenguaje.
- Un conjunto de producciones, que representan las reglas según las cuales un símbolo no terminal se puede reemplazar por una secuencia de símbolos terminales y/o no terminales.

Una gramática genera cadenas de caracteres aplicando las producciones a una secuencia de símbolos, empezando por el axioma y continuando hasta que la secuencia solo contenga símbolos terminales. Este proceso se puede representar con un árbol sintáctico, en el cual los nodos representan símbolos (siendo la raíz el axioma, los nodos internos símbolos no terminales, y las hojas símbolos terminales), y las relaciones entre un nodo y sus hijos representan la aplicación de una producción determinada. La tarea de un analizador sintáctico (*parser*) consiste en determinar el árbol sintáctico que produce la cadena de entrada. [22]

Un posible problema en este proceso es la existencia de múltiples árboles sintácticos para una misma cadena. Como el árbol sintáctico determina el significado de una cadena, la presencia de múltiples posibilidades implica que su significado no está bien definido. Debido a esto, se dice que las gramáticas que pueden llevar a este caso son ambiguas. La mayoría de las técnicas de análisis sintáctico necesitan que la gramática no sea ambigua para poder utilizarse. Algunas técnicas permiten añadir reglas adicionales para seleccionar un único árbol en estos casos, resolviendo la ambigüedad, pero permitirla sin reglas adicionales tiene un coste de rendimiento. [22]

Las técnicas para realizar el análisis sintáctico se pueden agrupar en dos grandes familias: análisis descendente (*top-down*) y análisis ascendente (*bottom-up*). La diferencia entre estas familias está en cómo generan el árbol sintáctico. Un analizador descendente empieza por la raíz (axioma) y determina las producciones que se tienen que aplicar para obtener la cadena de entrada, realizando una búsqueda en profundidad. Un analizador ascendente, en cambio, empieza con los nodos hoja del árbol, y determina cómo agruparlos en símbolos no terminales hasta llegar a la raíz, siguiendo las producciones. [22]

2.3.1. Análisis descendente

Dentro de los analizadores descendentes, la técnica más general y utilizada es el análisis descendente recursivo, que consiste en un programa que modela la gramática mediante un conjunto de funciones mutuamente recursivas, cada una de las cuales representa un símbolo no terminal. Analizando la gramática, se puede crear un analizador predictivo en el cual el siguiente símbolo de la entrada a procesar permite determinar sin ambigüedades el flujo de control en cada función. Aunque según la teoría esta técnica es puramente recursiva, en la práctica también se utilizan bucles para aumentar el rendimiento y solucionar algunas de sus limitaciones. [22]

Gracias a su sencillez, esta técnica típicamente se implementa a mano, aunque también existen herramientas como ANTLR [23] que permiten generar un analizador automáticamente a partir de una gramática (conocido como generador de *parsers* [22]).

También existen otros métodos para crear analizadores descendentes recursivos a partir de una gramática, como los combinadores de *parsers*. Este método consiste en definir un analizador sintáctico como combinación de otros más simples, utilizando diversos

operadores que permiten realizar combinaciones muy expresivas [24]. En la actualidad, existen muchas bibliotecas que cuentan con estos operadores ya implementados, como Parsec [25], Chumsky [26], y pom [27].

La principal ventaja de crear un analizador descendente recursivo manualmente es su gran flexibilidad, ya que ofrece control total sobre el funcionamiento del analizador sintáctico. Esto permite utilizar información obtenida durante la ejecución del compilador para decidir cómo procesar la entrada, permitiendo un cierto grado de sensibilidad al contexto. Además, esto ayuda a generar buenos mensajes de error y a aplicar estrategias de recuperación de errores [28]. El funcionamiento de esta técnica también es más similar a la forma en la que se escribe el código, lo que facilita obtener información de bloques grandes de código aún en presencia de errores sintácticos [29]. Muchos de los lenguajes de programación utilizados en la actualidad, como C y C++, utilizan este método [30].

El principal problema de crear un analizador descendente recursivo a mano es su complejidad, requiriendo mucho tiempo para su desarrollo, además de que las modificaciones de la gramática requieren muchos cambios en la implementación. Los combinadores de *parsers*, en cambio, facilitan mucho estos procesos, aunque sacrifican un poco de rendimiento para lograrlo. Esto hace que los combinadores de *parsers* sean un método ideal para realizar prototipos, durante los cuales la gramática puede variar, mientas que crear un analizador manualmente es una mejor opción para un producto final cuando el rendimiento se vuelve un problema.

Estas técnicas, en general, son muy efectivas. Sin embargo, para gramáticas que cumplen ciertas propiedades (gramáticas de precedencia de operadores) típicas de expresiones aritméticas existen técnicas más eficientes, especialmente cuando cuentan con muchos niveles de precedencia. Estas técnicas se pueden agrupar en los *parsers* de precedencia de operadores [31]. Un ejemplo de estas técnicas es el análisis Pratt [32] [33], que representa los niveles de precedencia con funciones para comparar operadores. Gracias a esto, se puede evitar representar los niveles de precedencia en la propia gramática, permitiendo utilizar una representación más natural de la gramática y obteniendo un *parser* más eficiente.

2.3.2. Análisis ascendente

Dentro de los analizadores ascendentes, existen muchas técnicas distintas. La mayoría de estas técnicas pertenecen a la familia LR, como SLR, LALR, o GLR. Todas las técnicas en esta subfamilia utilizan un algoritmo similar: cuentan con una pila inicialmente vacía y un autómata finito (representado por una tabla) que, según su estado, determina su siguiente estado y la operación a aplicar a la entrada y pila. Esta operación puede ser añadir un símbolo de la entrada a la cima de la pila o agrupar los últimos símbolos de la pila en un símbolo no terminal según una producción determinada. [22]

La diferencia entre estos métodos está en cómo se construye el autómata finito y el

conjunto de gramáticas que pueden representar, siendo SLR un método simple que puede representar pocas gramáticas, mientras que LALR puede representar más gramáticas, pero es más complejo de construir. [22] Por otro lado, GLR es una generalización de las técnicas LR para permitir procesar gramáticas ambiguas, obteniendo como resultado todos los posibles árboles sintácticos que darían lugar a la cadena de entrada [34].

Debido a la gran cantidad de estados que puede tener el autómata, estos analizadores típicamente se construyen automáticamente a partir de la gramática con un generador de parsers [22]. Existen muchas herramientas de este tipo como Lex y yacc [35] (basado en LALR), sus versiones modernas de GNU Flex y bison [36] (basado en LALR por defecto, aunque pueden utilizar GLR), o Tree-sitter [37] (basado en GLR).

2.4. Mensajes de error

En la actualidad, los mensajes de error de los compilador tienen muchas formas y calidades. Un compilador con buenos mensajes de error puede ayudar mucho a solucionar los problemas más rápidamente, ya que permite obtener toda la información necesaria para solucionar el problema del propio mensaje de error.

Un ejemplo de malos mensajes de error es el compilador de C#, cuyos mensajes de error se pueden ver en la Figura 2.8. Como se puede ver, estos mensajes contienen muy poca información sobre el problema. Esto dificulta corregir los errores, especialmente para alguien que está intentando aprender el lenguaje.

```
Versión de MSBuild 17.8.22+bfbb05667 para .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
/home/alvaro/Documentos/temp/csharp/Program.cs(23,28): error CS0118: 'vec' es variable pero se usa como tipo [/home/a
lvaro/Documentos/temp/csharp/csharp.csproj]

ERROR al compilar.

/home/alvaro/Documentos/temp/csharp/Program.cs(23,28): error CS0118: 'vec' es variable pero se usa como tipo [/home/a
lvaro/Documentos/temp/csharp/csharp.csproj]
    0 Advertencia(s)
    1 Errores

Tiempo transcurrido 00:00:04.36
```

Fig. 2.8. Mensaje de error del compilador de C#

En el otro extremo, demasiada información también puede convertirse en un problema si esta llega a ser redundante o irrelevante (“ruido”). Una gran cantidad de información en el mensaje incrementa la carga cognitiva necesaria para comprenderlo. Si una gran parte de esta información es “ruido”, se dificulta sin necesidad la comprensión del mensaje, y aumenta el tiempo necesario para extraer la información realmente relevante. Un ejemplo de este problema son los errores de *templates* de C++, que pueden llegar a contener varias centenas de líneas para un único error.

Esto se puede mejorar si se añade resaltado con colores u otros medios a las partes más relevantes. En el caso de los colores, se puede utilizar un código de colores para indicar visualmente el tipo de la información resaltada, facilitando el filtrado de la información.

Por ejemplo, es común utilizar el color rojo para errores, el amarillo para avisos, y el verde o azul para posibles soluciones o información adicional.

Actualmente, el compilador de Rust [38] es uno de los compiladores que mejores mensajes de error genera. La Figura 2.9 muestra un ejemplo de estos mensajes, obtenido en un caso de uso real. Como se puede observar, este mensaje cuenta con varios componentes importantes:

- Un mensaje de error con una explicación clara del error.
- La localización en el código que ha producido el fallo.
- Información adicional sobre el error y su causa, resaltada sobre el propio código.
- Un mensaje de ayuda que indica una posible solución del error, incluyendo una explicación de esta y las modificaciones necesarias resaltadas sobre el código.
- Una forma de acceder a una explicación detallada sobre el problema.

Además de esto, la información está resaltada con un código de colores y se utilizan diversos indicadores. También se deja cierto espacio en blanco para separar las diferentes partes. Esto ayuda a extraer rápidamente la información más importante para solucionar el problema y filtrar la que no se necesita.

```
   Compiling tfg_test v0.1.0 (/home/alvaro/Documentos/temp/tfg_test)
error[E0373]: closure may outlive the current function, but it borrows `name`, which is owned by the current function
  --> src/main.rs:13:30
   |
13 |         self.0.iter().filter(|&inst| inst.name == name)
   |                              -------               ---- `name` is borrowed here
   |                              |
   |                              may outlive borrowed value `name`
   |
note: closure is returned here
  --> src/main.rs:13:9
   |
13 |         self.0.iter().filter(|&inst| inst.name == name)
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `name` (and any other referenced variables), use the `move` keyword
   |
13 |         self.0.iter().filter(move |&inst| inst.name == name)
   |                              ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `tfg_test` (bin "tfg_test") due to 1 previous error
```

Fig. 2.9. Mensaje de error del compilador de Rust

2.5. Lenguajes de programación

Actualmente, las herramientas utilizadas durante el desarrollo de *software* para los diversos lenguajes, como compiladores, formateadores de código, o linters, se implementan en muchos lenguajes de programación distintos.

Para los lenguajes de bajo nivel, como C/C++ o ensamblador, típicamente las herramientas se implementan en lenguajes de bajo nivel, comúnmente el mismo para el que

se utilizan. Esto es debido a su mayor rendimiento, ya que el *software* realizado en estos lenguajes suele tener grandes bases de código. La mayoría de herramientas de estos lenguajes, como `gcc` [12] y la *suite* de `clang` (`clang`, `clang-tidy` y `clang-format`) [39], utilizan C/C++. Una excepción importante a esto es `conan` [40], el gestor de paquetes de C/C++, que está escrito en un lenguaje de alto nivel (Python).

En los lenguajes de alto nivel, como Python o TypeScript (TS), las herramientas tradicionalmente se han implementado en lenguajes de alto nivel para facilitar su desarrollo. Este es el caso de `pylint` [41], `black` [42] o el compilador de TS [43]. Sin embargo, recientemente muchas de estas herramientas se están reescribiendo en lenguajes de más bajo nivel como Rust o Go debido a la mejora significativa de rendimiento que ofrecen estos lenguajes [44]. Algunos ejemplos de herramientas que han tenido éxito con esto son `ruff` [45] y `uv` [46].

CAPÍTULO 3

ANÁLISIS

Este capítulo describirá la solución propuesta y está dividido en tres secciones. La primera (Sección 3.1, *Descripción del proyecto*), hará una breve descripción del proyecto. La segunda (Sección 3.2, *Requisitos*), describirá los requisitos del sistema a desarrollar. Por último, la tercera (Sección 3.3, *Casos de uso*), especificará los diferentes casos de uso del sistema.

3.1. Descripción del proyecto

El objetivo de este proyecto es rehacer el compilador utilizado por CREATOR, para solucionar los problemas que presenta el compilador utilizado actualmente. Este compilador será utilizado principalmente por estudiantes para programar en lenguaje ensamblador. Debido a esto, es fundamental que el compilador genere buenos mensajes de error que orienten a los estudiantes en su aprendizaje de este lenguaje.

El compilador actual tiene múltiples problemas y carencias, y está poco documentado. Esto lo vuelve difícil de modificar para corregir los problemas y añadir nuevas funcionalidades. Además, solo permite el uso de las funcionalidades más básicas, lo cual se vuelve un problema para realizar tareas más complejas, tiene problemas de rendimiento al aumentar el tamaño de los programas, y sus mensajes de error se podrían mejorar.

Debido a esto, se propone crear un nuevo compilador con una arquitectura robusta que le permita ser flexible para añadir nuevas funcionalidades y corregir errores, solucionando los problemas del compilador actual. Este nuevo compilador también mejorará los mensajes de error para ayudar a los estudiantes en su aprendizaje del lenguaje. Además, añadirá nuevas funcionalidades no soportadas en el compilador actual, que facilitarán el uso de CREATOR para tareas más complejas y permitirán su uso como entorno de pruebas para nuevas arquitecturas.

3.2. Requisitos

Esta sección contiene la especificación de los requisitos del sistema a desarrollar. Para esta especificación de requisitos se han seguido las prácticas recomendadas por IEEE [47]. Estas prácticas indican que una buena especificación de requisitos debe explicar la funcionalidad del software, los requisitos de rendimiento, las interfaces externas, otros atributos, y restricciones de diseño.

Además, la especificación de requisitos debe ser:

- **No ambigua:** Los requisitos tienen una única interpretación.
- **Completa:** Incluye todos los requisitos relevantes.
- **Verificable:** Existe un proceso finito y no costoso que permite comprobar que el sistema cumple con todos los requisitos.
- **Consistente:** No existe ningún conjunto de requisitos contradictorios entre sí.
- **Modificable:** La estructura y estilo de los requisitos permite que cualquier cambio necesario se haga de forma fácil, completa, y consistente.
- **Trazable:** El origen de cada requisito es claro, y facilita la referencia de cada requisito en otras etapas.
- **Clasificada:** Los requisitos deben estar clasificados según su importancia y estabilidad.

Para realizar la especificación de requisitos, se parte de los requisitos de usuario (Subsección 3.2.1, *Requisitos de usuario*), que contienen una especificación informal de las necesidades del cliente y lo que este espera del producto. A partir de estos, se crean los requisitos de software (Subsección 3.2.2, *Requisitos de software*). Estos requisitos guían el proceso de diseño, aportando información sobre las funcionalidades del sistema y cualquier otra característica adicional.

3.2.1. Requisitos de usuario

Esta sección contiene el listado detallado de los requisitos de usuario. Estos requisitos explican la funcionalidad principal del sistema y las restricciones que este debe cumplir para ser aceptado por el cliente.

Estos requisitos se pueden dividir en dos tipos:

- **Capacidad:** Representa una funcionalidad que el sistema debe tener.
- **Restricción:** Representa una condición que el sistema debe cumplir.

Cada requisito de usuario se identifica con un ID que sigue el formato *RU-YY-XX*, donde *YY* identifica el tipo del requisito, ya sea capacidad (*CA*) o restricción (*RE*); y *XX* identifica el número secuencial del requisito dentro de su tipo, empezando en *01*.

La tabla 3.1 contiene la plantilla utilizada para la especificación de los requisitos, incluyendo la descripción de cada atributo. Tras esta, se incluyen las tablas con la especificación de cada uno de los requisitos de usuario.

TABLA 3.1
PLANTILLA DE REQUISITO DE USUARIO

RU-YY-XX	
Descripción	Especificación del requisito en un lenguaje claro, conciso y no ambiguo.
Necesidad	Prioridad del requisito para el usuario (*Esencial*, *Conveniente* u *Opcional*).
Prioridad	Prioridad del requisito para el desarrollador (*Alta*, *Media* o *Baja*).
Estabilidad	Indica si el requisito se modifica durante el desarrollo (*No cambia*, *Cambiante* o *Muy inestable*).
Verificabilidad	Capacidad de comprobar la validez del requisito (*Alta*, *Media* o *Baja*).

TABLA 3.2
REQUISITO RU-CA-01

RU-CA-01	
Descripción	El sistema debe soportar todas las funcionalidades soportadas en el compilador actual.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Media

TABLA 3.3
REQUISITO RU-CA-02

RU-CA-02	
Descripción	El sistema debe soportar cadenas de caracteres codificadas en UTF-8 [13].
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.4
REQUISITO RU-CA-03

RU-CA-03	
Descripción	El sistema debe soportar secuencias de escape en las cadenas de caracteres y caracteres literales.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Cambiante
Verificabilidad	Media

TABLA 3.5
REQUISITO RU-CA-04

RU-CA-04	
Descripción	El sistema debe permitir especificar la ISA a utilizar.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.6
REQUISITO RU-CA-05

RU-CA-05	
Descripción	El sistema debe permitir utilizar comentarios de línea y comentarios multilínea.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.7
REQUISITO RU-CA-06

RU-CA-06	
Descripción	El sistema debe permitir especificar la cadena utilizada como prefijo de los comentarios de línea.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.8
REQUISITO RU-CA-07

RU-CA-07	
Descripción	El sistema debe permitir utilizar cualquier cantidad de etiquetas en una sentencia.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.9
REQUISITO RU-CA-08

RU-CA-08	
Descripción	El sistema debe permitir referenciar etiquetas definidas en el código ensamblador después de su uso.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media

TABLA 3.10
REQUISITO RU-CA-09

RU-CA-09	
Descripción	El sistema debe permitir utilizar etiquetas como valores de las directivas de datos.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.11
REQUISITO RU-CA-10

RU-CA-10	
Descripción	El sistema debe permitir evaluar y utilizar expresiones con valores constantes.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Cambiante
Verificabilidad	Alta

TABLA 3.12
REQUISITO RU-CA-11

RU-CA-11	
Descripción	El sistema debe permitir definir múltiples instrucciones con el mismo nombre, y utilizar su sintaxis y el tamaño de los argumentos (en bits) para seleccionar la correcta durante la compilación de un programa ensamblador.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.13
REQUISITO RU-CA-12

RU-CA-12	
Descripción	El sistema debe detectar errores sintácticos y semánticos en el código ensamblador.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media

TABLA 3.14
REQUISITO RU-CA-13

RU-CA-13	
Descripción	El sistema debe tener buenos mensajes de error.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Baja

TABLA 3.15
REQUISITO RU-CA-14

RU-CA-14	
Descripción	El sistema debe permitir utilizar números enteros de cualquier tamaño para las direcciones de memoria y valores de las expresiones.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.16
REQUISITO RU-RE-01

RU-RE-01	
Descripción	El sistema se tiene que integrar con el simulador CREATOR.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.17
REQUISITO RU-RE-02

RU-RE-02	
Descripción	El sistema se tiene que poder ejecutar en los entornos soportados actualmente por CREATOR (navegadores y Node.js).
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.18
REQUISITO RU-RE-03

RU-RE-03	
Descripción	El sistema tiene que ser Free and Open Source Software (FOSS).
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta

TABLA 3.19
REQUISITO RU-RE-04

RU-RE-04	
Descripción	El sistema tiene que utilizar la sintaxis del ensamblador GAS [11].
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media

TABLA 3.20
REQUISITO RU-RE-05

RU-RE-05	
Descripción	El sistema tiene que ser rápido.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Muy inestable
Verificabilidad	Baja

3.2.2. Requisitos de software

Esta sección contiene el listado detallado de los requisitos de software. Estos requisitos se han creado a partir de los requisitos de usuario descritos en la sección anterior, y definen las características del sistema.

Estos requisitos se pueden dividir en dos tipos:

- **Funcional:** Define las funcionalidades y características del software.
- **No funcional:** Define una condición que el software debe cumplir.

Cada requisito de software se identifica con un ID que sigue el formato *RS-YY-XX*, donde *YY* identifica el tipo del requisito, ya sea funcional (*FN*) o no funcional (*NF*); y *XX* identifica el número secuencial del requisito dentro de su tipo, empezando en *01*.

La tabla 3.21 contiene la plantilla utilizada para la especificación de los requisitos, incluyendo la descripción de cada atributo. Tras esta, se incluyen las tablas con la especificación de cada uno de los requisitos de software.

TABLA 3.21
PLANTILLA DE REQUISITO DE SOFTWARE

RS-YY-XX	
Descripción	Especificación del requisito en un lenguaje claro, conciso y no ambiguo.
Necesidad	Prioridad del requisito para el usuario (*Esencial*, *Conveniente* u *Opcional*).
Prioridad	Prioridad del requisito para el desarrollador (*Alta*, *Media* o *Baja*).
Estabilidad	Indica si el requisito se modifica durante el desarrollo (*No cambia*, *Cambiante* o *Muy inestable*).
Verificabilidad	Capacidad de comprobar la validez del requisito (*Alta*, *Media* o *Baja*).
Origen	Referencia a los requisitos de usuario que dieron lugar al requisito.

TABLA 3.22
REQUISITO RS-FN-01

RS-FN-01	
Descripción	El sistema debe permitir compilar instrucciones.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.23
REQUISITO RS-FN-02

RS-FN-02	
Descripción	El sistema debe permitir compilar pseudo-instrucciones.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.24
REQUISITO RS-FN-03

RS-FN-03	
Descripción	El sistema debe soportar las siguientes directivas de ensamblador: cambiar el tipo de la sección actual, declarar etiquetas globales, directivas de datos, e ignorar directiva.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.25
REQUISITO RS-FN-04

RS-FN-04	
Descripción	El sistema debe soportar los siguientes tipos de secciones: datos e instrucciones.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.26
REQUISITO RS-FN-05

RS-FN-05	
Descripción	El sistema debe soportar las siguientes directivas de datos: reservar espacio, cadenas de caracteres, números enteros, números decimales, y alineamiento de la memoria de datos.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.27
REQUISITO RS-FN-06

RS-FN-06	
Descripción	El sistema debe soportar directivas de datos de cadenas de caracteres terminadas y no terminadas en un byte nulo.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.28
REQUISITO RS-FN-07

RS-FN-07	
Descripción	El sistema debe soportar directivas de datos de números enteros con los siguientes tamaños: byte, media palabra, palabra, y doble palabra.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.29
REQUISITO RS-FN-08

RS-FN-08	
Descripción	El sistema debe soportar directivas de datos de números decimales con el formato IEEE 745 [48] de precisión simple (binary32) y doble (binary64).
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.30
REQUISITO RS-FN-09

RS-FN-09	
Descripción	El sistema debe soportar directivas de datos de alineamiento a bytes y a potencias de 2.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.31
REQUISITO RS-FN-10

RS-FN-10	
Descripción	El sistema debe soportar bibliotecas: debe permitir reservar espacio para instrucciones de un programa compilado previamente, y utilizar las etiquetas declaradas como globales definidas en este.
Necesidad	Esencial
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-01

TABLA 3.32
REQUISITO RS-FN-11

RS-FN-11	
Descripción	El sistema debe soportar cadenas de caracteres codificadas en UTF-8 [13].
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-02

TABLA 3.33
REQUISITO RS-FN-12

RS-FN-12	
Descripción	El sistema debe soportar las siguientes secuencias de escape en las cadenas de caracteres y caracteres literales: `\\`, `\"`, `\'`, `\a` (alerta), `\b` (backspace), `\e` (escape), `\f` (salto de página), `\n`, `\r`, `\t`, y `\0`.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Cambiante
Verificabilidad	Alta
Origen	RU-CA-03

TABLA 3.34
REQUISITO RS-FN-13

RS-FN-13	
Descripción	El sistema debe permitir especificar la ISA a utilizar.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-04

TABLA 3.35
REQUISITO RS-FN-14

RS-FN-14	
Descripción	El sistema debe permitir utilizar comentarios de línea y comentarios multilínea.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-05

TABLA 3.36
REQUISITO RS-FN-15

RS-FN-15	
Descripción	El sistema debe permitir especificar la cadena utilizada como prefijo de los comentarios de línea.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-06

TABLA 3.37
REQUISITO RS-FN-16

RS-FN-16	
Descripción	El sistema debe permitir utilizar cualquier cantidad de etiquetas en una sentencia.
Necesidad	Conveniente
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-07

TABLA 3.38
REQUISITO RS-FN-17

RS-FN-17	
Descripción	El sistema debe permitir referenciar etiquetas definidas en el código ensamblador después de su uso.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media
Origen	RU-CA-08

TABLA 3.39
REQUISITO RS-FN-18

RS-FN-18	
Descripción	El sistema debe permitir evaluar expresiones con los siguientes tipos de valores constantes: números enteros, números decimales, caracteres literales (utilizando su representación Unicode [13]) y etiquetas.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-09, RU-CA-10

TABLA 3.40
REQUISITO RS-FN-19

RS-FN-19	
Descripción	El sistema debe permitir utilizar paréntesis y los siguientes operadores en las expresiones: ▪ Unarios: +, -, y ~ (complemento). ▪ Binarios aritméticos: +, -, *, /, y %. ▪ Binarios bit a bit: | (OR), & (AND), y ^ (XOR).
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Cambiante
Verificabilidad	Alta
Origen	RU-CA-10

TABLA 3.41
REQUISITO RS-FN-20

RS-FN-20	
Descripción	El sistema debe permitir utilizar expresiones como valores inmediatos de las instrucciones y pseudo-instrucciones.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-10

TABLA 3.42
REQUISITO RS-FN-21

RS-FN-21	
Descripción	El sistema debe permitir utilizar expresiones como valores de las directivas de datos de reservar espacio, números enteros y decimales, y alineamiento.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-10

TABLA 3.43
REQUISITO RS-FN-22

RS-FN-22	
Descripción	El sistema debe permitir definir múltiples instrucciones con el mismo nombre, y utilizar su sintaxis y el tamaño de los argumentos (en bits) para seleccionar la correcta durante la compilación de un programa ensamblador.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-11

TABLA 3.44
REQUISITO RS-FN-23

RS-FN-23	
Descripción	El sistema debe detectar errores de sintaxis en el código ensamblador.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Media
Origen	RU-CA-12

TABLA 3.45
REQUISITO RS-FN-24

RS-FN-24	
Descripción	El sistema debe detectar errores semánticos en el código ensamblador.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media
Origen	RU-CA-12

TABLA 3.46
REQUISITO RS-FN-25

RS-FN-25	
Descripción	El sistema debe generar mensajes de error que contengan, al menos: posición exacta del error (línea y columna), mensaje de error, segmento de código con el error resaltado, e información adicional relevante como motivo del error u posible solución.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Alta
Origen	RU-CA-13

TABLA 3.47
REQUISITO RS-FN-26

RS-FN-26	
Descripción	Los mensajes de error del sistema que se refieran a identificadores no definidos deben contener el identificador definido más similar según la distancia de edición.
Necesidad	Opcional
Prioridad	Baja
Estabilidad	Cambiante
Verificabilidad	Alta
Origen	RU-CA-13

TABLA 3.48
REQUISITO RS-FN-27

RS-FN-27	
Descripción	El sistema debe resaltar las diferentes partes de los mensajes de error con un código de colores.
Necesidad	Opcional
Prioridad	Baja
Estabilidad	No cambia
Verificabilidad	Media
Origen	RU-CA-13

TABLA 3.49
REQUISITO RS-FN-28

RS-FN-28	
Descripción	El sistema debe permitir utilizar números enteros de cualquier tamaño para las direcciones de memoria y valores de las expresiones.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-CA-14

TABLA 3.50
REQUISITO RS-NF-01

RS-NF-01	
Descripción	El formato de definición de la ISA debe ser el utilizado actualmente en CREATOR, serializado en formato JSON [49].
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-01

TABLA 3.51
REQUISITO RS-NF-02

RS-NF-02	
Descripción	La definición de la sintaxis de las instrucciones debe estar contenida en la ISA.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-01

TABLA 3.52
REQUISITO RS-NF-03

RS-NF-03	
Descripción	La Application Programming Interface (API) de entrada y salida del sistema debe ser la utilizada actualmente en CREATOR.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-01

TABLA 3.53
REQUISITO RS-NF-04

RS-NF-04	
Descripción	El sistema debe tener una API para JS.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-01

TABLA 3.54
REQUISITO RS-NF-05

RS-NF-05	
Descripción	El sistema se tiene que poder ejecutar en Google Chrome 70+.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-02

TABLA 3.55
REQUISITO RS-NF-06

RS-NF-06	
Descripción	El sistema se tiene que poder ejecutar en Mozilla Firefox 60+.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-02

TABLA 3.56
REQUISITO RS-NF-07

RS-NF-07	
Descripción	El sistema se tiene que poder ejecutar en Safari 10+.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-02

TABLA 3.57
REQUISITO RS-NF-08

RS-NF-08	
Descripción	El sistema se tiene que poder ejecutar en Node.js.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-02

TABLA 3.58
REQUISITO RS-NF-09

RS-NF-09	
Descripción	El sistema tiene que utilizar una licencia FOSS según las definiciones de la Free Software Foundation (FSF) [50] y Open Source Initiative (OSI) [51].
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-03

TABLA 3.59
REQUISITO RS-NF-10

RS-NF-10	
Descripción	El código fuente del sistema tiene que ser público.
Necesidad	Esencial
Prioridad	Media
Estabilidad	No cambia
Verificabilidad	Alta
Origen	RU-RE-03

TABLA 3.60
REQUISITO RS-NF-11

RS-NF-11	
Descripción	El sistema tiene que utilizar la sintaxis del ensamblador GAS [11] para los programas compilados.
Necesidad	Esencial
Prioridad	Alta
Estabilidad	Cambiante
Verificabilidad	Media
Origen	RU-RE-04

TABLA 3.61
REQUISITO RS-NF-12

RS-NF-12	
Descripción	El sistema tiene que poder compilar 1000 instrucciones por segundo.
Necesidad	Conveniente
Prioridad	Media
Estabilidad	Cambiante
Verificabilidad	Alta
Origen	RU-RE-05

3.2.3. Trazabilidad

La matriz de trazabilidad permite verificar que todos los requisitos de usuario están cubiertos por al menos un requisito de software. Como se puede ver, todos los requisitos de capacidad están cubiertos por los requisitos funcionales (Tabla 3.62), y todos los requisitos de restricción están cubiertos por los requisitos no funcionales (Tabla 3.63)

TABLA 3.62

TRAZABILIDAD ENTRE LOS REQUISITOS DE CAPACIDAD Y LOS REQUISITOS FUNCIONALES

	RU-CA-01	RU-CA-02	RU-CA-03	RU-CA-04	RU-CA-05	RU-CA-06	RU-CA-07	RU-CA-08	RU-CA-09	RU-CA-10	RU-CA-11	RU-CA-12	RU-CA-13	RU-CA-14
RS-FN-01	•													
RS-FN-02	•													
RS-FN-03	•													
RS-FN-04	•													
RS-FN-05	•													
RS-FN-06	•													
RS-FN-07	•													
RS-FN-08	•													
RS-FN-09	•													
RS-FN-10	•													
RS-FN-11		•												
RS-FN-12			•											
RS-FN-13				•										
RS-FN-14					•									
RS-FN-15						•								
RS-FN-16							•							
RS-FN-17								•						
RS-FN-18									•	•				
RS-FN-19										•				
RS-FN-20										•				
RS-FN-21										•				
RS-FN-22											•			
RS-FN-23												•		
RS-FN-24												•		
RS-FN-25													•	
RS-FN-26													•	
RS-FN-27													•	
RS-FN-28														•

TABLA 3.63
TRAZABILIDAD ENTRE LOS REQUISITOS DE RESTRICCIÓN Y LOS REQUISITOS NO FUNCIONALES

	RU-RE-01	RU-RE-02	RU-RE-03	RU-RE-04	RU-RE-05
RS-NF-01	•				
RS-NF-02	•				
RS-NF-03	•				
RS-NF-04	•				
RS-NF-05		•			
RS-NF-06		•			
RS-NF-07		•			
RS-NF-08		•			
RS-NF-09			•		
RS-NF-10			•		
RS-NF-11				•	
RS-NF-12					•

3.3. Casos de uso

El modelo de casos de uso UML [52], mostrado en la Figura 3.1, representa las acciones que el usuario puede realizar con el sistema. Cabe destacar que el usuario del sistema será el simulador CREATOR, ya que el sistema a desarrollar será un componente utilizado internamente por CREATOR para llevar a cabo su funcionalidad.

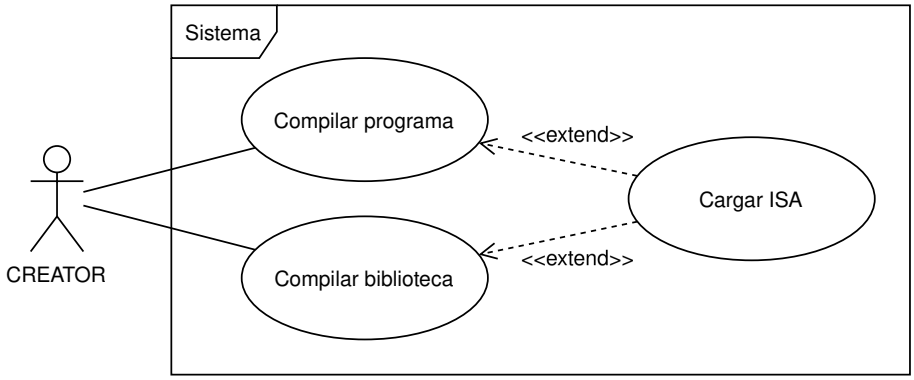

Fig. 3.1. Modelo de casos de uso

Cada caso de uso se identifica con un ID que sigue el formato *CU-XX*, donde *XX* identifica el número secuencial del caso de uso, empezando en *01*.

La tabla 3.64 contiene la plantilla utilizada para la especificación de los casos de uso, incluyendo la descripción de cada atributo. Tras esta, se incluyen las tablas con la especificación de cada uno de los casos de uso.

TABLA 3.64
PLANTILLA DE CASO DE USO

CU-XX	
Nombre	Descripción breve del caso de uso.
Actores	Agente externo que ejecuta el caso de uso.
Objetivo	Propósito del caso de uso.
Descripción	Pasos que debe seguir el actor para ejecutar el caso de uso.
Pre-condición	Condiciones previas que se deben cumplir para ejecutar el caso de uso.
Post-condición	Condiciones que se deben cumplir después de ejecutar el caso de uso.

TABLA 3.65
CASO DE USO CU-01

CU-01	
Nombre	Cargar ISA
Actores	CREATOR
Objetivo	Cargar la ISA a utilizar durante la compilación de un código ensamblador.
Descripción	1. CREATOR entrega una cadena de caracteres con la ISA a la API de cargar ISA. 2. Se obtiene el objeto que representa la ISA cargada.
Pre-condición	CREATOR debe tener la ISA correctamente definida en una cadena de caracteres.
Post-condición	La ISA está cargada y disponible para realizar la compilación.

TABLA 3.66
CASO DE USO CU-02

CU-02	
Nombre	Compilar programa
Actores	CREATOR
Objetivo	Compilar un programa para su posterior ejecución en CREATOR.
Descripción	1. CREATOR carga la ISA si no está ya cargada. 2. CREATOR entrega una cadena de caracteres con el programa a la API de compilar programa, junto con el objeto que representa la ISA. 3. Se obtienen los datos correspondientes al programa compilado.
Pre-condición	CREATOR debe tener la ISA correctamente definida en una cadena de caracteres, y el código a compilar.
Post-condición	El programa está compilado y disponible para su ejecución en CREATOR.

TABLA 3.67
CASO DE USO CU-03

CU-03	
Nombre	Compilar biblioteca
Actores	CREATOR
Objetivo	Compilar una biblioteca para su posterior uso en un programa.
Descripción	1. CREATOR carga la ISA si no está ya cargada. 2. CREATOR entrega una cadena de caracteres con el programa a la API de compilar biblioteca, junto con el objeto que representa la ISA. 3. Se obtienen los datos correspondientes a la biblioteca compilada.
Pre-condición	CREATOR debe tener la ISA correctamente definida en una cadena de caracteres, y el código a compilar.
Post-condición	La biblioteca está compilada y disponible para su uso en un programa.

CAPÍTULO 4

DISEÑO

Este capítulo describirá el diseño del sistema y está dividido en dos secciones. La primera (Sección 4.1, *Estudio de la solución final*), hará un estudio de las alternativas posibles para el desarrollo, llegando a una solución que cumple con los requisitos establecidos en la Sección 3.2, *Requisitos*. La segunda (Sección 4.2, *Arquitectura del compilador*), describirá la arquitectura del compilador, incluyendo sus componentes y decisiones de diseño.

4.1. Estudio de la solución final

Para realizar el estudio de la solución, se ha tenido en cuenta la elicitación de requisitos (Sección 3.2, *Requisitos*), ya que esto determina las capacidades y restricciones del sistema y, por lo tanto, el diseño dependerá de ellos.

4.1.1. Lenguaje de programación

Según los requisitos RS-NF-05 a RS-NF-04, el sistema debe poder ejecutarse en navegador y Node.js, y tener una API para JS. Esto deja dos alternativas principales: utilizar JS o WebAssembly (Wasm).

JS es el lenguaje estándar utilizado por los navegadores y Node.js, por lo que es la primera opción para el desarrollo del sistema. El principal problema de esta opción es el rendimiento: JS es un lenguaje interpretado, lo cual lo hace significativamente más lento que otros lenguajes compilados, y, debido a esto, proyectos que fueron originalmente desarrollados en JS se están moviendo a otros lenguajes [44]. La mayor velocidad de los lenguajes compilados es debido a que pueden generar código que se ejecuta directamente en el procesador (ejecución nativa), sin el sobrecoste de una máquina virtual o intérprete.

El Requisito RS-NF-12 indica que el sistema debe ser rápido, por lo que sería preferible utilizar un lenguaje que soporte la ejecución nativa.

Otro problema de JS es su sistema de tipos. JS utiliza un sistema de tipos dinámico y débil, con muchas conversiones implícitas de tipos [53]. Esto dificulta razonar sobre el código, ya que no garantiza que los tipos de datos serán los esperados, y permite que aparezcan errores muy difíciles de encontrar. Además, las conversiones implícitas de tipos permiten que los errores se propaguen antes de ser detectados, dificultando encontrar su origen. Debido a esto, sería preferible utilizar un lenguaje con un sistema de tipos estático y fuerte. TS [54] ha surgido como una alternativa a JS que soluciona este problema, pero al ser compilado a JS no soluciona el problema de rendimiento.

Wasm [55] es un nuevo estándar soportado por los navegadores y Node.js que permite la ejecución de programas compilados de forma casi nativa, obteniendo un rendimiento significativamente mejor que JS. Muchos lenguajes como C [56], C++ [57], o Rust [38] soportan la compilación a Wasm, posibilitando su uso para desarrollar el sistema. Todos estos lenguajes tienen un rendimiento prácticamente idéntico, por lo que se necesitan analizar otras características para elegir la mejor opción.

En la actualidad, la mayoría de las vulnerabilidades de seguridad en software están causadas por problemas en el uso y gestión de memoria [58] [59]. Utilizar un lenguaje que ofrezca seguridad de memoria permite evitar estos fallos, y actualmente se está instando por la adopción de este tipo de lenguaje [60] [61]. De los anteriormente mencionados, el único que cumple con esta característica es Rust [62].

Otra característica importante es la capacidad de generación de *bindings*. Estos son necesarios para permitir que el código JS pueda utilizar un módulo Wasm como una biblioteca, ofreciendo una API para JS como pide el Requisito RS-NF-04. Existen herramientas como Emscripten (C/C++) [63] o Wasm-pack (Rust) [64] que realizan esto para todos los lenguajes mencionados anteriormente. Cabe destacar que Emscripten es algo más manual, ya que, por ejemplo, requiere redefinir los tipos de los argumentos y valor de retorno para cada función que se quiera ejecutar desde JS. Wasm-pack, en cambio, es capaz de generar automáticamente todo el código necesario para poder utilizar las funciones definidas en el módulo Wasm como si se tratasen de funciones nativas de JS.

La Tabla 4.1 resume las características analizadas. Teniendo en cuenta todas estas características, se ha elegido Rust como lenguaje a utilizar para el desarrollo del sistema.

TABLA 4.1
COMPARACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

Lenguaje	JS	TS	C	C++	Rust
Ejecución nativa			✓	✓	✓
Alto rendimiento			✓	✓	✓
Seguridad de memoria	✓	✓			✓
Buena interoperabilidad con JS	✓	✓			✓
Sistema de tipos estático		✓	✓	✓	✓
Sistema de tipos fuerte		✓	✓	✓	✓

4.1.2. Renderizado de errores

El Requisito RS-FN-25 pide que el segmento de código que causó el error aparezca resaltado en el mensaje de error, y el Requisito RS-FN-27 pide que los mensajes de error utilicen un código de colores. Esto significa que hay que realizar un renderizado [1] de los mensajes de error, es decir, convertir los datos del error en algo que se pueda mostrar por pantalla con el formato correcto.

El Requisito RS-NF-08 pide que el sistema soporte Node.js, el cual se ejecuta en terminal. Debido a esto, los mensajes de error solo pueden estar formados de texto, pudiendo usar secuencias de escape ANSI [65] para añadir formato básico. Además, los requisitos RS-NF-05 a RS-NF-07 indican que el sistema tiene que poder ejecutarse en navegador, donde se utiliza Hypertext Markup language (HTML) y Cascading Style Sheets (CSS) para añadir formato al texto. Cabe destacar que HTML y CSS tienen una mayor flexibilidad para dar formato al texto que las secuencias de escape ANSI.

Teniendo esto en cuenta, se ha decidido que los errores estén formados por texto con formato básico, permitiendo utilizar tanto secuencias de escape ANSI como HTML y CSS para esta tarea, según el entorno en el que se ejecute el sistema.

4.1.3. Análisis sintáctico

Para realizar el análisis sintáctico, como ya se ha explicado en la Sección 2.3, *Técnicas de análisis sintáctico*, actualmente existen múltiples métodos: utilizar un generador de *parsers*, utilizar un combinador de *parsers*, o crear un analizador sintáctico manualmente. [22] [24]

Los generadores de *parsers*, durante la generación del analizador sintáctico, realizan un procesado de la gramática que verifica ciertas restricciones y les permite generar un analizador sintáctico eficiente. Generalmente, esto es útil, ya que ayuda a detectar errores o ambigüedades en la gramática. Sin embargo, el Requisito RS-NF-02 indica que la de-

[1] Aunque la palabra 'renderizado' es un anglicismo, se utiliza en este TFG debido a su amplio uso en el campo de la informática y a que no existe un buen equivalente en castellano.

finición de la sintaxis de las instrucciones debe estar contenida en la ISA, y, por lo tanto, la gramática es parcialmente desconocida en tiempo de compilación. Esto imposibilita el uso de estas herramientas.

El siguiente método es crear el analizador sintáctico manualmente, típicamente utilizando la técnica de descenso recursivo [22]. Como ya se explicó, el principal problema de este método es su complejidad, requiriendo mucho tiempo para su desarrollo.

El último método es utilizar un combinador de *parsers*. Este método permite crear un analizador sintáctico de una forma similar a como se haría un *parser* descendente recursivo, obteniendo así muchos de los beneficios de la creación manual, pero simplificando mucho el proceso gracias a un mayor nivel de abstracción. Aunque el Requisito RS-NF-12 indica que el sistema debe ser rápido, la pérdida de rendimiento no es demasiado grande. Debido a esto, es preferible el menor consumo de tiempo de este método, ya que permitirá implementar más funcionalidades. Cabe destacar que en un futuro se podría reemplazar el analizador sintáctico por uno creado manualmente. Algunas de las herramientas de este tipo son Parsec [25], Chumsky [26], y pom [27].

Teniendo en cuenta esto, se ha elegido Chumsky como biblioteca para la creación del analizador sintáctico debido a varios motivos:

- Es una biblioteca de combinadores de *parsers*, lo que permite un corto tiempo de desarrollo.
- Está diseñada para generar buenos mensajes de error, y ser lo más rápida posible sin sacrificar la calidad de estos mensajes.
- Tiene soporte para Rust, el lenguaje de programación elegido anteriormente.

4.1.4. Análisis semántico

Como se explicó en la Sección 2.3, *Técnicas de análisis sintáctico*, el análisis semántico se encarga de, utilizando la representación intermedia generada por el *parser*, verificar que el programa cumple la semántica del lenguaje y generar el código compilado.

Aunque lógicamente el analizador semántico toma como entrada el resultado del *parser*, ambos componentes se pueden mezclar para evitar representar el código intermedio explícitamente, realizando ambas tareas simultáneamente y aumentando la eficiencia del compilador. Sin embargo, esto dificulta la tarea del analizador semántico al no poder explorar libremente el código intermedio. En la práctica, esto solo es posible para lenguajes y compiladores muy simples. Debido a esto, se ha optado por construir el código intermedio explícitamente para tener una mayor flexibilidad. [66]

Para la representación del código intermedio, existen principalmente dos métodos: utilizar un Abstract Syntax Tree (AST) y utilizar códigos de tres direcciones. Un AST es un árbol que representa la estructura sintáctica del código fuente, utilizando los nodos

internos para representar estructuras sintácticas del lenguaje y los nodos hoja para representar los *tokens* del código fuente. Los códigos de tres direcciones son instrucciones que toman como mucho dos variables, aplican una operación (típicamente un cálculo, comparación, o salto), y almacenan el resultado en otra variable. De entre estas opciones, se ha elegido utilizar un AST porque permite una representación más flexible que facilita la tarea del analizador semántico. [22]

Otra decisión importante para el desarrollo del analizador semántico es el tratamiento de las referencias hacia delante, pedidas por el Requisito RS-FN-17. Si el compilador escanease el código secuencialmente, al llegar a una etiqueta usada antes de su definición no podría realizar la traducción al desconocer el valor de esta etiqueta. Este problema se puede solucionar permitiendo la modificación de código ya traducido o realizando múltiples pasadas sobre el código. En el primer caso, cuando se encuentra una etiqueta sin definir, se registra en una tabla, y cuando se encuentra su definición se utiliza esta tabla para corregir sus usos. En el segundo caso, se realizan al menos dos pasadas por el código: en la primera se determina el espacio que ocupará cada elemento para asignar sus direcciones de memoria y se obtienen las definiciones de las etiquetas, mientras que en la segunda pasada se realiza la traducción del código. [67]

Realizar una única pasada es más eficiente, pero también es menos flexible y añade más complejidad al código. Actualmente, muchos ensambladores optan por realizar múltiples pasadas, ya que da más flexibilidad para implementar funcionalidades avanzadas que no pueden ser implementadas en un ensamblador de una única pasada [67]. Teniendo en cuenta esto, se ha elegido crear un compilador de múltiples pasadas para permitir una mayor flexibilidad para añadir nuevas funcionalidades en el futuro.

4.2. Arquitectura del compilador

El compilador a desarrollar está formado por cuatro componentes principales:

- Gestor de la arquitectura: encargado de validar y cargar la ISA a utilizar durante la compilación.
- Renderizador de errores: encargado de transformar la información relacionada con un error a un formato que puede ser mostrado a un usuario.
- Analizador sintáctico: encargado de realizar el análisis sintáctico del código ensamblador para permitir su análisis y traducción.
- Analizador semántico: encargado de realizar el análisis semántico del código ensamblador, y traducirlo a un formato que pueda ser utilizado por CREATOR.

La Figura 4.1 muestra el modelo de componentes UML [52] del compilador, incluyendo las relaciones entre los componentes.

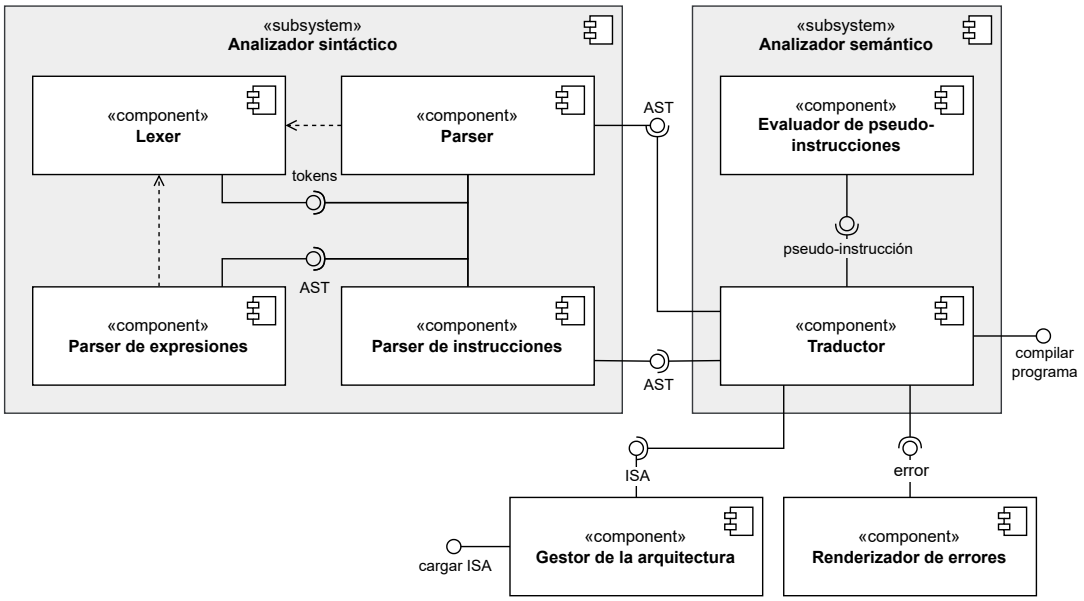

Fig. 4.1. Modelo de componentes del compilador

La tabla 4.2 contiene la plantilla utilizada para la especificación de los componentes, incluyendo la descripción de cada atributo. Tras esta, se incluyen las tablas con la especificación de cada uno de los componentes.

TABLA 4.2
PLANTILLA DE COMPONENTE

Nombre	
Rol	Papel del componente en el sistema.
Dependencias	Componentes que dependen de este.
Descripción	Explicación del funcionamiento del componente.
Datos	Datos de entrada (*in*) y salida (*out*) del componente.
Origen	Requisitos software que dieron lugar al componente.

TABLA 4.3
COMPONENTE 'GESTOR DE LA ARQUITECTURA'

Gestor de la arquitectura	
Rol	Valida y carga la ISA, permitiendo su uso en los otros componentes.
Dependencias	'Traductor'
Descripción	El gestor de la arquitectura se encarga de validar y cargar la especificación de la ISA proporcionada, generando diferentes errores en caso de fallo en la validación para evitar el uso de una ISA mal definida.
Datos	▪ **in:** Especificación de la ISA. ▪ **out:** Objeto que representa la ISA validada.
Origen	RS-FN-13

TABLA 4.4
COMPONENTE 'RENDERIZADOR DE ERRORES'

Renderizador de errores	
Rol	Transforma la información de un error a un formato que puede ser mostrado a un usuario.
Dependencias	'Traductor'
Descripción	El renderizador de errores se encarga de transformar la información relacionada con un error a un formato que puede ser mostrado a un usuario en la pantalla.
Datos	▪ **in:** Error en formato lógico. ▪ **out:** Representación del error que puede ser mostrada a un usuario.
Origen	RS-FN-25, RS-FN-26, RS-FN-27

TABLA 4.5
COMPONENTE 'LEXER'

Lexer	
Rol	Transforma el código del programa en una secuencia de *tokens*.
Dependencias	'Parser', 'Parser de instrucciones', y 'Parser de expresiones'
Descripción	El lexer recibe el código del programa a compilar y realiza un análisis léxico, transformado la secuencia de caracteres que representa el programa en una secuencia de *tokens* que puede ser utilizada por los demás componentes del analizador sintáctico.
Datos	▪ **in:** Programa a analizar. ▪ **out:** Secuencia de *tokens*.
Origen	RS-FN-01, RS-FN-02, RS-FN-03, RS-FN-05, RS-FN-11, RS-FN-12, RS-FN-08, RS-FN-14, RS-FN-15, RS-FN-18, RS-FN-19, RS-FN-23, RS-FN-25

TABLA 4.6
COMPONENTE 'PARSER DE EXPRESIONES'

Parser de expresiones	
Rol	Realiza el análisis sintáctico de una expresión aritmética.
Dependencias	'Parser' y 'Parser de instrucciones'
Descripción	El parser de expresiones realiza el análisis sintáctico de una expresión, transformando la secuencia de *tokens* que la representa en un AST para su posterior evaluación. Este componente tiene en cuenta el nivel de precedencia de los operadores al generar el AST.
Datos	▪ **in:** Secuencia de *tokens*. ▪ **out:** AST correspondiente a una expresión.
Origen	RS-FN-08, RS-FN-18, RS-FN-19, RS-FN-20, RS-FN-21, RS-FN-23, RS-FN-25

TABLA 4.7
COMPONENTE 'PARSER DE INSTRUCCIONES'

Parser de instrucciones	
Rol	Realiza el análisis sintáctico de una instrucción.
Dependencias	'Traductor'
Descripción	El parser de instrucciones realiza el análisis sintáctico de los argumentos de una instrucción según una cadena que define su sintaxis.
Datos	▪ **in:** Secuencia de *tokens* y definición de la sintaxis de la instrucción. ▪ **out:** AST correspondiente a la instrucción.
Origen	RS-FN-01, RS-FN-02, RS-FN-22, RS-FN-23, RS-FN-25

TABLA 4.8
COMPONENTE 'PARSER'

Parser	
Rol	Realiza el análisis sintáctico de un programa.
Dependencias	'Traductor'
Descripción	El parser realiza el análisis sintáctico de un programa, transformando la secuencia de *tokens* que lo representa en un AST para su posterior análisis semántico.
Datos	▪ **in:** Secuencia de *tokens*. ▪ **out:** AST correspondiente al programa.
Origen	RS-FN-01, RS-FN-02, RS-FN-03, RS-FN-05, RS-FN-16, RS-FN-23, RS-FN-25

TABLA 4.9
COMPONENTE 'EVALUADOR DE PSEUDO-INSTRUCCIONES'

Evaluador de pseudo-instrucciones	
Rol	Transforma una pseudo-instrucción en una secuencia de instrucciones.
Dependencias	'Traductor'
Descripción	El evaluador de pseudo-instrucciones se encarga de transformar las pseudo-instrucciones en una secuencia de instrucciones con el mismo comportamiento.
Datos	▪ **in:** AST de la pseudo-instrucción, ISA, tabla de símbolos, y dirección de la siguiente instrucción. ▪ **out:** Secuencia de ASTs correspondientes a las instrucciones por las que se tiene que remplazar la pseudo-instrucción.
Origen	RS-FN-02, RS-FN-25

TABLA 4.10
COMPONENTE 'TRADUCTOR'

Traductor	
Rol	Realiza el análisis semántico y traducción de un programa ensamblador.
Dependencias	N/A
Descripción	El traductor se encarga de realizar el análisis semántico y traducción de programas ensamblador según una ISA dada. Durante este proceso, verifica que no haya ningún error en el código, y lo traduce a una representación simplificada que puede ser ejecutada en CREATOR.
Datos	▪ **in:** ISA y código del programa a compilar. ▪ **out:** Programa compilado.
Origen	RS-FN-01, RS-FN-02, RS-FN-03, RS-FN-04, RS-FN-05, RS-FN-06, RS-FN-07, RS-FN-08, RS-FN-09, RS-FN-10, RS-FN-16, RS-FN-17, RS-FN-18, RS-FN-19, RS-FN-22, RS-FN-24, RS-FN-25, RS-FN-28

Trazabilidad

La matriz de trazabilidad (Tabla 4.11) permite verificar que todos los requisitos funcionales están cubiertos por al menos un componente, y que todos los componentes implementan algún requisito funcional.

TABLA 4.11

TRAZABILIDAD ENTRE LOS REQUISITOS FUNCIONALES Y LOS COMPONENTES

	RS-FN-01	RS-FN-02	RS-FN-03	RS-FN-04	RS-FN-05	RS-FN-06	RS-FN-07	RS-FN-08	RS-FN-09	RS-FN-10	RS-FN-11	RS-FN-12	RS-FN-13	RS-FN-14	RS-FN-15	RS-FN-16	RS-FN-17	RS-FN-18	RS-FN-19	RS-FN-20	RS-FN-21	RS-FN-22	RS-FN-23	RS-FN-24	RS-FN-25	RS-FN-26	RS-FN-27	RS-FN-28
Evaluador de pseudo-instrucciones		•																						•				
Gestor de la arquitectura												•																
Lexer	•	•	•		•			•			•	•		•	•			•	•				•		•			
Parser	•	•	•		•											•							•		•			
Parser de expresiones								•										•	•	•	•		•		•			
Parser de instrucciones	•	•																			•	•		•				
Renderizador de errores																								•	•	•		
Traductor	•	•	•	•	•	•	•	•	•	•						•	•	•	•			•		•				•

4.2.1. Analizador sintáctico

El analizador sintáctico se compondrá principalmente de un *lexer* y un *parser*. El *lexer* se encargará de realizar un procesado inicial del código para transformarlo en una secuencia de *tokens* (eliminando espacios y comentarios), mientras que el *parser* tomará esta secuencia de *tokens* y la transformará en un AST según la gramática del lenguaje. Además, también utilizará un *parser* de expresiones para realizar el análisis de las expresiones aritméticas, el cual tendrá en cuenta el nivel de precedencia de los diferentes operadores. Como el Requisito RS-FN-25 pide que los errores contengan la posición exacta del error, se añadirá también a los nodos del AST la posición y tamaño en el código original de cada elemento, para que el analizador semántico pueda acceder a esta información.

El Requisito RS-FN-22 indica que una instrucción puede tener múltiples sintaxis distintas. Para permitir esto, se ha decidido que en el análisis sintáctico inicial del código se interpretará la secuencia de *tokens* entre el nombre de la instrucción y el siguiente salto de línea como los argumentos de la instrucción, ya que es el funcionamiento de GAS [11] (como pide el Requisito RS-NF-11). El último componente del analizador sintáctico, el *parser* de instrucciones, intentará realizar el análisis sintáctico de esta secuencia de *tokens* según una especificación de la sintaxis de la instrucción.

A continuación se incluye la gramática en forma normal de Backus-Naur [68] que utilizará el *parser*. En general, se ha preferido simplificar la gramática y tratar los posibles errores en el analizador semántico, ya que esto permite generar errores con mayor información que pueden ser más útiles para un usuario, como pide el Requisito RS-FN-25.

⟨*label*⟩ ::= ⟨*identifier*⟩ ‘`:`’

⟨*directive_name*⟩ ::= ‘`.`’ ⟨*identifier*⟩

⟨*labelled_statement*⟩ ::= ⟨*labels*⟩ ⟨*statement*⟩ ‘`\n`’

⟨*labels*⟩ ::= λ | ⟨*label*⟩ ⟨*labels*⟩

⟨*statement*⟩ ::= ⟨*directive*⟩ | ⟨*instruction*⟩

⟨*directive*⟩ ::= ⟨*directive_name*⟩ ⟨*args*⟩

⟨*args*⟩ ::= λ | ⟨*arg*⟩ ⟨*args_list*⟩

⟨*args_list*⟩ ::= λ | ‘`,`’ ⟨*arg*⟩ ⟨*args_list*⟩

⟨*arg*⟩ ::= ⟨*expression*⟩ | ⟨*string*⟩

⟨*instruction*⟩ ::= ⟨*identifier*⟩ ⟨*tokens*⟩

⟨*tokens*⟩ ::= λ | ⟨*not_newline*⟩ ⟨*tokens*⟩

⟨*not_newline*⟩ ::= *Cualquier token excepto* ‘`\n`’

4.2.2. Analizador semántico

El analizador semántico se compondrá principalmente por un traductor y un evaluador de pseudo-instrucciones. El traductor realizará el análisis y traducción del AST obtenido del analizador semántico, y utilizará el evaluador de pseudo-instrucciones para transformar las pseudo-instrucciones en secuencias de instrucciones.

Como se especificó en la Subsección 4.1.4, *Análisis semántico*, el traductor realizará varias pasadas por el AST durante la traducción:

1. Pre-procesado del código
    - Para cada instrucción, se selecciona la definición de la instrucción a utilizar durante la traducción, determinando así su tamaño. En caso de corresponderse con una pseudo-instrucción, se expande en una secuencia de instrucciones que son pre-procesadas recursivamente.
    - Se asigna el espacio y posición de memoria que ocupará cada directiva de datos e instrucción.
    - Se obtiene el valor de todas las etiquetas definidas, y se almacenan en una tabla de símbolos.
2. Traducción del código: se traducen todos los elementos del código utilizando los valores de las etiquetas obtenidos en la primera pasada.
    - Se traducen las instrucciones utilizando la definición seleccionada en la primera pasada.
    - Se determinan los valores de las directivas de datos.

Durante este proceso, el analizador semántico verificará que el código no contiene errores semánticos. En caso de encontrarse un error, se abortará la compilación del programa con el error apropiado, incluyendo toda la información relevante sobre el error encontrado. Se deja como trabajo futuro aplicar estrategias de recuperación de error para permitir la detección de múltiples errores.

CAPÍTULO 5

IMPLEMENTACIÓN Y DESPLIEGUE

Este capítulo describirá la implementación y despliegue del sistema, y está dividido en dos secciones. La primera (Sección 5.1, *Implementación*), hará un resumen de las decisiones tomadas durante la implementación del sistema, junto con una descripción de la estructura de ficheros creada para organizar el código fuente. La segunda (Sección 5.2, *Despliegue*), describirá los requisitos y pasos necesarios para desplegar el sistema.

5.1. Implementación

Como se indicó en la Subsección 4.1.1, *Lenguaje de programación*, el sistema se ha desarrollado en Rust [38], utilizando la edición de 2021. Se ha utilizado Wasm-pack [64] para la generación de bindings de JS.

El sistema se ha implementado como se indicó en la Sección 4.2, *Arquitectura del compilador*. Cada subsistema se ha implementado en un módulo, con sus componentes almacenados en submódulos. Cabe destacar que el componente principal de cada subsistema se ha implementado en el módulo principal del subsistema. Además, se han añadido submódulos adicionales al analizador semántico para la gestión de diversas funcionalidades necesarias: campos de bits, etiquetas y tabla de símbolos, enteros de tamaño fijo pero arbitrario, y gestión de secciones de memoria. También se ha añadido un módulo para la implementación de la API de JS. Dentro de cada módulo y submódulo también se incluyen pruebas unitarias que verifican su correcta funcionalidad.

Para la implementación del sistema se han utilizado múltiples bibliotecas externas:

- Para la deserialización de JSON se han utilizado *serde* [69] y *serde_json* [70].
- Para el renderizado de los errores se ha utilizado *ariadne* [71].

- Como se indicó en la Subsección 4.1.3, *Análisis sintáctico*, para la realización del analizador sintáctico se ha utilizado *chumsky* [26].
- Para el uso de expresiones regulares se ha utilizado *regex* [72] y *once_cell* [73].
- Para el uso de números enteros de tamaño arbitrario se ha utilizado *num-bigint* [74] y *num-traits* [75].
- Para la generación de *bindings* para JS se ha utilizado *wasm-bindgen* [76] y *self_cell* [77].
- Para el soporte de renderización de errores con formato HTML se ha utilizado *ansi-to-html* [78].

La Figura 5.1 muestra la estructura de ficheros del proyecto, relacionando los ficheros con los componentes descritos. Cabe destacar que los módulos y submódulos se han organizado siguiendo el estándar moderno de Rust, según el cual un módulo se implementa en un fichero, y sus submódulos en ficheros dentro de un directorio con el nombre del módulo padre [38].

El código fuente está disponible en https://github.com/ALVAROPING1/CreatorCompiler.

```
/
├── js_example/ ..... Ejemplos de uso
│   ├── compiler.mjs ..... Uso compilador
│   ├── index.html ..... Página navegador
│   ├── node.js ..... Carga en Node.js
│   └── web.js ..... Carga en navegador
├── src/
│   ├── architecture/ ..... Deserialización de la ISA
│   │   ├── json.rs ..... Deserialización JSON
│   │   └── utils.rs ..... Utilidades de deserialización JSON
│   ├── compiler/ ..... Subsistema analizador semántico
│   │   ├── bit_field.rs ..... Campo de bits
│   │   ├── error.rs ..... Gestión errores
│   │   ├── integer.rs ..... Enteros
│   │   ├── label.rs ..... Etiquetas y tabla de símbolos
│   │   ├── pseudoinstruction.rs ..... Evaluador de pseudo-instrucciones
│   │   └── section.rs ..... Gestor de segmentos de memoria
│   ├── parser/ ..... Subsistema analizador sintáctico
│   │   ├── error.rs ..... Gestión errores
│   │   ├── expression.rs ..... Parser de expresiones
│   │   ├── instruction.rs ..... Parser de instrucciones
│   │   └── lexer.rs ..... Lexer
│   ├── architecture.rs ..... Gestor de la arquitectura
│   ├── compiler.rs ..... Traductor
│   ├── error_rendering.rs ..... Renderizador de errores
│   ├── js.rs ..... API JS
│   ├── lib.rs ..... Biblioteca del compilador
│   ├── parser.rs ..... Parser
│   └── span.rs ..... Región de código
├── tests/ ..... Arquitecturas de prueba
│   ├── architecture.json
│   └── architecture2.json
├── CONTRIBUTING.md
├── Cargo.lock ..... Versiones de dependencias
├── Cargo.toml ..... Metadatos de la biblioteca
├── LICENSE
├── README.md
└── build.sh ..... Compilación biblioteca
```

Fig. 5.1. Estructura de ficheros del proyecto

5.2. Despliegue

Las especificaciones técnicas recomendadas para que un usuario tenga una buena experiencia utilizando la herramienta son las siguientes:

- **Sistema Operativo (SO):** Ubuntu 22.04 LTS (Linux) / Windows 11.
- **Procesador:** Intel® Core™ i3 CPU 7100 @3.9GHz o superior.
- **Memoria RAM:** 2GB o superior.
- **Almacenamiento:** 1GB o superior.
- **Red:** se necesita una conexión de red para descargar las dependencias durante la compilación del sistema.
- **Software:**
  - Los navegadores recomendados para utilizar el sistema son Google Chrome 70+, Mozilla Firefox 60+, y Safari 12+.
  - Para compilar el sistema, es necesario tener instalado el compilador de Rust y `cargo` [79], y Wasm-pack [64].

Para compilar el sistema, es necesario seguir los siguientes pasos:

1. Descargar y descomprimir el código fuente desde el repositorio.
2. Compilar el código fuente con Wasm-pack [64]. Se incluye un *script* con el código fuente para simplificar esta tarea, que se puede ejecutar con el comando `./build.sh`.
3. Incluir el sistema en una aplicación web o Node.js. Junto con el código fuente, dentro del directorio `js_example`, se incluye un ejemplo funcional de cómo se puede realizar esta tarea.

Además, el sistema se ha integrado en CREATOR. Actualmente, está disponible en su versión de desarrollo, que se puede encontrar en https://creatorsim-community.github.io/creator-development/. En un futuro cercano se añadirá a la versión estable, disponible en https://creatorsim.github.io/. Esto permite el uso del sistema sin necesidad de realizar el proceso anterior.

CAPÍTULO 6

VALIDACIÓN, VERIFICACIÓN, Y EVALUACIÓN

Este capítulo verificará que la solución propuesta cumple con todos los requisitos especificados en el Capítulo 3, *Análisis*, y está dividido en dos secciones. La primera (Sección 6.1, *Verificación y Validación*), contendrá la verificación y validación de software, mientras que la segunda (Sección 6.2, *Evaluación*) contendrá una evaluación del sistema, comparándolo con otras herramientas existentes.

6.1. Verificación y Validación

En ingeniería de software, la verificación y validación de software es el proceso de comprobar que el sistema cumple con los requisitos establecidos y la función para la que se diseñó [80]. La Figura 6.1 muestra un resumen de este proceso.

Como se explicó en el Capítulo 3, *Análisis*, el cliente establece sus necesidades y lo que espera del producto, definiendo los requisitos de software. A partir de estos requisitos, el analista se encarga de definir los requisitos de software.

La verificación de software (Subsección 6.1.1, *Verificación*) se encarga de, durante el desarrollo del sistema, comprobar que los productos de una fase de desarrollo cumplen con los requisitos establecidos al inicio de la fase; es decir, comprobar que se está desarrollando el sistema correctamente. Por otro lado, la validación de software (Subsección 6.1.2, *Validación*) se encarga de, al final del desarrollo del sistema, comprobar que cumple con los requisitos establecidos por el cliente; es decir, comprobar que se ha desarrollado el sistema correcto. [80]

Cada caso de prueba se identifica con un ID que sigue el formato *YYY-XX*, donde *YYY* identifica el tipo del caso de prueba, ya sea *VET* (verificación) o *VAT* (validación); y *XX*

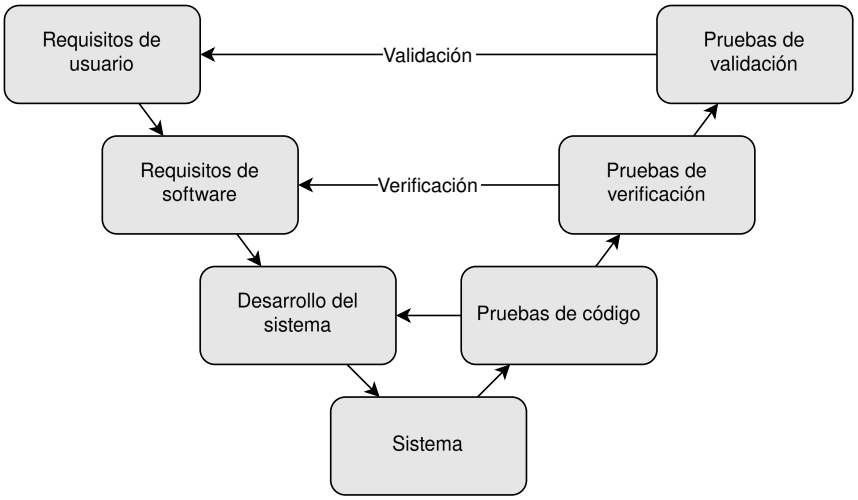

Fig. 6.1. Resumen de la verificación y validación de software

identifica el número secuencial del caso de prueba, empezando en *01*.

La tabla 6.1 contiene la plantilla utilizada para la especificación de los casos de prueba, incluyendo la descripción de cada atributo.

TABLA 6.1
PLANTILLA DE PRUEBA

YYY-XX	
Descripción	Descripción del test.
Pre-condición	Condiciones que deben cumplirse para realizar el test.
Procedimiento	Descripción de los pasos a seguir para realizar el test.
Post-condición	Condiciones que deben cumplirse después de realizar el test para pasarlo.
Origen	Requisitos que originaron el test.
Evaluación	Resultado del test (*OK* o *Error*).

6.1.1. Verificación

Esta sección contiene las pruebas de verificación realizadas. Estas pruebas, además de verificar que se cumplen los requisitos de software, también verifican que el resultado del sistema desarrollado es el correcto en todos los casos probados. Después de realizar estas pruebas, se ha realizado una matriz de trazabilidad (Tabla 6.19) para comprobar que todos los requisitos de software están cubiertos por al menos una prueba de verificación.

TABLA 6.2
PRUEBA VET-01

VET-01	
Descripción	Verificar que el sistema puede compilar programas según una ISA.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA básica con diferentes registros, instrucciones, y directivas de ensamblador. 2. Escribir un programa que utilice todos los registros, instrucciones, y directivas de ensamblador definidas en la ISA, múltiples etiquetas en una misma instrucción y directiva de datos, y etiquetas definidas después de su uso. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente al programa introducido.
Origen	RS-FN-01, RS-FN-03, RS-FN-04, RS-FN-13, RS-FN-16, RS-FN-17, RS-NF-01, RS-NF-02, RS-NF-11
Evaluación	OK

TABLA 6.3
PRUEBA VET-02

VET-02	
Descripción	Verificar que el sistema puede compilar pseudo-instrucciones.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA básica con diferentes instrucciones y pseudo-instrucciones. 2. Escribir un programa que utilice las pseudo-instrucciones, definidas en la ISA. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado con las secuencias de instrucciones correspondientes a las pseudo-instrucciones introducidas.
Origen	RS-FN-01, RS-FN-02
Evaluación	OK

TABLA 6.4
PRUEBA VET-03

VET-03	
Descripción	Verificar que el sistema permite el uso de todas las directivas de datos soportadas.
Pre-condición	El sistema está instalado y se tiene una ISA cargada con todos los tipos de directivas de datos soportadas.
Procedimiento	1. Escribir un programa que utilice todos los tipos de directivas de datos soportadas: ▪ Cadenas de caracteres terminadas y no terminadas en un byte nulo. ▪ Números enteros de todos los tamaños soportados. ▪ Números decimales de precisión simple y doble. ▪ Reservar espacio. ▪ Alinear la memoria de datos a una potencia de 2 y a un tamaño en bytes. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con los valores de las directivas de datos introducidas.
Origen	RS-FN-03, RS-FN-05, RS-FN-06, RS-FN-07, RS-FN-08, RS-FN-09
Evaluación	OK

TABLA 6.5
PRUEBA VET-04

VET-04	
Descripción	Verificar que el sistema permite el uso bibliotecas.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa que declare algunas de sus etiquetas como globales. 2. Compilar el programa y guardarlo como una biblioteca. 3. Escribir un nuevo programa que utilice las etiquetas declaradas como globales en el programa original. 4. Compilar el nuevo programa añadiendo el original como una biblioteca.
Post-condición	Se obtiene el código compilado correspondiente al programa introducido.
Origen	RS-FN-03, RS-FN-10
Evaluación	OK

TABLA 6.6
PRUEBA VET-05

VET-05	
Descripción	Verificar que el sistema permite el uso secuencias de escape en cadenas de caracteres y caracteres literales, y que las cadenas están codificadas en UTF-8 [13].
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con cadenas de caracteres y caracteres literales que contengan todas las secuencias de escape soportadas y caracteres no ASCII. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con la codificación UTF-8 [13] correspondiente a las cadenas de caracteres introducidas, y los valores Unicode [13] correspondientes a los caracteres literales.
Origen	RS-FN-11, RS-FN-12
Evaluación	OK

TABLA 6.7
PRUEBA VET-06

VET-06	
Descripción	Verificar que el sistema permite el uso comentarios y la selección del prefijo de los comentarios de línea.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con comentarios de línea y multilínea. 2. Compilar el programa y verificar que los comentarios son ignorados. 3. Modificar el prefijo de los comentarios de línea en la definición de la ISA, y cargar la nueva versión. 4. Modificar el prefijo de los comentarios de línea en el programa. 5. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente al programa, ignorando los comentarios introducidos.
Origen	RS-FN-14, RS-FN-15
Evaluación	OK

TABLA 6.8
PRUEBA VET-07

VET-07	
Descripción	Verificar que el sistema permite el uso expresiones aritméticas de números enteros.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa que utilice expresiones de números enteros, caracteres literales, y etiquetas, y todos los operadores soportados, como valores inmediatos de instrucciones y valores de directivas de datos que acepten números enteros como argumentos. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con los valores de las expresiones introducidas.
Origen	RS-FN-18, RS-FN-19, RS-FN-20, RS-FN-21
Evaluación	OK

TABLA 6.9
PRUEBA VET-08

VET-08	
Descripción	Verificar que el sistema permite el uso expresiones aritméticas de números decimales.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa que utilice expresiones de números enteros y decimales, caracteres literales, y etiquetas, y todos los operadores soportados (excepto los operadores bit a bit), como valores de directivas de datos que acepten números decimales como argumentos. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con los valores de las expresiones introducidas.
Origen	RS-FN-18, RS-FN-19, RS-FN-21
Evaluación	OK

TABLA 6.10
PRUEBA VET-09

VET-09	
Descripción	Verificar que el sistema permite definir múltiples instrucciones con el mismo nombre, y utilizar su sintaxis y el tamaño de sus campos para seleccionar la correcta durante la compilación.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA con múltiples instrucciones con el mismo nombre pero diferente sintaxis y/o tamaño de sus argumentos. 2. Escribir un programa que utilice todas las definiciones de la instrucción creada. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente a las instrucciones introducidas.
Origen	RS-FN-22
Evaluación	OK

TABLA 6.11
PRUEBA VET-10

VET-10	
Descripción	Verificar que el sistema detecta errores de sintaxis y genera buenos mensajes de error.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con un error de sintaxis. 2. Compilar el programa.
Post-condición	Se obtiene un mensaje de error de sintaxis con toda la información pedida en el Requisito RS-FN-25 resaltada con un código de colores.
Origen	RS-FN-23, RS-FN-25, RS-FN-27
Evaluación	OK

TABLA 6.12
PRUEBA VET-11

VET-11	
Descripción	Verificar que el sistema detecta errores semánticos y genera buenos mensajes de error.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con un error semántico. 2. Compilar el programa.
Post-condición	Se obtiene un mensaje de error semántico con toda la información pedida en el Requisito RS-FN-25 resaltada con un código de colores.
Origen	RS-FN-24, RS-FN-25, RS-FN-27
Evaluación	OK

TABLA 6.13
PRUEBA VET-12

VET-12	
Descripción	Verificar que los mensajes de error del sistema referentes a identificadores no definidos contienen el nombre del elemento más similar resaltado con un código de colores.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con una directiva, instrucción, registro, o etiqueta no definida, utilizando un nombre similar a una definida. 2. Compilar el programa.
Post-condición	Se obtiene un mensaje de error semántico con el nombre del elemento más similar resaltado con un código de colores.
Origen	RS-FN-24, RS-FN-26, RS-FN-27
Evaluación	OK

TABLA 6.14
PRUEBA VET-13

VET-13	
Descripción	Verificar que el sistema permite el uso de números enteros de cualquier tamaño.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA con una directiva de datos de enteros con un tamaño máximo arbitrariamente grande. 2. Escribir un programa que utilice la directiva definida con un número entero cercano al valor máximo. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado con el valor introducido.
Origen	RS-FN-28
Evaluación	OK

TABLA 6.15
PRUEBA VET-14

VET-14	
Descripción	Verificar que el sistema utiliza la API implementada en el compilador antiguo.
Pre-condición	El código del sistema y del compilador antiguo está descargado.
Procedimiento	1. Acceder a la definición de la interfaz de JS del compilador. 2. Acceder a la definición de la interfaz del compilador antiguo.
Post-condición	El sistema utiliza la API de entrada y salida en JS implementada en el compilador antiguo.
Origen	RS-NF-03, RS-NF-04
Evaluación	OK

TABLA 6.16
PRUEBA VET-15

VET-15	
Descripción	Verificar que el sistema funciona en todas las plataformas soportadas por el compilador antiguo.
Pre-condición	El sistema está instalado.
Procedimiento	1. Acceder el sistema desde Google Chrome 70, Mozilla Firefox 70, Safari 10, y Node.js. 2. Utilizar el sistema en esas plataformas.
Post-condición	El sistema funciona correctamente en todas las plataformas probadas.
Origen	RS-NF-05, RS-NF-06, RS-NF-07, RS-NF-08
Evaluación	OK

TABLA 6.17
PRUEBA VET-16

VET-16	
Descripción	Verificar que el sistema es FOSS.
Pre-condición	N/A
Procedimiento	1. Acceder al repositorio con el código fuente del sistema. 2. Verificar que el código fuente es público y se distribuye con una licencia FOSS.
Post-condición	El sistema cumple con los requisitos para ser FOSS según las definiciones de la FSF [50] y OSI [51].
Origen	RS-NF-09, RS-NF-10
Evaluación	OK

TABLA 6.18
PRUEBA VET-17

VET-17	
Descripción	Verificar que el sistema es rápido.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con 1000 instrucciones. 2. Compilar el programa.
Post-condición	El sistema cumple con los requisitos de velocidad.
Origen	RS-NF-12
Evaluación	OK

Tabla 6.19

Trazabilidad entre los requisitos de software y las pruebas de verificación

	RS-FN-01	RS-FN-02	RS-FN-03	RS-FN-04	RS-FN-05	RS-FN-06	RS-FN-07	RS-FN-08	RS-FN-09	RS-FN-10	RS-FN-11	RS-FN-12	RS-FN-13	RS-FN-14	RS-FN-15	RS-FN-16	RS-FN-17	RS-FN-18	RS-FN-19	RS-FN-20	RS-FN-21	RS-FN-22	RS-FN-23	RS-FN-24	RS-FN-25	RS-FN-26	RS-FN-27	RS-FN-28	RS-NF-01	RS-NF-02	RS-NF-03	RS-NF-04	RS-NF-05	RS-NF-06	RS-NF-07	RS-NF-08	RS-NF-09	RS-NF-10	RS-NF-11	RS-NF-12
VET-01	•		•	•									•			•	•												•	•								•	
VET-02	•	•																																					
VET-03			•		•	•	•	•	•																														
VET-04			•							•																														
VET-05											•	•																												
VET-06													•	•																									
VET-07																	•	•	•	•																			
VET-08																	•	•		•																			
VET-09																					•																		
VET-10																						•		•		•													
VET-11																							•	•		•													
VET-12																							•		•	•													
VET-13																											•												
VET-14																														•	•								
VET-15																																•	•	•	•				
VET-16																																				•	•		
VET-17																																							•

6.1.2. Validación

Esta sección contiene las pruebas de validación realizadas. Después de realizar estas pruebas, se ha realizado una matriz de trazabilidad (Tabla 6.31) para comprobar que todos los requisitos de usuario están cubiertos por al menos una prueba de validación.

TABLA 6.20
PRUEBA VAT-01

VAT-01	
Descripción	Validar que el sistema ofrece todas las funcionalidades del compilador antiguo.
Pre-condición	El sistema está instalado.
Procedimiento	1. Obtener los programas de ejemplo ofrecidos por CREATOR junto con la definición de su ISA. 2. Cargar la ISA en el compilador. 3. Intentar compilar todos los programas anteriores.
Post-condición	Todos los programas evaluados obtienen el mismo resultado que con el compilador antiguo.
Origen	RU-CA-01, RU-CA-04, RU-RE-04
Evaluación	OK

TABLA 6.21
PRUEBA VAT-02

VAT-02	
Descripción	Validar que el sistema permite el uso secuencias de escape en cadenas de caracteres y caracteres literales, y que las cadenas están codificadas en UTF-8 [13].
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con cadenas de caracteres y caracteres literales que contengan secuencias de escape y caracteres no ASCII. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con la codificación UTF-8 [13] correspondiente a las cadenas de caracteres introducidas, y los valores Unicode [13] correspondientes a los caracteres literales.
Origen	RU-CA-02, RU-CA-03
Evaluación	OK

TABLA 6.22
PRUEBA VAT-03

VAT-03	
Descripción	Validar que el sistema permite el uso comentarios y la selección del prefijo de los comentarios de línea.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa con comentarios de línea y multilínea. 2. Compilar el programa y verificar que los comentarios son ignorados. 3. Modificar el prefijo de los comentarios de línea en la definición de la ISA, y cargar la nueva versión. 4. Modificar el prefijo de los comentarios de línea en el programa. 5. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente al programa, ignorando los comentarios introducidos.
Origen	RU-CA-05, RU-CA-06
Evaluación	OK

TABLA 6.23
PRUEBA VAT-04

VAT-04	
Descripción	Validar que el sistema puede permite el uso de múltiples etiquetas en una sentencia, y utilizar etiquetas definidas después de su uso.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa que utilice múltiples etiquetas en una misma instrucción y directiva de datos, y etiquetas definidas después de su uso. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente al programa introducido.
Origen	RU-CA-07, RU-CA-08
Evaluación	OK

TABLA 6.24
PRUEBA VAT-05

VAT-05	
Descripción	Validar que el sistema permite el uso expresiones aritméticas con etiquetas.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa que utilice expresiones aritméticas de números enteros, caracteres literales, y etiquetas como valores inmediatos de instrucciones y valores de directiva de datos. 2. Compilar el programa.
Post-condición	Se obtiene el código compilado con los valores de las expresiones introducidas.
Origen	RU-CA-09, RU-CA-10
Evaluación	OK

TABLA 6.25
PRUEBA VAT-06

VAT-06	
Descripción	Validar que el sistema permite definir múltiples instrucciones con el mismo nombre, y utilizar su sintaxis y el tamaño de sus campos para seleccionar la correcta durante la compilación.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA con múltiples instrucciones con el mismo nombre pero diferente sintaxis y/o tamaño de sus argumentos. 2. Escribir un programa que utilice todas las definiciones de la instrucción creada. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado correspondiente a las instrucciones introducidas.
Origen	RU-CA-11
Evaluación	OK

TABLA 6.26
PRUEBA VAT-07

VAT-07	
Descripción	Validar que el sistema detecta errores en el código ensamblador y genera buenos mensajes de error.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir varios programas con un diferentes errores, tanto sintácticos como semánticos. 2. Compilar los programas.
Post-condición	Se obtienen los mensajes de error apropiados con toda la información relevante para solucionar el problema.
Origen	RU-CA-12, RU-CA-13
Evaluación	OK

TABLA 6.27
PRUEBA VAT-08

VAT-08	
Descripción	Validar que el sistema permite el uso de números enteros de cualquier tamaño.
Pre-condición	El sistema está instalado.
Procedimiento	1. Implementar una ISA con una directiva de datos de enteros con un tamaño máximo arbitrariamente grande. 2. Escribir un programa que utilice la directiva definida con un número entero cercano al valor máximo. 3. Cargar la ISA en el compilador. 4. Compilar el programa.
Post-condición	Se obtiene el código compilado con el valor introducido.
Origen	RU-CA-14
Evaluación	OK

TABLA 6.28
PRUEBA VAT-09

VAT-09	
Descripción	Validar que el sistema está integrado en CREATOR.
Pre-condición	N/A.
Procedimiento	1. Acceder a la página de CREATOR desde los navegadores soportados por el compilador antiguo. 2. Escribir un programa que utilice funcionalidades del nuevo compilador no soportadas por el antiguo. 3. Compilar el programa.
Post-condición	El programa compila sin errores.
Origen	RU-RE-01, RU-RE-02
Evaluación	OK

TABLA 6.29
PRUEBA VAT-10

VAT-10	
Descripción	Validar que el sistema es FOSS.
Pre-condición	N/A
Procedimiento	1. Acceder al repositorio con el código fuente del sistema. 2. Verificar que el código fuente es público y se distribuye con una licencia FOSS.
Post-condición	El sistema cumple con los requisitos para ser FOSS según las definiciones de la FSF [50] y OSI [51].
Origen	RU-RE-03
Evaluación	OK

TABLA 6.30
PRUEBA VAT-11

VAT-11	
Descripción	Validar que el sistema es rápido.
Pre-condición	El sistema está instalado y se tiene una ISA cargada.
Procedimiento	1. Escribir un programa complejo. 2. Compilar el programa.
Post-condición	El sistema cumple con los requisitos de velocidad.
Origen	RU-RE-05
Evaluación	OK

TABLA 6.31
TRAZABILIDAD ENTRE LOS REQUISITOS DE USUARIO Y LAS PRUEBAS DE VALIDACIÓN

	RU-CA-01	RU-CA-02	RU-CA-03	RU-CA-04	RU-CA-05	RU-CA-06	RU-CA-07	RU-CA-08	RU-CA-09	RU-CA-10	RU-CA-11	RU-CA-12	RU-CA-13	RU-CA-14	RU-RE-01	RU-RE-02	RU-RE-03	RU-RE-04	RU-RE-05
VAT-01	•			•													•		
VAT-02		•	•																
VAT-03					•	•													
VAT-04							•	•											
VAT-05									•	•									
VAT-06											•								
VAT-07												•	•						
VAT-08														•					
VAT-09															•	•			
VAT-10																	•		
VAT-11																			•

6.2. Evaluación

Para evaluar el sistema desarrollado se van a comparar los mensajes de error generados por este con los generados por el compilador anteriormente utilizado por CREATOR [1]. Además, también se compararán con los generados por GAS [9], uno de los ensambladores más utilizados. Para realizar esto, se realizarán varias pruebas con programas a los que se ha añadido un error, y se compararán los mensajes de error generados por cada una de las herramientas. Además, también se comparará el rendimiento de estas alternativas con un programa complejo.

La primera prueba consiste en definir una etiqueta múltiples veces. Los resultados obtenidos se pueden ver en la Figura 6.2. Como se puede observar, el nuevo compilador, además de indicar el error producido, también indica por qué se ha producido y cómo corregirlo (Figura 6.2a), ayudando a los usuarios a solucionar el problema. El compilador antiguo y GAS (figuras 6.2b y 6.2c), en cambio, únicamente indican la etiqueta repetida y su localización.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 1.s -o min --color
[E11] Error: Label repeated is already defined
     ┌─[ assembly:10:1 ]
     │
   3 │ repeated: li t0, 0
     │ ────┬─────
     │     └─────── Note: Label also defined here
     │
  10 │ repeated: add t2, t0, t1
     │ ────┬─────
     │     └─────── Duplicate label
     │
     │ Help: Consider renaming either of the labels
─────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(a) Compilador nuevo

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 1.s -o min
Error at line 10 (repeated):
Repeated tag: repeated

  9 li t1, 10
->10 repeated: add t2, t0, t1
  11

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(b) Compilador antiguo

```
$ riscv64-unknown-elf-as -march=rv32i -o 1.o 1.s
1.s: Assembler messages:
1.s:10: Error: symbol `repeated' is already defined
```

(c) GNU Assembler

Fig. 6.2. Evaluación 1

El siguiente caso de prueba consiste en intentar utilizar una directiva con menos argumentos de los necesarios. Los resultados obtenidos se pueden ver en la Figura 6.3. Como se puede ver, el nuevo compilador es el único que detecta correctamente el error, e indica cómo solucionarlo (Figura 6.3a). El compilador antiguo (Figura 6.3b) produce un error incorrecto al interpretar erróneamente el código. Cabe destacar que GAS (Figura 6.3c) no produce ningún error en este caso, pero un análisis del código generado indica que asume un valor de 0 para el argumento faltante, en vez de generar un error.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 2.s -o min --color
[E09] Error: Incorrect amount of arguments, expected 1 but found 0
    ┌─[ assembly:3:7 ]
    │
  3 │ .align
    │       ┬
    │       └── This directive has 0 arguments
    │
    │ Help: Consider adding the missing 1 argument
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(a) Compilador nuevo

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 2.s -o min
Error at line 4 (.word):
Invalid value '.word' as number.

  3 .align
->4 .word 10000
  5 .text

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(b) Compilador antiguo

```
$ riscv64-unknown-elf-as -march=rv32i -o 2.o 2.s
```

(c) GNU Assembler

Fig. 6.3. Evaluación 2

El siguiente caso de prueba consiste en intentar utilizar un entero demasiado grande en una directiva de datos. Los resultados obtenidos se pueden ver en la Figura 6.4. Como se puede ver, el nuevo compilador es el único que muestra cuál es el rango de enteros válidos para la directiva de ensamblador (Figura 6.4a), ayudando a los usuarios a aprender qué valores pueden ser utilizados. El compilador antiguo y GAS (figuras 6.4b y 6.4c), en cambio, únicamente indican que el número es demasiado grande.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 3.s -o min --color
[E22] Error: Value 65536 is outside of the valid range of the field
    ┌─[ assembly:2:7 ]
    │
  2 │ .half 0x10000
    │       ──┬───
    │         └──── This expression has value 65536
    │
    │ Note: Allowed range is [-32768, 65535]
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(a) Compilador nuevo

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 3.s -o min
Error at line 2 (0x10000):
Number '0x10000' is too big

  1 .data
->2 .half 0x10000
  3 .text

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(b) Compilador antiguo

```
$ riscv64-unknown-elf-as -march=rv32i -o 3.o 3.s
3.s: Assembler messages:
3.s:2: Warning: value 0x10000 truncated to 0x0
```

(c) GNU Assembler

Fig. 6.4. Evaluación 3

El siguiente caso de prueba consiste en añadir instrucciones a la sección de datos. Los resultados obtenidos se pueden ver en la Figura 6.5. Como se puede ver, el nuevo compilador es el único que detecta correctamente el error, e indica cómo solucionarlo (Figura 6.5a). El compilador antiguo (Figura 6.5b) produce un error incorrecto al interpretar erróneamente el código. Cabe destacar que GAS (Figura 6.5c) no produce ningún error en este caso, pero un análisis del código generado indica que añade la codificación binaria de las instrucciones como si fueran datos.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 4.s -o min --color
[E17] Error: Can't use instruction statements while in section Data
    ┌[ assembly:3:7 ]
  1 │ .data
    │ ┬──┘
    │ └─── Note: Section previously started here
    │
  3 │ main: li t0, 0
    │       ┬──┘
    │       └─── This statement can't be used in the current section
    │
    │ Help: Consider changing the section to Text, using .text
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(a) Compilador nuevo

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 4.s -o min
Error at line 3 (li):
Invalid directive: li

  2 .word 4096
->3 main: li t0, 0
  4 addi t0, t0, 100

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(b) Compilador antiguo

```
$ riscv64-unknown-elf-as -march=rv32i -o 4.o 4.s
```

(c) GNU Assembler

Fig. 6.5. Evaluación 4

El último caso de prueba consiste en utilizar una instrucción con un error tipográfico en su nombre. Los resultados obtenidos se pueden ver en la Figura 6.6. Como se puede ver, el nuevo compilador es el único que, además de detectar que el nombre no se corresponde con una instrucción definida, muestra las instrucciones cuyo nombre es similar al utilizado (Figura 6.6a). Esto ayuda a los usuarios a ver que se han equivocado al escribir el nombre de la instrucción, permitiéndoles solucionar el problema rápidamente. El compilador antiguo y GAS (figuras 6.6b y 6.6c), en cambio, únicamente indican que el nombre de la instrucción no se corresponde con una instrucción definida.

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 5.s -o min --color
[E02] Error: Instruction addo isn't defined
    ┌─[ assembly:7:7 ]
    │
  7 │        addo t0, t0, 1
    │        ──┬──
    │          └─── Unknown instruction
    │
    │     Help: Did you mean add or addi?
────┘

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(a) Compilador nuevo

```
$ ./creator.sh -a RISC_V_RV32IMFD.json -s 5.s -o min
Error at line 7 (addo):
Instruction 'addo' not found

  6       add t2, t2, t0
->7       addo t0, t0, 1
  8       j loop

Not executed
keyboard[0x0]:''; display[0x0]:'';
```

(b) Compilador antiguo

```
$ riscv64-unknown-elf-as -march=rv32i -o 5.o 5.s
5.s: Assembler messages:
5.s:7: Error: unrecognized opcode `addo t0,t0,1'
```

(c) GNU Assembler

Fig. 6.6. Evaluación 5

Para evaluar el rendimiento del sistema se va a utilizar ForthV [81]. ForthV es un intérprete del lenguaje de programación Forth escrito en ensamblador RISC-V con unas 2800 líneas de código, por lo que representa un programa complejo real. Debido a esto, es un buen caso de prueba en el que evaluar el rendimiento del sistema. Cabe destacar que este programa utiliza funcionalidades no soportadas por el compilador antiguo (como expresiones aritméticas y uso de etiquetas como valores de directivas de datos). Debido a esto, para evaluar el rendimiento en el compilador antiguo, se ha tenido que modificar el programa para evitar el uso de estas funcionalidades.

La Figura 6.7 muestra los resultados de la ejecución. GAS no se ha podido ejecutar en un entorno web. Como se puede observar, el nuevo compilador ofrece una mejora significativa de rendimiento con respecto del compilador antiguo, especialmente en un entorno web que se corresponde con el entorno en el que más se utiliza CREATOR. Se puede ver

que GAS es significativamente más rápido que el compilador nuevo, lo que indica que el rendimiento todavía se podría mejorar mucho si este se volviese un problema.

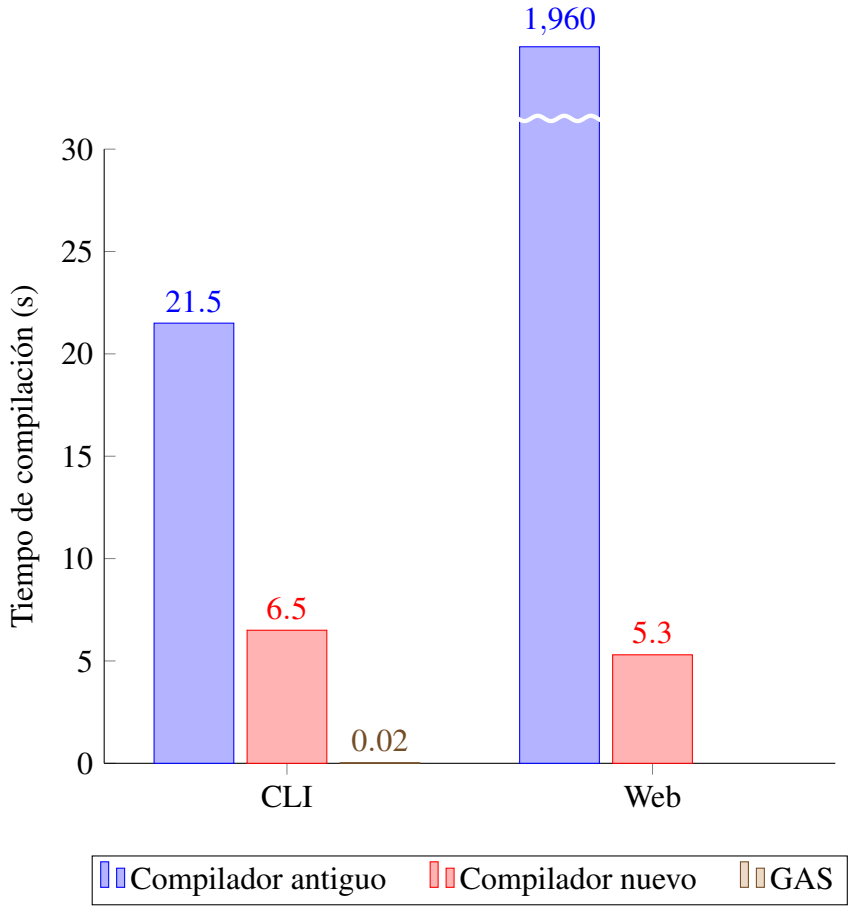

Fig. 6.7. Comparativa de rendimiento entre los diferentes compiladores

CAPÍTULO 7

PLAN DEL PROYECTO

Este capítulo describirá el plan del proyecto a desarrollar y está dividido en cuatro secciones. La primera (Sección 7.1, *Planificación*), explicará la planificación del proyecto. La segunda (Sección 7.2, *Presupuesto*), describirá el presupuesto con el que se desarrollará el sistema y sus costes. La tercera (Sección 7.3, *Marco regulador*), explicará las restricciones legales que se aplican al proyecto. Por último, la cuarta (Sección 7.4, *Entorno socioeconómico*), describirá el entorno socioeconómico en el que se desarrollará el proyecto.

7.1. Planificación

Esta sección explicará la planificación del proyecto, incluyendo la metodología utilizada, las etapas en las que se dividirá el proyecto, y la duración de cada etapa.

7.1.1. Metodología

Actualmente, existen muchas metodologías para el desarrollo de *software*. Algunas de estas son:

- **Modelo en cascada [82]:** esta metodología está fundamentada en la realización de las diversas tareas del desarrollo de *software* (análisis, diseño, implementación, pruebas, y evaluación) de forma secuencial. Es una buena opción para sistemas pequeños y bien definidos, pero al aumentar la complejidad del sistema pierde su efectividad, ya que los errores no se descubren hasta el final del desarrollo. Además, no permite cambios en los requisitos durante el desarrollo, perdiendo la capacidad de reacción a cambios en el entorno y necesidades. Debido a esto, en la actualidad, es una metodología poco utilizada y no recomendada para sistemas complejos.

- **Modelo en espiral [83]:** este es un modelo iterativo en el que el desarrollo se fragmenta en varias iteraciones, durante las que se realizan prototipos cada vez más complejos que se utilizan para orientar el desarrollo. Esto permite explorar los requisitos y encontrar problemas más rápidamente, por lo que es una buena opción para proyectos complejos en los que los requisitos pueden no estar bien definidos inicialmente.

Teniendo esto en cuenta, se ha elegido utilizar el modelo en espiral [83], ya que es más flexible a los cambios y permite corregir errores en elementos ya desarrollados en anteriores iteraciones.

7.1.2. Ciclo de vida

El ciclo de vida de esta metodología (representado en la Figura 7.1) se puede dividir en cuatro etapas que se realizan en cada una de las iteraciones del desarrollo. Estas etapas son:

1. **Planificación:** se determinan los requisitos de usuario, que se utilizan para decidir los objetivos de la iteración.
2. **Análisis:** se analizan los requisitos de usuario para identificar los posibles riesgos, y se diseñan las pruebas.
3. **Desarrollo y pruebas:** se diseña e implementa el sistema, y se realizan las pruebas.
4. **Evaluación:** el cliente evalúa el sistema y aporta *feedback*. Si este aprueba el sistema, se continúa con la siguiente iteración, y si no, se corrigen los problemas antes de continuar.

El desarrollo del proyecto estará formado por cinco iteraciones:

I. **Gestor de la arquitectura:** esta etapa se centra en la implementación del 'Gestor de la arquitectura'. El objetivo de esta etapa es permitir el procesado de la definición de una ISA para su posterior uso.

II. **Parser:** esta etapa se centra en la implementación del analizador sintáctico. El objetivo de esta etapa es permitir el procesado de un código ensamblador, extrayendo un AST.

III. **Analizador semántico:** esta etapa se centra en la implementación del analizador semántico y su conexión con los componentes anteriores. El objetivo de esta etapa es permitir compilar programas, obteniendo un sistema completamente funcional.

IV. **Integración con CREATOR:** esta etapa se centra en integrar el sistema con CREATOR. Su objetivo es permitir el uso del sistema dentro de CREATOR.

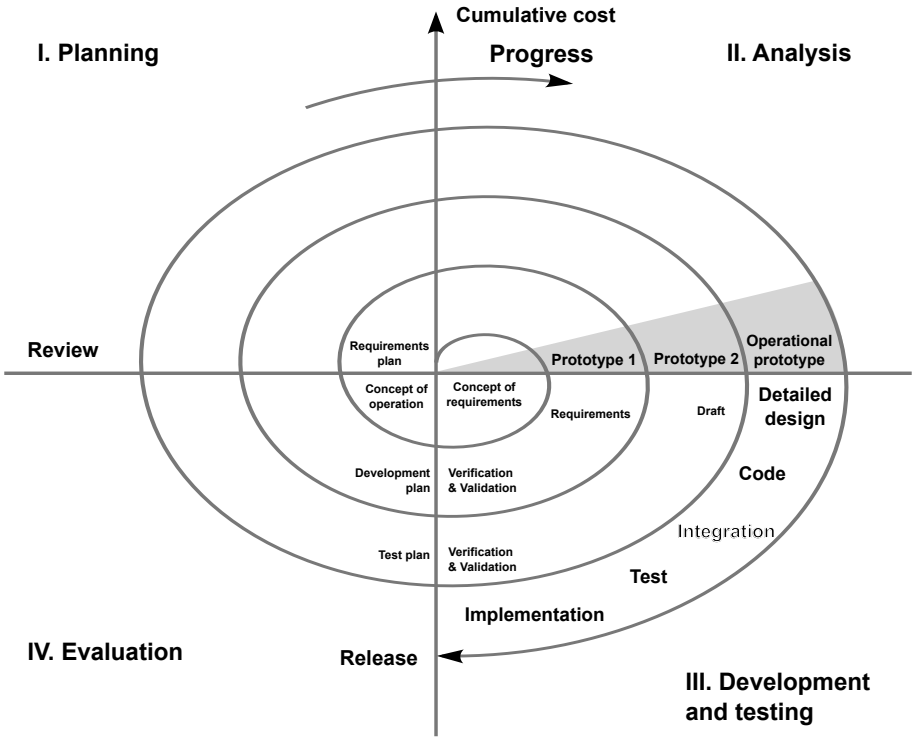

Fig. 7.1. Ciclo de vida del modelo en espiral

V. **Pulido:** esta etapa se centra en implementar las funcionalidades restantes de baja prioridad, que, aunque no son esenciales para el funcionamiento del sistema, mejoran su calidad y dan valor añadido.

7.1.3. Tiempo estimado

La planificación de tiempo del proyecto se realizó con un diagrama de Gantt [84] (Figura 7.2). Este diagrama muestra el tiempo dedicado a cada tarea del desarrollo del sistema, incluyendo todas las iteraciones realizadas y sus etapas. Las etapas de cada una de las iteraciones son las descritas en la Subsección 7.1.2, *Ciclo de vida*, con una etapa adicional de documentación durante la que se realizará este documento. Además, se ha añadido una tarea adicional ("Memoria") para terminar este documento.

El proyecto tuvo una duración final de 12 meses, con una media de 60 horas mensuales (sin contar días no lectivos y vacaciones). Debido a esto, el tiempo total dedicado al proyecto ha sido de 720 horas.

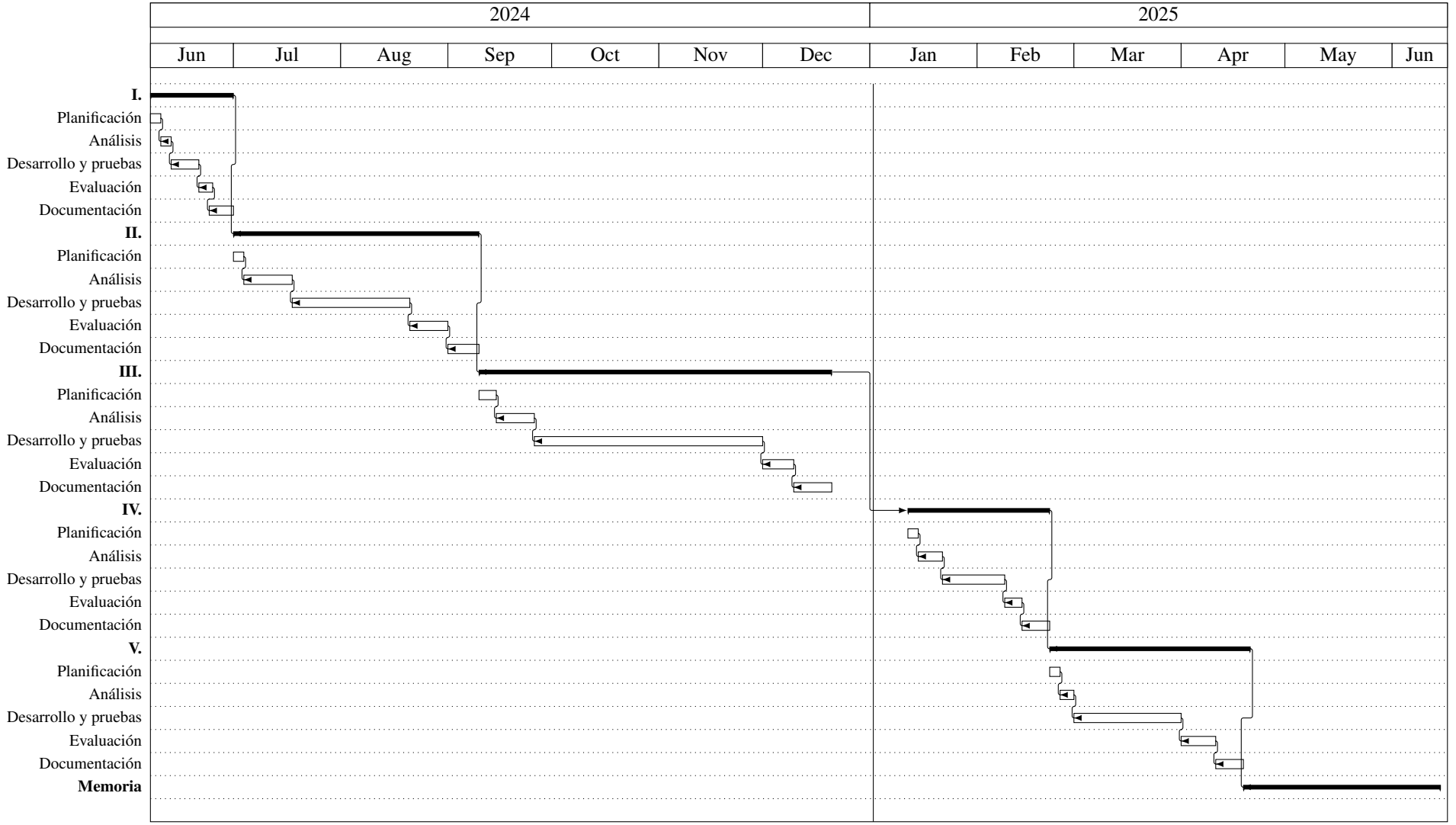

Fig. 7.2. Diagrama de Gantt

7.2. Presupuesto

Esta sección explicará el presupuesto del proyecto, basado en la planificación de tiempo descrita en la anterior sección. Primero, en la Subsección 7.2.1, *Coste del proyecto*, se detallará el coste del proyecto, y, tras esto, en la Subsección 7.2.2, *Oferta del proyecto*, se expondrá la oferta presentada al cliente.

7.2.1. Coste del proyecto

La Tabla 7.1 contiene un resumen de las características del proyecto y el presupuesto total.

TABLA 7.1
INFORMACIÓN DEL PROYECTO

Título	*Desarrollo de un Compilador Genérico de Lenguaje Ensamblador para el Simulador CREATOR*
Autor	Álvaro Guerrero Espinosa
Departamento	Departamento de Informática
Fecha de inicio	7 de junio del 2024
Fecha de fin	16 de junio del 2025
Duración	12 meses
Presupuesto total	48.132,99 €

Los costes se dividen en costes directos (asociados con el personal y equipamiento) e indirectos (con una influencia indirecta en el proyecto). Estos costes no incluirán impuestos, ya que esos se incluyen en *Resumen de costes*.

Costes directos

Los costes directos son los costes relacionados directamente con el desarrollo del proyecto. Se pueden dividir en costes de personal, que dependen de la cualificación, experiencia, y ubicación de cada miembro, y costes de equipamiento, asociados a las herramientas utilizadas durante el desarrollo.

Los costes de personal se pueden dividir en cuatro roles:

- **Jefe de proyecto:** gestiona la planificación del proyecto, y aporta *feedback* sobre el desarrollo.
- **Analista:** analiza los requisitos de usuario, realiza la arquitectura del sistema, y escribe la documentación.
- **Programador:** implementa las funcionalidades del sistema.

- **Tester:** diseña y realiza las pruebas de las funcionalidades del sistema.

El tutor realizó el rol de gestor del proyecto, mientras que el estudiante realizó los otros roles. La Tabla 7.2 muestra los costes del personal totales y para cada rol.

TABLA 7.2
COSTES DE PERSONAL

Rol	Horas	Coste por hora (€/h)	Total (€)
Jefe de proyecto	60 h	60,00	3.600,00
Analista	240 h	45,00	10.800,00
Programador	270 h	35,00	9.450,00
Tester	150 h	30,00	4.500,00
Total	720 h		**28.350,00 €**

Los costes de equipamiento están asociados a la compra y uso del equipamiento, incluyendo *software* y *hardware*. Con respecto al *software*, todas las herramientas utilizadas fueron FOSS y, por lo tanto, no tienen coste asociado. El coste de cada equipo de *hardware* se calcula teniendo en cuenta el tiempo que este se utiliza para el proyecto con la siguiente fórmula:

$$c = \frac{C \cdot t \cdot p}{a} \tag{7.1}$$

Donde:

- c: coste amortizado.
- C: coste del equipamiento.
- t: tiempo durante el que se utiliza el equipamiento para el proyecto.
- p: porcentaje del tiempo total que se utilizó para el proyecto.
- a: tiempo de amortización.

La Tabla 7.3 muestra el coste de cada equipamiento y el coste total de equipamiento.

Costes indirectos

Los costes indirectos son aquellos que tienen una influencia indirecta en el proyecto y no pueden ser asignados a un producto específico, como el consumo eléctrico, la conexión a internet, o el transporte.

Para el consumo el eléctrico, se asume que el portátil consume 65 W de media, el PC de sobremesa 300 W, y el monitor y ratón 20 W. Además, de las 720 horas del proyecto,

TABLA 7.3
COSTES DE EQUIPAMIENTO

Objeto	**Coste** (C)	**Uso** (p)	**Dedicación** (t)	**Amortización** (a)	**Coste amortizado** (c)
PC sobremesa	999,00 €	30 %	12 meses	60 meses	59,94 €
Portátil	749,00 €	60 %	12 meses	60 meses	89,88 €
Monitor	149,99 €	40 %	12 meses	48 meses	15,00 €
Ratón	39,99 €	30 %	12 meses	36 meses	4,00 €
Cable HDMI	4,99 €	40 %	12 meses	48 meses	0,50 €
Software	0,00 €	60 %	12 meses	120 meses	0,00 €
Total	1.942,97 €				**169,32 €**

se ha utilizado el portátil el 70 % del tiempo, y el monitor, ratón, y PC de sobremesa el 30 % restante. Teniendo este en cuenta, la energía total utilizada es de $(65W \cdot 0.7 + 320 \cdot 0.3) \cdot 720h = 101.880Wh$.

La conexión a internet es un plan de 1 gbps de fibra óptica, con un precio de 65,00 €/mes. Esta conexión se comparte por 4 personas y solo una de ellas participa en el proyecto, por lo que el coste para el proyecto es un cuarto de eso.

La tabla Tabla 7.4 contiene los costes indirectos totales del proyecto.

TABLA 7.4
COSTES INDIRECTOS

Recurso	**Coste unitario**	**Unidades**	**Total (€)**
Electricidad	0,15 €/kWh	101.880 Wh	15,28
Internet	16,25 €/mes	12 months	195,00
Transporte	8 €/mes	12 months	96,00
Total			**306,28 €**

Resumen de costes

La Tabla 7.5 contiene un resumen de los costes del proyecto, incluyendo los costes directos e indirectos.

TABLA 7.5
RESUMEN DE COSTES

Personal	28.350,00 €
Equipamiento	169,32 €
Costes indirectos	306,28 €
Total	**28.825,60 €**

7.2.2. Oferta del proyecto

La Tabla 7.6 detalla la oferta del proyecto. En esta oferta se incluyen unos riesgos esperados del 20 %, unos beneficios del 15 %, y los impuestos del IVA, que es del 21 % en España. Teniendo en cuenta esto, el coste final del proyecto es de **48.132,99 € (Cuarenta y ocho mil ciento treinta y dos euros con noventa y nueve céntimos)**.

TABLA 7.6
OFERTA DEL PROYECTO

Concepto	Incremento (%)	Valor parcial (€)	Coste agregado (€)
Coste del proyecto	–	28.825,60	28.825,60
Riesgos	20	5.765,12	34.590,72
Beneficios	15	5.188,61	39.779,33
IVA	21	8.353,66	48.132,99
Total	67		**48.132,99 €**

7.3. Marco regulador

En esta sección se expondrán las restricciones legales que se aplican al proyecto, y está dividido en tres subsecciones. La primera (Subsección 7.3.1, *Legislación*), explicará las leyes y regulaciones aplicables al proyecto. La segunda (Subsección 7.3.2, *Estándares técnicos*), hará un resumen de los estándares técnicos utilizados en el proyecto. Por último, la tercera (Subsección 7.3.3, *Licencias*), indicará las licencias bajo las que se distribuye el sistema y sus componentes.

7.3.1. Legislación

El sistema se ejecuta localmente, y, aunque se puede utilizar en un entorno web, se ejecuta exclusivamente en el navegador cliente. Además, el sistema no utiliza ni transmite datos personales, por lo que no se aplica ninguna ley de protección de datos como la Ley Orgánica de Protección de Datos Personales y Garantía de los Derechos Digitales (en adelante, "LOPDGDD"). Aunque el código ensamblador que procesa el sistema se podría considerar propiedad intelectual, este no se almacena ni se envía, por lo que no se aplica ninguna regulación. Además, no hay ningún riesgo involucrado en la ejecución del sistema, por lo que no hay ninguna otra regulación que se aplique al sistema.

7.3.2. Estándares técnicos

El sistema utiliza usa los siguientes estándares técnicos:

- ISO/IEC 21778:2017 [49], el estándar del formato JSON utilizado para la definición de las ISAs.
- ISO/IEC 10646:2020 [13], el estándar de la codificación UTF-8 usada en las cadenas de caracteres.
- La especificación de Wasm [85], utilizado para ejecutar el compilador en un entorno web.

7.3.3. Licencias

El sistema utiliza y redistribuye múltiples bibliotecas externas:

- *serde* [69], *serde_json* [70], *ariadne* [71], *chumsky* [26], *regex* [72], *once_cell* [73] *num-bigint* [74], *num-traits* [75], *wasm-bindgen* [76], y *ansi-to-html* [78] utilizan la licencia MIT [86], que permite utilizar, copiar, modificar, publicar, y redistribuir el código sin restricciones.
- *self_cell* [77] utiliza la licencia Apache 2.0 [87], que permite utilizar, copiar, modificar, publicar, y redistribuir el código sin restricciones.

El sistema desarrollado utiliza la licencia GNU Lesser General Public License (LGPL) versión 3 [88], ya que permite el uso, modificación, y redistribución del código de forma libre con la única restricción de que las modificaciones se tienen que publicar, asegurando que el sistema continúa siendo abierto. Aunque también se consideró la licencia GNU General Public License (GPL) versión 3 [10], con características similares, se decidió utilizar la licencia LGPL para ser compatible con CREATOR, que se distribuye bajo esta misma licencia.

7.4. Entorno socioeconómico

Históricamente, la gran mayoría de los computadores de escritorio y servidores han utilizado las arquitecturas x86 o x86-64. Estas arquitecturas ofrecen un alto rendimiento, pero tienen una mala eficiencia energética y un alto coste de fabricación debido a su gran complejidad. Debido a esto, el mercado móvil y de dispositivos del Internet de las Cosas (IoT) utilizan otras arquitecturas como ARM o RISC-V que, aunque ofrecen un peor rendimiento, son mucho más simples, gracias a lo cual pueden lograr una mejor eficiencia energética y coste de producción.

Recientemente, sin embargo, se ha descubierto que arquitecturas simples como ARM pueden lograr un rendimiento similar a x86-64. Además, a diferencia de x86-64, cuyas licencias para diseñar procesadores son difíciles de conseguir por estar controladas por Intel y AMD, obtener una licencia para ARM es mucho más fácil. Esto es debido a que

la venta de estas licencias es una parte importante del negocio de Arm Holdings, la empresa propietaria de ARM. La capacidad de diseñar procesadores propios también ayuda a reducir los costes de fabricación y operación al eliminar intermediarios, y ofrece una gran flexibilidad al poder crear procesadores para tareas específicas. Debido a esto, recientemente, grandes empresas como Amazon o Apple han adoptado procesadores con diseños propios basados en ARM en muchos de sus productos, incluyendo computadores de escritorio y servidores.

Esto se ve influenciado por la reciente inestabilidad política y económica de Estados Unidos con políticas como los aranceles. Mientras que Intel y AMD son empresas americanas y, por lo tanto, se ven afectadas por las políticas de Estados Unidos, Arm Holdings es una empresa inglesa sin estos problemas.

En este contexto surge RISC-V como una arquitectura completamente abierta que no requiere de licencias para el diseño de procesadores. Esto reduce mucho más su coste de fabricación al permitir a cualquier empresa diseñar procesadores con esta arquitectura, y elimina muchos problemas políticos. El diseño modular de RISC-V también fomenta el desarrollo de extensiones para tareas específicas, que pueden lograr un mayor rendimiento y eficiencia energética. Sin embargo, su adopción fuera de procesadores para IoT está siendo muy lenta, posiblemente debido a que al ser una arquitectura relativamente reciente, existe poco *software* y se tiene poca experiencia en su uso y diseño.

Esto lleva a la necesidad de desarrollar buenos simuladores que permitan desarrollar y aprender a utilizar nuevas arquitecturas como RISC-V. Como ya se explicó en el Capítulo 1, *Introducción*, el proyecto se ha integrado en un simulador FOSS que contribuye al aprendizaje del lenguaje ensamblador. Este simulador se utiliza actualmente en múltiples universidades entre las que se incluyen la Universidad Carlos III de Madrid, la Universidad de Castilla-La Mancha, la Universidad de León, la Universidad de Almería, y la Universidad Sureste del Estado de Missouri. El proyecto desarrollado ayudará a los estudiantes de estas universidades a aprender el lenguaje ensamblador.

CAPÍTULO 8

CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se presentarán las conclusiones obtenidas con la realización del proyecto (Sección 8.1, *Conclusiones del proyecto*), estudiando los objetivos propuestos en el Capítulo 1, *Introducción*; y las conclusiones personales (Sección 8.2, *Conclusiones personales*), indicando los conocimientos adquiridos y las asignaturas relacionadas con el proyecto. Tras esto, la Sección 8.3, *Contribuciones adicionales* explicará contribuciones adicionales realizadas durante el desarrollo del proyecto que no están directamente relacionadas con el mismo. Finalmente, la Sección 8.4, *Trabajos futuros* indicará trabajos futuros que se podrían realizar para mejorar el sistema desarrollado.

8.1. Conclusiones del proyecto

En este documento se ha descrito el análisis, diseño, e implementación de un compilador de lenguaje ensamblador genérico, robusto, y flexible. El sistema desarrollado es FOSS, y está centrado en tener muchas funcionalidades y generar buenos mensajes de error que ayuden a los usuarios aprender el lenguaje ensamblador.

Como se explicó en la Sección 1.2, *Objetivos*, el objetivo principal del proyecto era crear un compilador capaz de reemplazar al actualmente utilizado por CREATOR, lo cual se ha logrado. Además, también se han logrado todos los objetivos secundarios:

- **O1:** El sistema permite definir las instrucciones permitidas en la ISA, y utilizar estas en los programas.
- **O2:** El sistema permite compilar programas según la definición de una ISA.
- **O3:** Los mensajes de error generados por el sistema incluyen toda la información

relevante para comprender la causa del problema y cómo solucionarlo, incluyendo notas con información adicional relevante para el error y mensajes de ayuda con posibles soluciones.

- **O4:** El sistema es compatible con todas las definiciones de ISAs creadas previamente para CREATOR.

- **O5:** El sistema tiene soporte para nuevas funcionalidades avanzadas como el uso de expresiones aritméticas y etiquetas como valores. Esto permite, por ejemplo, definir los vectores de interrupciones utilizados en la asignatura *Sistemas Operativos*.

La principal dificultad encontrada durante el desarrollo del sistema fue el analizador semántico y su gestión de las referencias hacia delante. Como se explicó en la Subsección 4.1.4, *Análisis semántico*, se consideraron múltiples opciones para realizar esto antes de encontrar una buena solución del problema. Además, como se puede ver en el diagrama de Gantt (Figura 7.2), esto requirió mucho tiempo. Un mayor conocimiento sobre ensambladores podría haber reducido este tiempo.

8.2. Conclusiones personales

La realización de este proyecto me ha permitido aplicar los conocimientos adquiridos durante el grado, además de aprender nuevas herramientas y tecnologías que no había tenido la oportunidad de utilizar hasta ahora.

La principal fuente de información para este proyecto ha sido la asignatura *Procesadores del Lenguaje*, ya que me enseñó la estructura, componentes, y funcionamiento de un compilador. Otras asignaturas que han sido fundamentales para su desarrollo han sido *Estructura de Computadores* y *Arquitectura de Computadores*, en las que se explica el funcionamiento de un computador y el lenguaje ensamblador. Estas tres asignaturas son de las que más me han gustado de la carrera. Otras asignaturas que me ayudaron para este proyecto fueron *Programación*, *Estructura de Datos y Algoritmos*, y *Teoría de Autómatas y Lenguajes Formales*.

He aprendido mucho con la realización de este proyecto, tanto con la implementación del *software* como escribiendo este documento. Aunque ya conocía Rust antes de empezar y había escrito algunos programas pequeños, no tenía demasiada experiencia realizando programas más grandes y complejos, y tampoco tenía ninguna experiencia utilizando Wasm. Escribí este documento en LaTeX, y, aunque ya tenía cierta experiencia utilizando esta herramienta para otros documentos, no había utilizado ninguna de sus funcionalidades avanzadas. Escribir este documento me ha permitido aprender mucho más de esta herramienta, y he podido utilizar muchas de sus funcionalidades avanzadas.

8.3. Contribuciones adicionales

Como ya se ha mencionado, este documento está escrito en LaTeX [89]. Su código fuente se encuentra disponible en https://github.com/ALVAROPING1/TFG. Este código fuente utiliza una plantilla de LaTeX desarrollada por Luis Daniel Casais Mezquida [90]. Esta plantilla contiene una clase de LaTeX que define el formato del documento, siguiendo la normativa de la universidad [91].

Durante la escritura de este documento, se encontraron y corrigieron varios fallos en esta plantilla, particularmente en su uso para documentos en español. Estas aportaciones se añadieron a la plantilla para permitir su uso por otros estudiantes. Además, el código fuente de esta memoria puede servir como un ejemplo de uso de la plantilla, para ayudar a futuros estudiantes que quieran utilizarla.

Este documento también utiliza el paquete SRS, originalmente desarrollado por Javier López Gómez y modificado por Luis Daniel Casais Mezquida [92]. Este paquete automatiza el proceso de generación de tablas para el proceso de ingeniería de *software*, particularmente tablas de requisitos, casos de uso, y componentes, y matrices de trazabilidad entre estos elementos. Durante la realización de este documento, se modificó este paquete para incluir varias nuevas funcionalidades: uso de identificadores nominales para evitar problemas de referencias al reordenarlos, gestión de errores para identificadores desconocidos o repetidos, y posibilidad de incluir múltiples orígenes en todos los elementos. Estos cambios se contribuyeron al paquete original para permitir su uso por otros estudiantes de ingeniería informática.

8.4. Trabajos futuros

Este proyecto implementa un compilador de ensamblador, pero se podría mejorar de diversas formas:

- Se podría añadir soporte para macros.
- Se podría añadir soporte para la compilación condicional.
- Se podría añadir soporte para la definición de constantes (similar a la directiva `.set` de GAS).
- La mayoría de compiladores implementan estrategias de recuperación de errores para permitir detectar la mayor cantidad posible de errores en cada compilación. Como se mencionó en la Subsección 4.2.2, *Analizador semántico*, se podrían añadir estas estrategias para poder mostrar al usuario la mayor cantidad posible de problemas en su código.

- El sistema actualmente soporta operadores aritméticos y bit a bit básicos, pero se podrían añadir más operadores como desplazamientos, comparaciones, o booleanos.

Además, también se podrían mejorar algunos aspectos de la implementación del sistema:

- Como se mencionó en la Subsección 4.1.3, *Análisis sintáctico*, crear un analizador sintáctico descendente recursivo a mano podría permitir una mayor velocidad.
- Por compatibilidad con definiciones antiguas de ISAs, el sistema utiliza una expansión de pseudo-instrucción muy similar a la implementada por el compilador antiguo, y esto evita el uso de referencias hacia delante en ciertos casos. Si se acepta modificar estas definiciones, se podría rehacer este sistema para corregir este fallo.
- Se podría investigar el uso de `no_std` [93] y otros métodos para reducir el tamaño del fichero ejecutable, que es un factor importante para aplicaciones web, ya que se tienen que transmitir por la red antes de su uso.
- Actualmente, cuando se evalúa una expresión aritmética a un número flotante, todos los valores se convierten a números flotantes. Para aumentar la precisión, se podrían mantener los números enteros como bigints hasta que se les aplique una operación con un número flotante

BIBLIOGRAFÍA

[1] D. Camarmas Alonso, F. García Carballeira, E. Del Pozo Puñal y A. Calderón Mateos. «CREATOR – didaCtic and geneRic assEmbly progrAmming simulaTOR.» (2019), [En línea]. Disponible en: https://creatorsim.github.io/ (Acceso: 04-05-2025).

[2] RISC-V International. «Spike RISC-V ISA Simulator.» (2019), [En línea]. Disponible en: https://github.com/riscv-software-src/riscv-isa-sim (Acceso: 04-05-2025).

[3] The GDB developers. «GDB: The GNU Project Debugger.» (2025), [En línea]. Disponible en: https://www.sourceware.org/gdb/ (Acceso: 04-05-2025).

[4] W. Song, *Kite: An Architecture Simulator for RISC-V Instruction Set*, Yonsei University, [Online], Available: https://casl.yonsei.ac.kr/kite, mayo de 2019.

[5] Computer Architecture and Systems Lab, Yonsei University. «Kite: Architecture Simulator for RISC-V Instruction Set.» (2019), [En línea]. Disponible en: https://github.com/yonseicasl/Kite (Acceso: 10-05-2025).

[6] D. A. Patterson y J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware and Software Interface* (Issn Series), 1st ed. 2018.

[7] REMS Project. «Sail architecture definition language.» (2019), [En línea]. Disponible en: https://github.com/rems-project/sail/ (Acceso: 04-05-2025).

[8] «Linux assemblers: A comparison of GAS and NASM.» (2007), [En línea]. Disponible en: https://web.archive.org/web/20180330012248/https://www.ibm.com/developerworks/linux/library/l-gas-nasm/index.html (Acceso: 06-05-2025).

[9] «GNU Binutils,» GNU. (2025), [En línea]. Disponible en: https://www.gnu.org/software/binutils/ (Acceso: 09-02-2025).

[10] Open Source Initiative. «GNU General Public License.» ver. 3.0. (2007), [En línea]. Disponible en: https://opensource.org/license/gpl-3-0 (Acceso: 03-05-2025).

[11] «Using as,» Free Software Foundation. (2024), [En línea]. Disponible en: https://sourceware.org/binutils/docs/as.html (Acceso: 01-02-2025).

[12] The GCC Team. «GCC, the GNU Compiler Collection.» (1987), [En línea]. Disponible en: https://gcc.gnu.org/ (Acceso: 03-05-2025).

[13] «Information technology – Universal Coded Character Set (UCS),» International Organization for Standardization, International Standard ISO/IEC 10646:2020, 2020.

[14] «Tiny C Compiler.» (2018), [En línea]. Disponible en: https://bellard.org/tcc/ (Acceso: 07-05-2025).

[15] «Re: [Tinycc-devel] Some benchmarks on RPi.» (2013), [En línea]. Disponible en: https://lists.nongnu.org/archive/html/tinycc-devel/2013-02/msg00043.html (Acceso: 07-05-2025).

[16] «Tiny C Compiler Reference Documentation.» (2018), [En línea]. Disponible en: https://bellard.org/tcc/tcc-doc.html (Acceso: 07-05-2025).

[17] «[Tinycc-devel] TCC version 0.9.27 is out.» (2017), [En línea]. Disponible en: https://lists.nongnu.org/archive/html/tinycc-devel/2017-12/msg00015.html (Acceso: 07-05-2025).

[18] «[Tinycc-devel] RISC-V support.» (2019), [En línea]. Disponible en: https://lists.nongnu.org/archive/html/tinycc-devel/2019-09/msg00000.html (Acceso: 07-05-2025).

[19] «NASM.» (1996), [En línea]. Disponible en: https://www.nasm.us/ (Acceso: 06-05-2025).

[20] Open Source Initiative. «The 2-Clause BSD License.» (), [En línea]. Disponible en: https://opensource.org/license/BSD-2-Clause (Acceso: 07-05-2025).

[21] «NASM – The Netwide Assembler.» (2024), [En línea]. Disponible en: https://www.nasm.us/xdoc/2.16.03/html/nasmdoc0.html (Acceso: 06-05-2025).

[22] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Addison-Wesley, 2006.

[23] T. J. Parr, *Language implementation patterns: create your own domain-specific and general programming languages* (Pragmatic programmers), 1st edition. 2010.

[24] D. Holden. «You could have invented Parser Combinators.» (2014), [En línea]. Disponible en: https://theorangeduck.com/page/you-could-have-invented-parser-combinators (Acceso: 07-01-2025).

[25] D. Leijen, P. Martini y A. Latter. «Parsec.» (2006), [En línea]. Disponible en: https://github.com/haskell/parsec (Acceso: 07-01-2025).

[26] J. Barretto. «Chumsky.» (2021), [En línea]. Disponible en: https://github.com/zesterer/chumsky (Acceso: 07-01-2025).

[27] J. Liu. «pom.» (2017), [En línea]. Disponible en: https://github.com/J-F-Liu/pom (Acceso: 07-01-2025).

[28] LLVM Developer Group. «Clang – Features and Goals.» (), [En línea]. Disponible en: https://clang.llvm.org/features.html (Acceso: 07-01-2025).

[29] A. Kladov. «Resilient LL Parsing Tutorial.» (2023), [En línea]. Disponible en: https://matklad.github.io/2023/05/21/resilient-ll-parsing-tutorial.html (Acceso: 11-05-2025).

[30] P. Eaton. «Parser generators vs. handwritten parsers: surveying major language implementations in 2021.» (2021), [En línea]. Disponible en: https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html (Acceso: 07-01-2025).

[31] S. Harwell. «Operator precedence parser.» (2008), [En línea]. Disponible en: https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687077/Operator+precedence+parser (Acceso: 29-04-2025).

[32] V. R. Pratt, «Top down operator precedence,» en *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ép. POPL '73, Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 41-51. DOI: 10.1145/512927.512931.

[33] A. Kladov. «Simple but Powerful Pratt Parsing.» (2020), [En línea]. Disponible en: https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html (Acceso: 29-04-2025).

[34] M. Tomita, *Generalized LR Parsing*, 1st ed. Springer New York, 2012.

[35] J. R. Levine, D. Brown y T. Mason, *Lex & yacc*, 2nd ed. O'Reilly, 1992.

[36] J. R. Levine, *Flex & bison*, 1st ed. O'Reilly Media, 2009.

[37] M. Brunsfeld. «Tree-sitter.» (2018), [En línea]. Disponible en: https://tree-sitter.github.io/tree-sitter/ (Acceso: 07-01-2025).

[38] S. Klabnik y C. Nichols, *The Rust programming language*, 2nd ed. No Starch Press, 2023.

[39] LLVM Developer Group. «Clang: a C language family frontend for LLVM.» (2007), [En línea]. Disponible en: https://clang.llvm.org/ (Acceso: 03-05-2025).

[40] The Conan Team. «Conan, software package manager for C and C++ developers.» (2018), [En línea]. Disponible en: https://conan.io/ (Acceso: 03-05-2025).

[41] S. Thénault. «Pylint – Code analysis for python.» (2001), [En línea]. Disponible en: https://www.pylint.org/ (Acceso: 03-05-2025).

[42] L. Langa. «Black – The Uncompromising Code Formatter.» (2018), [En línea]. Disponible en: https://github.com/psf/black (Acceso: 03-05-2025).

[43] «TypeScript,» Microsoft. (2014), [En línea]. Disponible en: https://github.com/microsoft/TypeScript (Acceso: 03-05-2025).

[44] «A 10x Faster TypeScript,» Microsoft. (2025), [En línea]. Disponible en: https://devblogs.microsoft.com/typescript/typescript-native-port/ (Acceso: 04-04-2025).

[45] «Ruff, An extremely fast Python linter,» Astral. (2023), [En línea]. Disponible en: https://astral.sh/ruff (Acceso: 03-05-2025).

[46] «Ruff, An extremely fast Python package and project manager,» Astral. (2024), [En línea]. Disponible en: https://github.com/astral-sh/uv (Acceso: 03-05-2025).

[47] «IEEE Guide for Software Requirements Specifications,» Institute of Electrical y Electronics Engineers, IEEE Std. 830-1984, 1984. DOI: 10.1109/IEEESTD.1984.119205.

[48] «IEEE Standard for Floating-Point Arithmetic,» Institute of Electrical y Electronics Engineers, IEEE Std. 754-2019, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[49] «Information technology – The JSON data interchange syntax,» International Organization for Standardization, International Standard ISO/IEC 21778:2017, 2017.

[50] «What is Free Software?» Free Software Foundation. (2024), [En línea]. Disponible en: https://www.gnu.org/philosophy/free-sw.html (Acceso: 07-02-2025).

[51] «The Open Source Definition,» Open Source Initiative. (2007), [En línea]. Disponible en: https://opensource.org/osd (Acceso: 07-02-2025).

[52] S. Cook et al., «Unified Modeling Language Specification,» ver. 2.5.1, dic. de 2017. [En línea]. Disponible en: https://www.omg.org/spec/UML/2.5.1.

[53] «JavaScript data types and data structures,» MDN Web Docs. (2024), [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures (Acceso: 06-01-2025).

[54] «TypeScript is JavaScript with syntax for types,» Microsoft. (2014), [En línea]. Disponible en: https://www.typescriptlang.org/ (Acceso: 03-05-2025).

[55] A. Haas et al., «Bringing the web up to speed with WebAssembly,» en *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ép. PLDI 2017, vol. 52, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185-200. DOI: 10.1145/3140587.3062363.

[56] B. W. Kernighan y D. M. Ritchie, *The C programming language* (Prentice-Hall software series), 2nd ed. Prentice-Hall, 1988.

[57] B. Stroustrup, *The C++ programming language*, 4th ed. Addison-Wesley, 2013.

[58] S. Fernandez. «A proactive approach to more secure code,» Microsoft. (2019), [En línea]. Disponible en: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (Acceso: 04-01-2025).

[59] D. Hosfelt. «Implications of Rewriting a Browser Component in Rust,» Mozilla. (2019), [En línea]. Disponible en: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/ (Acceso: 04-01-2025).

[60] B. Lord. «The Urgent Need for Memory Safety in Software Products,» Cybersecurity and Infrastructure Security Agency. (2023), [En línea]. Disponible en: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products (Acceso: 04-01-2025).

[61] «Back to the building blocks: A path toward secure and measurable software,» The White House, Government Report, 2024. [En línea]. Disponible en: https://whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (Acceso: 04-01-2025).

[62] P. Akritidis, «Practical memory safety for C,» University of Cambridge, Computer Laboratory, inf. téc. UCAM-CL-TR-798, jun. de 2011. DOI: 10.48456/tr-798. [En línea]. Disponible en: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-798.pdf.

[63] A. Zakai. «Emscripten.» (2011), [En línea]. Disponible en: https://emscripten.org/index.html (Acceso: 04-01-2025).

[64] The Rust and WebAssembly Working Group. «Wasm-pack.» (2018), [En línea]. Disponible en: https://rustwasm.github.io/wasm-pack/ (Acceso: 04-01-2025).

[65] «Information technology – Control functions for coded character sets,» International Organization for Standardization, International Standard ISO/IEC 6429:1992, 1992.

[66] T. Æ. Mogensen, *Basics of compiler design*, Anniversary edition. Torben Ægidius Mogensen, 2010.

[67] L. L. Beck, *System Software: An Introduction to Systems Programming*, 3rd ed. Addison-Wesley, 1997.

[68] J. W. Backus et al., «Revised report on the algorithmic language ALGOL 60,» *Communications of the ACM*, vol. 6, n.º 1, pp. 1-17, ene. de 1963. DOI: 10.1145/366193.366201.

[69] D. Tolnay. «Serde.» (2017), [En línea]. Disponible en: https://serde.rs/ (Acceso: 04-02-2025).

[70] D. Tolnay. «Serde JSON.» (2017), [En línea]. Disponible en: https://github.com/serde-rs/json (Acceso: 04-02-2025).

[71] J. Barretto. «Ariadne.» (2021), [En línea]. Disponible en: https://github.com/zesterer/ariadne (Acceso: 04-02-2025).

[72] The Rust Project Developers. «Regex.» (2021), [En línea]. Disponible en: https://github.com/rust-lang/regex (Acceso: 04-02-2025).

[73] A. Kladov. «OnceCell.» (2019), [En línea]. Disponible en: https://github.com/matklad/once_cell (Acceso: 04-02-2025).

[74] The Rust Project Developers. «num-bigint.» (2016), [En línea]. Disponible en: https://github.com/rust-num/num-bigint (Acceso: 04-02-2025).

[75] The Rust Project Developers. «num-traits.» (2016), [En línea]. Disponible en: https://github.com/rust-num/num-traits (Acceso: 04-02-2025).

[76] The Rust and WebAssembly Working Group. «wasm-bindgen.» (2018), [En línea]. Disponible en: https://github.com/rustwasm/wasm-bindgen (Acceso: 04-02-2025).

[77] L. Bergdoll. «self_cell.» (2023), [En línea]. Disponible en: https://github.com/Voultapher/self_cell (Acceso: 04-02-2025).

[78] L. Stecher. «ansi-to-html.» (2020), [En línea]. Disponible en: https://github.com/Aloso/to-html/tree/main/crates/ansi-to-html (Acceso: 04-02-2025).

[79] The Rust Team. «Rust.» (2012), [En línea]. Disponible en: https://www.rust-lang.org/ (Acceso: 04-02-2025).

[80] «IEEE Standard for System and Software Verification and Validation,» Institute of Electrical y Electronics Engineers, IEEE Std. 1012-2012, 1984. DOI: 10.1109/IEEESTD.2012.6204026.

[81] J. A. V. Jiménez. «ForthV.» (2025), [En línea]. Disponible en: https://codeberg.org/joseaverde/ForthV/ (Acceso: 11-02-2025).

[82] H. D. Benington, «Production of Large Computer Programs,» *Annals of Large Computer Programs*, vol. 5, n.º 4, pp. 350-361, 1983. DOI: 10.1109/MAHC.1983.10102.

[83] B. W. Boehm, «A spiral model of software development and enhancement,» *IEEE Computer*, vol. 21, n.º 5, pp. 61-72, 1988. DOI: 10.1109/2.59.

[84] W. Clark, W. Polakov y F. Trabold, *The Gantt Chart: A Working Tool of Management* (Ronald manufacturing management and administration series). Ronald Press Company, 1922.

[85] World Wide Web Consortium. «WebAssembly Specification.» ver. 2.0. (2025), [En línea]. Disponible en: https://webassembly.github.io/spec/core/ (Acceso: 24-04-2025).

[86] Open Source Initiative. «The MIT License.» (1998), [En línea]. Disponible en: https://opensource.org/license/MIT (Acceso: 24-04-2025).

[87] Open Source Initiative. «Apache License.» ver. 2.0. (2004), [En línea]. Disponible en: https://opensource.org/license/apache-2-0 (Acceso: 24-04-2025).

[88] Open Source Initiative. «GNU Lesser General Public License.» ver. 3.0. (2007), [En línea]. Disponible en: https://opensource.org/license/lgpl-3-0 (Acceso: 24-04-2025).

[89] L. Lamport, *LATEX: A Document Preparation System*. Addison-Wesley, 1986.

[90] L. D. C. Mezquida. «Plantilla de TFG de la Universidad Carlos III de Madrid (IEEE).» (2024), [En línea]. Disponible en: https://github.com/ldcas-uc3m/thesis-template (Acceso: 11-05-2025).

[91] Biblioteca UC3M. «Trabajo de Fin de Grado UC3M: Escribir el TFG.» (2025), [En línea]. Disponible en: https://uc3m.libguides.com/TFG/escribir (Acceso: 11-05-2025).

[92] L. D. C. Mezquida y J. L. Gómez. «srs-latex.» (2024), [En línea]. Disponible en: https://github.com/rajayonin/srs-latex (Acceso: 11-05-2025).

[93] Embedded devices Working Group. «A `no_std` Rust Environment.» (2025), [En línea]. Disponible en: https://docs.rust-embedded.org/book/intro/no-std.html (Acceso: 20-05-2025).

GLOSARIO

A

API

Interfaz de un programa diseñada para ser utilizada por otro programa. 46

AST

Estructura de datos utilizada para representar la estructura de un programa. 62

B

biblioteca

Conjunto de recursos que implementan una funcionalidad y cuya finalidad es ser utilizado por otro programa. 35, 57, 60, 84, *véase* programa

bigint

Número entero de tamaño arbitrariamente grande. 11, 13–15, 124

binding

API que provee código diseñado para permitir que un lenguaje de programación utilice una biblioteca escrita en otro lenguaje. 60, 75, 76, *véase* API & lenguaje de programación

byte nulo

Byte con el valor 0. 33, 83

C

compilación

Proceso de traducción de un programa escrito en un lenguaje de programación a otro. 2, 3, 9, 11–15, 27, 31, 35, 41, 50, 55–57, 59, 60, 66, 69, 73, 112, 121, 123, *véase* lenguaje de programación

compilador

Programa cuyo propósito es compilar otro programa. 1–3, 5, 11, 15, 17–21, 23, 62, 63, 77, 78, 92, 93, 96, 103, 105–110, 119, 121–124, *véase* compilación & programa

computador

Máquina que puede ser programada para llevar a cabo una secuencia de operaciones. 1, 2, 119, 120, 122

código máquina

Código de un computador formado por instrucciones utilizadas para controlar su comportamiento, típicamente mezcladas con datos. 9, *véase* computador

D

directiva de datos

Directiva de ensamblador que especifica datos a añadir en la memoria de datos, con su tipo, tamaño, y alineamiento. 13, 14, 26, 32–35, 41, 73, 81, 83, 87, 88, 92, 99, 100, 102, 107, 109, *véase* directiva de ensamblador & memoria de datos

directiva de ensamblador

Instrucción del ensamblador para realizar una tarea o cambiar un ajuste. 9, 11–13, 32, 81, 106, 107, *véase* ensamblador

distancia de edición

Métrica de cadenas de caracteres que mide la diferencia entre dos secuencias, definida como la cantidad mínima de modificaciones requeridas para transformar una cadena en la otra. Típicamente las operaciones permitidas son inserción, sustitución, o borrado de un carácter, y se puede añadir la transposición de dos caracteres adyacentes. 43

E

ensamblador

Compilador que traduce un programa escrito en lenguaje ensamblador a código máquina. 2, 3, 5, 6, 9–15, 29, 50, 63, 105, 122, *véase* compilador & código máquina

expresión

Elemento sintáctico de un lenguaje de programación que se puede evaluar para obtener un valor. 10–15, 17, 26, 28, 39–41, 44, 66, 72, 87, 88, 100, 109, 122, 124, *véase* lenguaje de programación

expresión regular

Secuencia de caracteres que especifica un patrón de texto. 76

F

formateador de código

Herramienta que aplica un formato específico a un código fuente, típicamente modificando espacios, saltos de línea, e indentación. 19

FOSS

Software disponible bajo una licencia que concede el derecho de utilizar, modificar, y distribuir el software a cualquier persona libre de cargos. 29

G

gramática

Descripción formal de la sintaxis de un lenguaje, basada en el uso de reglas de transformación de secuencias de símbolos. 15–18, 61, 62, 72, *véase* compilador

gramática ambigua

Gramática para la que existe alguna cadena que se puede obtener con varios árboles de sintaxis distintos. 18, *véase* gramática

gramática de precedencia de operadores

Gramática en la que ninguna producción tiene una parte derecha vacía o con varios símbolos no terminales juntos. 17, *véase* gramática

I

instrucción

Orden dada al procesador de un computador para que realice alguna acción. 2, 3, 6, 7, 9, 10, 12, 13, 27, 31, 40, 41, 50, 63, 67, 68, 73, 81, 82, 87, 89, 94, 99–101, 108, 121, *véase* procesador & computador

L

lenguaje de alto nivel

Lenguaje de programación con fuertes abstracciones de los detalles del computador. 1, 2, *véase* lenguaje de programación & computador

lenguaje de bajo nivel

Lenguaje de programación con pocas o ninguna abstracción de los detalles del computador y su ISA. 1, 2, *véase* lenguaje de programación, computador & ISA

lenguaje de programación

Sistema de notación para escribir un programa. 1–3, 5, 8, 14, 15, 17–21, 59, 60, 62, 109, *véase* programa

lenguaje ensamblador

Lenguaje de programación de bajo nivel con una fuerte correspondencia entre las instrucciones en el lenguaje y las instrucciones soportadas por el computador. 1–3, 5, 7, 9, 11–13, 19, 21, 26, 27, 39, 41, 42, 55, 69, 102, 109, 112, 118, 120–123, *véase* lenguaje de programación

lexer

Componente de un compilador encargado de transformar el código a una secuencia de tokens. 72, *véase* compilador & token

linter

Herramienta que realiza un análisis de un código fuente para buscar errores y código sospechoso. 19

M

macro

Regla o patrón que especifica cómo una cierta entrada debe ser convertida a una cadena de reemplazo. 11–15, 123

memoria

Dispositivo de almacenamiento utilizado para almacenar información. 6, 7, 9, 28, 44, 60, 63, 77, 78

memoria de datos

Segmento de la memoria de un computador utilizado para almacenar los datos de un programa. 9, 33, *véase* memoria & computador

memoria de texto

Segmento de la memoria de un computador utilizado para almacenar las instrucciones de un programa. 9, *véase* memoria, computador & instrucción

N

nodo hoja

Nodo de un árbol que no tiene hijos. 16, 63

nodo interno

Nodo de un árbol que tiene hijos. 16, 62

P

palabra

Tamaño de un registro de un computador. 34, *véase* computador & registro

parser

Componente de un compilador encargado de realizar el análisis sintáctico del código. 16–18, 61, 62, 66, 67, 72, 112, *véase* compilador

probador de teoremas

Herramienta que permite la generación automática de demostraciones de teoremas. 9

procesador

Componente de un computador encargado de ejecutar instrucciones y realizar operaciones. 5, 7–9, 59, 119, 120, *véase* instrucción & computador

programa

Secuencia de instrucciones escrita en un lenguaje de programación para que un computador la ejecute. 1, 2, 5–7, 9, 14–16, 27, 35, 41, 50, 56, 57, 60, 62, 66, 67, 69, 73, 81–92, 94, 96–105, 109, 121, 122, *véase* lenguaje de programación & computador

pseudo-instrucción

Instrucción permitida en un ensamblador que se reemplaza por una secuencia de instrucciones durante la compilación. 9, 31, 40, 68, 73, 82, 124, *véase* instrucción & ensamblador

R

referencia hacia delante

Uso de un elemento, típicamente una función o variable, antes de su definición en el código de un programa. 12, 63, 122, 124

registro

Localización de acceso rápido para un procesador con una pequeña cantidad de memoria. 6, 7, 9, 81, *véase* procesador

renderizador

Componente encargado de generar una imagen a partir de datos de entrada. 63, 65

S

seguridad de memoria

Protección contra errores y vulnerabilidades de seguridad causadas por errores en el uso y gestión de la memoria. 60, 61, *véase* memoria

sentencia

Elemento sintáctico de un lenguaje de programación que indica una acción a realizar. 25, 38, 99, *véase* lenguaje de programación

sistema de tipos

Sistema lógico con un conjunto de reglas que asignan un tipo a cada elemento del lenguaje, típicamente variables, expresiones, y funciones. 60

sistema de tipos dinámico

Sistema de tipos en el cual una variable puede tomar valores de diferentes tipos y la comprobación de tipos debe hacerse durante la ejecución del programa. 60, *véase* sistema de tipos

sistema de tipos débil

Sistema de tipos con reglas poco estrictas y muchas conversiones de tipos implícitas. 60, *véase* sistema de tipos

sistema de tipos estático

Sistema de tipos en el cual el tipo de una variable se decide durante la compilación, y esta solo puede tomar valores del tipo correspondiente. 60, 61, *véase* sistema de tipos & compilación

sistema de tipos fuerte

Sistema de tipos con reglas estrictas y pocas conversiones de tipos implícitas, típicamente solo entre diferentes tipos de números. 60, 61, *véase* sistema de tipos

T

tabla de símbolos

Estructura de datos utilizada por un compilador para almacenar la información asociada con cada identificador. 68, 73, 75, 77, *véase* compilador

tiempo de compilación

Tiempo durante el cual un programa se compila. 62, *véase* compilación

token

Secuencia de caracteres con un significado conjunto tratada como una unidad. 15, 63, 66, 67, 72

toolchain

Conjunto de herramientas utilizadas para compilar y desarrollar software. 9, 11, *véase* compilador

V

valor inmediato

Argumento de una instrucción cuyo valor se encuentra codificado en la propia instrucción, en vez de en un registro o memoria. Típicamente números enteros. 40, 87, 100, *véase* instrucción, registro & memoria

SIGLAS

A

API

Application Programming Interface. 46, 55–57, 59, 60, 75, 77, 92, *Glossary:* API

AST

Abstract Syntax Tree. 62, 63, 66–68, 72, 73, 112, *Glossary:* AST

C

CLI

Command-line Interface. 6, 7, 9, 110, *Glossary:* CLI

CSS

Cascading Style Sheets. 61

F

FOSS

Free and Open Source Software. 29, 49, 93, 103, 116, 120, 121, *Glossary:* FOSS

FSF

Free Software Foundation. 49, 93, 103

G

GAS

GNU Assembler. 10–13, 15, 29, 50, 72, 105–110, 123

GPL

GNU General Public License. 119

H

HTML

Hypertext Markup language. 61

I

IoT

Internet de las Cosas. 119, 120

ISA

Instruction Set Architecture. 2, 3, 5, 6, 8, 9, 11, 24, 37, 45, 55–57, 62, 63, 65, 68, 69, 77, 81–92, 94, 96–102, 104, 112, 119–122, 124, *Glossary:* ISA

J

JS

JavaScript. 9, 13, 15, 46, 59–61, 75–77, 92

JSON

JavaScript Object Notation. 9, 45, 75, 77

L

LGPL

GNU Lesser General Public License. 119

O

OSI

Open Source Initiative. 49, 93, 103

T

TS

TypeScript. 20, 60, 61

W

Wasm

WebAssembly. 59, 60, 119, 122