# UNIVERSIDAD ESTATAL DE BOLÍVAR

## FACULTAD DE CIENCIAS ADMINISTRATIVAS, GESTIÓN EMPRESARIAL E INFORMÁTICA

CARRERA DE SOFTWARE

**TRABAJO DE INTEGRACIÓN CURRICULAR**
**PREVIO A LA OBTENCIÓN DEL TÍTULO DE**
**INGENIERO EN SOFTWARE**

**FORMA:** PROYECTO TECNOLÓGICO

**TEMA:**

# LENGUAJE DE PROGRAMACIÓN CENTRADO EN LA EFICIENCIA Y FACILIDAD DE USO PARA PROGRAMADORES.

**AUTOR:**

ISMAEL ISAAC QUIROZ CACHIMUEL

**DIRECTOR:**

ING. DANILO BARRENO NARANJO

**GUARANDA – ECUADOR**

**2025**

## TEMA DEL PROYECTO TECNOLÓGICO

LENGUAJE DE PROGRAMACIÓN CENTRADO EN LA EFICIENCIA Y FACILIDAD DE USO PARA PROGRAMADORES.

## DEDICATORIA

Dedico este trabajo a mi familia, cuyo apoyo incondicional me ha motivado a perseverar en este proyecto, y a mis profesores, quienes han guiado mi formación como ingeniero en software.

# AGRADECIMIENTO

Deseo expresar mi gratitud a quienes facilitaron la culminación de este proyecto de titulación, el cual constituye una creación de mi autoría exclusiva, desarrollada de manera independiente y autónoma.

Extiendo mi reconocimiento al Ing. Danilo Barreno Naranjo, director de esta tesis, por su acompañamiento en la gestión metodológica del proyecto. Asimismo, agradezco al Ing. Darwin Carrión Buenaño por las directrices impartidas en la asignatura de Trabajo de Titulación, las cuales fueron fundamentales para estructurar la orientación de este documento.

Finalmente, un agradecimiento especial a mis pares académicos, la Ing. Galuth García Camacho y la Ing. Maricela Espín Morejón, por su valiosa retroalimentación y apoyo en la revisión de esta documentación, lo que permitió alcanzar una redacción más clara y profesional.

# CERTIFICADO DE VALIDACIÓN

Ing. Danilo Barreno, Ing. Galuth García e Ing. Maricela Espín, en su orden Director y Pares Académicos del Trabajo de Integración Curricular "LENGUAJE DE PROGRAMACIÓN CENTRADO EN LA EFICIENCIA Y FACILIDAD DE USO PARA PROGRAMADORES" desarrollado por el Sr. Quiroz Cachimuel Ismael Isaac.

## CERTIFICAN

Que, luego de revisado el Trabajo de Integración Curricular en su totalidad, cumple con las exigencias académicas de la carrera SOFTWARE, por lo tanto, autorizamos su presentación y defensa.

Guaranda, 04 del 03 del 2026

Firmado electrónicamente por:
DANILO GEOVANNY
BARRENO NARANJO
Validar únicamente con FirmaEC

Ing. Danilo Barreno
**Director**

Firmado electrónicamente por:
GALUTH IRENE GARCIA
CAMACHO
Validar únicamente con FirmaEC

Ing. Galuth García
**Par Académico**

Firmado electrónicamente por:
MARICELA ARACELI
ESPIN MOREJON
Validar únicamente con FirmaEC

Ing. Maricela Espín
**Par Académico**

# DERECHOS DE AUTOR

Yo, **QUIROZ CACHIMUEL ISMAEL ISAAC**, portador de la Cédula de Identidad No. **1050315512**, en calidad de autor y titular de los derechos morales y patrimoniales del Trabajo de Titulación: **LENGUAJE DE PROGRAMACIÓN CENTRADO EN LA EFICIENCIA Y FACILIDAD DE USO PARA PROGRAMADORES**, modalidad **PROYECTO TECNOLÒGICO**, de conformidad con el Art. 114 del CÓDIGO ORGÁNICO DE LA ECONOMÍA SOCIAL DE LOS CONOCIMIENTOS, CREATIVIDAD E INNOVACIÓN, concedo a favor de la Universidad Estatal de Bolívar, una licencia gratuita, intransferible y no exclusiva para el uso no comercial de la obra, con fines estrictamente académicos. Conservo a mi favor todos los derechos de autor sobre la obra, establecidos en la normativa citada.

Asimismo, autorizo a la Universidad Estatal de Bolívar, para que realice la digitalización y publicación de este trabajo de titulación en el Repositorio Digital, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior.

El autor declara que la obra objeto de la presente autorización es original en su forma de expresión y no infringe el derecho de autor de terceros, asumiendo la responsabilidad por cualquier reclamación que pudiera presentarse por esta causa y liberando a la Universidad de toda responsabilidad.

Quiroz Cachimuel Ismael Isaac

**INDICE DE CONTENIDOS**

**INDICE DE TABLAS**

## INDICE DE FIGURAS

# INTRODUCCIÓN

En el ámbito de la ingeniería de software, los lenguajes de programación son herramientas esenciales para transformar ideas en soluciones computacionales. Sin embargo, muchos lenguajes predominantes mantienen estructuras rígidas heredadas de paradigmas antiguos que limitan la expresividad, incrementan la carga cognitiva y dificultan la depuración, afectando directamente la productividad de los programadores. Frente a esta problemática, surge la necesidad de sistemas que reduzcan la verbosidad sintáctica sin sacrificar el rendimiento técnico.

Este proyecto propone el desarrollo de DinoCode, un lenguaje de programación que prioriza una sintaxis minimalista, legible y flexible. Su arquitectura se fundamenta en el paradigma de la Regla de Oro, el uso de comas implícitas, la reducción de paréntesis y un sistema de tipado dinámico optimizado. El objetivo principal es optimizar la productividad del desarrollador, facilitando tanto el aprendizaje como el desarrollo ágil de software mediante un motor de inferencia capaz de interpretar la intención del código de manera determinista.

Para la ejecución de este proyecto, se adoptó una metodología de ingeniería de software basada en componentes (ISBC), integrada con un marco de gestión visual (Kanban). El proceso se estructuró en fases secuenciales que abarcan desde el análisis comparativo de lenguajes existentes y el diseño de una gramática de adyacencia, hasta la implementación de un prototipo funcional desarrollado en Rust. Si bien se utiliza DinoIDE como instrumento de prueba, el alcance de este trabajo se centra exclusivamente en el diseño y ejecución del lenguaje DinoCode.

El proyecto se alinea con la línea de investigación de Ingeniería De Software, Redes y Telecomunicaciones para el diseño e implementación de sistemas de información de la Universidad Estatal de Bolívar.

# RESUMEN

Este proyecto presenta el desarrollo de DinoCode, un lenguaje de programación de propósito general diseñado bajo un enfoque de eficiencia estructural y facilidad de uso. La innovación técnica reside en la implementación de una gramática de adyacencia y el paradigma de la Regla de Oro, mecanismos que permiten una resolución sintáctica basada en la disposición espacial de los tokens, permitiendo que la estructura formal del código emerja de la intención lógica del programador sin depender de delimitadores redundantes.

La metodología de desarrollo integró la ingeniería de software basada en componentes (ISBC) con el marco de trabajo Kanban para la gestión de tareas. La arquitectura del sistema se implementó en el lenguaje Rust, incorporando un transductor de estados finitos enriquecido (AEFT) y un sistema de gestión de memoria basado en NaN boxing, lo que garantiza una ejecución de alto rendimiento y seguridad de memoria.

La validación del sistema consistió en un análisis comparativo de rendimiento frente a la implementación de referencia de Python (ejecutada con optimizaciones de bootstrap) y un estudio de usabilidad con una muestra de 30 participantes de diversos niveles de experticia. Los resultados cuantitativos indican que DinoCode reduce los tiempos de ejecución en un promedio del 87.58%, operando con una huella de memoria significativamente menor (~2.8 MB). En términos de interacción humano-computador, el sistema obtuvo un nivel de aceptación del 93.34%, con una media de 3.73 en la efectividad del motor de inferencia de intención. Estos hallazgos posicionan a DinoCode como una arquitectura innovadora de alto rendimiento, con aplicaciones directas en el desarrollo ágil y la educación tecnológica.

**Palabras clave:** lenguajes de programación, DinoCode, gramática de adyacencia, regla de oro, inferencia contextual.

## ABSTRACT

This project introduces the development of DinoCode, a general-purpose programming language designed with a focus on structural efficiency and ease of use. The technical innovation lies in the implementation of an adjacency grammar and the "Golden Rule" paradigm—mechanisms that enable syntactic resolution based on the spatial arrangement of tokens. This allows the formal structure of the code to emerge from the programmer's logical intent without relying on redundant delimiters.

The development methodology integrated Component-Based Software Engineering (CBSE) with the Kanban framework for task management. The system architecture was implemented using the Rust language, incorporating an Augmented Enriched Finite Transducer (AEFT) and a syntax-directed compilation that emits bytecode linearly, ensuring high-performance execution and memory safety.

System validation consisted of a comparative performance analysis against a Python reference implementation (executed with bootstrap optimizations) and a usability study with a sample of 30 participants of varying expertise levels. Quantitative results indicate that DinoCode reduces execution times by an average of 87.58%, operating with a significantly smaller memory footprint (~2.8 MB). In terms of human-computer interaction, the system achieved a 93.34% acceptance level, with a mean of 3.73 in the effectiveness of the intent inference engine. These findings position DinoCode as an innovative high-performance architecture with direct applications in agile development and technological education.

**Keywords:** programming languages, DinoCode, adjacency grammar, golden rule, contextual inference.

# KAWSAY YUYAY

Kay rurayka DinoCode nishka killka rimaytami riksichin, kayka ruraykunata paktachinkapak, alli purinkapakpashmi rurashka kan. Kay ruraypa mushuk yuyayka, tokens nishkakuna maypi kashkata rikunami kan (adjacency grammar), shinami killkakka rurakpa yuyaymanta llukshin, mana kalla kashkakunata (delimiters) mutsurishpa.

Kayta rurankapakka Component-Based Software Engineering (CBSE) kashkata, Kanban nishkata yachaykunawanmi tantachishka. Ukuta rurankapakka Rust killka rimaytami kariyarka, chashnami AEFT nishkawan, bytecode nishkawanpash alli utka purichun, yuyaypash (memory) ama pantachun rurarka.

Alli kashkata rikunkapakka, Python (CPython) nishkawanmi chimbapurarka, shinallatak 30 yachakkunawanmi rikurka. DinoCode nishkaka 87.58% utkaypimi ashtawan paktachin, yuyaytapash (memory) ashallatami mutsurin (~2.8 MB). Kayta rikukpi, 93.34% runakunami "alli kan" nishpa rimarkakuna. Chaimantami DinoCode nishkaka mushuk, utka paktachik hatun ruray kan, yachaykunapakpash alli kanmi.

**Sinchilla shimikuna:** killka rimaykuna, DinoCode, adjacency grammar, regla de oro, yuyay hapirina.

# CAPÍTULO I

# FORMULACIÓN GENERAL DEL PROYECTO

## 1.1. Tema

Lenguaje de programación centrado en la eficiencia y facilidad de uso para programadores.

## 1.2. Descripción del Problema

El desarrollo de lenguajes de programación busca mejorar la productividad de los programadores mediante código claro, mantenible y eficiente (Chaudhari et al., 2025, p. 400). Sin embargo, la mayoría de los lenguajes de programación predominantes mantienen estructuras rígidas heredadas de paradigmas de los años 70 y 80 (Rafael & Recto, 2024). Estas restricciones sintácticas y semánticas limitan la expresividad, obligando a los desarrolladores a adaptarse a convenciones estrictas que no siempre reflejan la naturalidad de sus ideas.

A pesar de que existen propuestas modernas que intentan simplificar esta interacción, muchas sacrifican el rendimiento técnico o presentan ambigüedades difíciles de resolver en gramáticas complejas. Por lo tanto, surge la necesidad de un sistema que no solo reduzca la verbosidad mediante una sintaxis minimalista, sino que posea un motor de inferencia capaz de interpretar la intención del código de manera determinista.

## 1.3. Justificación

*DinoCode* surge para resolver la rigidez de las Gramáticas Libres de Contexto (GLC), las cuales obligan al programador a gestionar manualmente la cohesión del código mediante delimitadores (UNCPBA, 2009a; Vladimir, n.d., p. 72). El proyecto propone un modelo de inferencia contextual lineal que optimiza la experiencia de desarrollo bajo dos fundamentos principales:

### El paradigma de la *Regla de Oro*

La estructura del programa es una propiedad emergente de la intención del programador y no una imposición sintáctica rígida. Mediante este paradigma, el lenguaje asume la responsabilidad de deducir la continuidad o ruptura de las expresiones al integrar la disposición física del texto con el estado lógico del sistema. Este mecanismo elimina la dependencia de delimitadores explícitos como puntos y coma, comas redundantes o el uso de barras invertidas para escapar saltos de línea en sentencias extensas. Al respecto, según lo planteado por Myers et al. (2016), la reducción de la verbosidad sintáctica es una estrategia clave para disminuir la carga cognitiva y la propensión a errores de escritura en la interacción humano-computadora.

### Eficiencia algorítmica y resolución determinista

A diferencia de los modelos convencionales que sacrifican el rendimiento por la flexibilidad, DinoCode garantiza una resolución sintáctica lineal $O(n)$. Su motor de inferencia resuelve ambigüedades en una sola pasada, manteniendo la velocidad de un autómata determinista (Lovelle, 2001, p. 40). Este enfoque permite que el lenguaje interprete contextos complejos, como la distinción de la aridad de los operadores o la invocación implícita de funciones, sin incurrir en los tiempos de procesamiento exponenciales propios de las Gramáticas Sensibles al Contexto (GSC) tradicionales (UNCPBA, 2009b).

El núcleo de DinoCode ha sido desarrollado en Rust, garantizando seguridad de memoria y un rendimiento equilibrado. Este trabajo se alinea con la línea de investigación de Ingeniería De Software, Redes y Telecomunicaciones para el diseño e implementación de sistemas de información de la Universidad Estatal de Bolívar.

## 1.4. Objetivos

### General

Crear un lenguaje de programación que optimice la eficiencia en el desarrollo de software y facilite su uso mediante una sintaxis clara, estructuras simplificadas y mecanismos que mejoren la productividad del programador

### Específicos

- Analizar las características de expresividad de lenguajes de programación existentes para identificar limitaciones y oportunidades de mejora.
- Diseñar la sintaxis, semántica y estructuras principales de un nuevo lenguaje de programación orientado a la eficiencia y facilidad de uso.
- Implementar un prototipo funcional del lenguaje propuesto y evaluar su expresividad y usabilidad.

# CAPÍTULO II
# MARCO TEÓRICO

## 2.1. Antecedentes

A continuación, se presenta una síntesis histórica que integra la perspectiva cronológica de Chaudhari et al. (2025) con el análisis funcional de Rafael y Recto (2024), permitiendo trazar la evolución técnica de los lenguajes de programación.

### Primera generación (1940 – 1950)

El campo de los lenguajes de programación inició con enfoques de bajo nivel, representados por el lenguaje máquina. Este requería un manejo detallado del hardware, resultando en procesos complejos y propensos a errores. Al carecer de capas de abstracción, los programadores debían interactuar directamente mediante códigos binarios, lo que limitaba su accesibilidad a especialistas (Chaudhari et al., 2025; Rafael & Recto, 2024).

### Segunda generación (1950 – 1960)

La evolución hacia los lenguajes ensambladores introdujo el uso de *mnemónicos* simbólicos para representar instrucciones de la máquina. Aunque esta innovación mejoró la legibilidad en comparación con el binario, los lenguajes permanecían atados a arquitecturas de hardware específicas. Esta generación facilitó parcialmente la programación, pero seguía limitando su uso a los sectores especializados (Chaudhari et al., 2025; Rafael & Recto, 2024).

### Tercera generación (1960 – 1980)

Esta etapa marcó la transición hacia lenguajes de alto nivel independientes del hardware. Un hito fundamental fue FORTRAN (1957), orientado a cómputos científicos con una sintaxis similar a notaciones matemáticas. Posteriormente, BASIC (1964) priorizó una sintaxis cercana al lenguaje natural con fines educativos. En este periodo, lenguajes como C y C++ introdujeron paradigmas estructurados y orientados a objetos que mejoraron la modularidad y la reutilización de código, aunque manteniendo una gramática rígida obligatoria (Chaudhari et al., 2025; Rafael & Recto, 2024).

### Cuarta generación (1980 – 2010)

Durante este periodo, lenguajes como Java y Python avanzaron hacia una mayor usabilidad. Rafael y Recto (2024) y Chaudhari et al. (2025) coinciden en que esta generación se caracteriza por una alta abstracción que permite que instrucciones simples reemplacen múltiples líneas de código. Python, en particular, innovó al utilizar la indentación como delimitador de bloques, eliminando símbolos redundantes como las llaves. La tendencia general se consolidó en la automatización de la gestión de memoria y el soporte multiparadigma.

### Quinta generación (2010 – presente y futuro)

Rafael y Recto (2024) describen una quinta generación enfocada en lenguajes impulsados por Inteligencia Artificial (IA) y Procesamiento de Lenguaje Natural (NLP), donde los sistemas interpretan intenciones humanas con menor intervención manual. Esta generación prioriza la usabilidad extrema y la reducción de la carga cognitiva del programador.

### La usabilidad de los lenguajes modernos

A pesar de la evolución histórica, estudios como los de Myers et al. (2016) señalan una desconexión entre el diseño de lenguajes y los principios de Interacción Humano-Computadora (HCI). Myers sostiene que lenguajes como Java o C++ no han sido debidamente evaluados desde el factor humano, resultando en sintaxis que a menudo son difíciles de aprender, ineficientes o propensas a errores. Según su investigación, un lenguaje usable debe optimizar la curva de aprendizaje, alineándose con los modelos mentales naturales del desarrollador.

### El prototipo DinoIDE

El desarrollo del presente trabajo no surge de forma aislada, sino como una evolución crítica de experimentos precedentes. En un desarrollo previo del autor iniciado en 2024 (Quiroz, 2026), se exploró la arquitectura de un editor de código denominado DinoIDE (publicado bajo el seudónimo BlassGO). Este prototipo fue diseñado originalmente como un proyecto personal para la automatización de flujos de trabajo en entornos Android. Aunque este sistema inicial se basaba en una arquitectura rudimentaria de expresiones regulares y reemplazos de cadenas en tiempo de ejecución, su implementación fue fundamental como prueba de concepto. Este experimento permitió validar la viabilidad de simplificar la sintaxis para el usuario final y motivó la creación de un lenguaje de programación formal en la etapa actual.

Pese a sus orígenes, el prototipo fue actualizado y refactorizado en 2026 para adaptarse a las necesidades de la investigación actual. La identificación de limitaciones técnicas en las versiones iniciales justificó la decisión de desarrollar el núcleo de DinoCode desde cero utilizando el lenguaje Rust. De este modo, la versión más reciente de DinoIDE se integra en este trabajo exclusivamente como una herramienta de soporte para las pruebas de usabilidad.

**Tabla 1**

*Resumen de la evolución generacional y sintáctica de los lenguajes de programación*

| Generación | Periodo | Lenguaje | Innovación sintáctica | Usabilidad y accesibilidad |
|---|---|---|---|---|
| 1ra | 1940 | Lenguaje máquina | Uso de código binario puro. | Muy baja. Limitada a especialistas en hardware. |
| 2da | 1950 | Ensamblador | Introducción de mnemónicos. | Baja. Apertura a técnicos y operadores. |
| 3ra | 1957 | FORTRAN | Sintaxis basada en fórmulas matemáticas. | Media-Baja. Accesibilidad a nichos científicos. |
| 3ra | 1964 | BASIC | Sintaxis cercana al lenguaje natural. | Alta. Primera apertura a usuarios no expertos. |
| 3ra | 1970 | C | Estructura de bloques con delimitadores. | Media. Accesible para usuarios técnicos. |
| 4ta | 1991 | Python | Eliminación de símbolos redundantes. | Alta. Accesible para usuarios instruidos. |
| 5ta | 2010 / Presente | IA / NLP | Generación de código por intención. | Muy alta. Accesible para cualquier usuario. |

*Nota*. Elaboración propia basada en Chaudhari et al. (2025), Rafael y Recto (2024) y Myers et al. (2016).

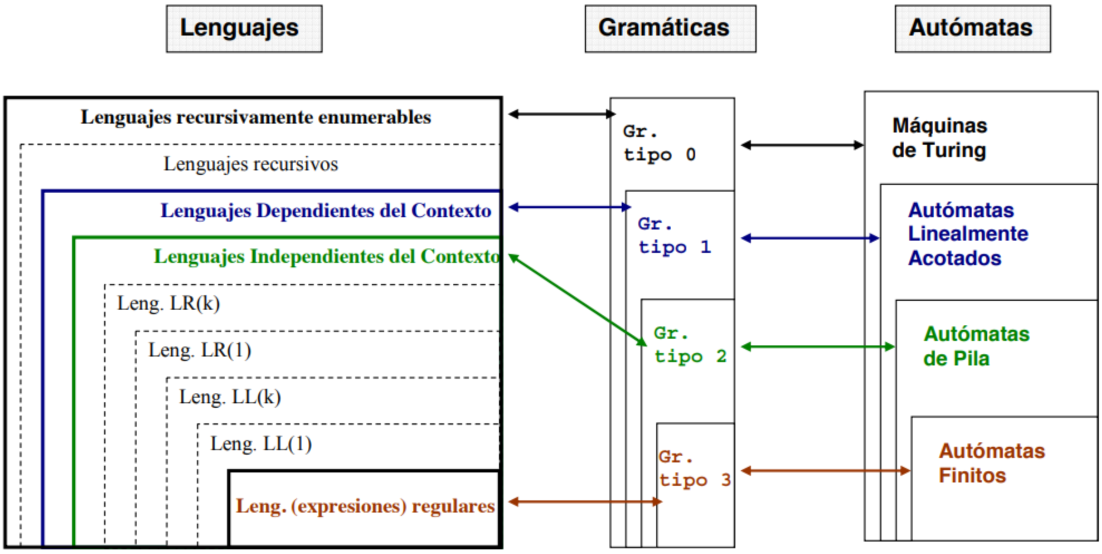
## 2.2. Científico

### 2.2.1 Jerarquía de Chomsky y teoría de autómatas

Acorde a lo expuesto por Málaga (2008) y Lovelle (2001), la arquitectura de cualquier lenguaje de programación se fundamenta en la clasificación de gramáticas propuesta por Noam Chomsky, la cual define el poder generativo de un sistema y la complejidad del autómata requerido para su procesamiento, como se presenta en la siguiente figura:

**Figura 1**

*Jerarquía de gramáticas, lenguajes y autómatas*



*Nota*. Tomado de *Teorías de Autómatas y Lenguajes Formales* (p. 8), por E. Jurado Málaga, 2008, Universidad de Extremadura.

En la ingeniería de compiladores, se emplean los siguientes niveles de abstracción:

**Tabla 2**

*Clasificación de las gramáticas de la jerarquía de Chomsky y su aplicación en las fases de compilación*

| Tipo de Gramática | Lenguaje asociado | Autómata reconocedor | Aplicación en compiladores |
|---|---|---|---|
| Tipo 3 | Lenguajes regulares | Autómata finito (AF) | Análisis léxico y expresiones regulares. |
| Tipo 2 | Lenguajes libres de contexto | Autómata de pila (PDA) | Análisis sintáctico y gramáticas BNF. |
| Tipo 1 | Sensibles al contexto | Autómata linealmente acotado | Análisis semántico y chequeo de tipos. |
| Tipo 0 | Recursivamente enumerables | Máquina de turing | Computación general e infraestructuras completas. |

*Nota*. Elaboración propia basada en Jurado Málaga (2008) y Lovelle (2001).

## A. Formalización de la gramática estructurada

De acuerdo con la formalización clásica de la jerarquía de Chomsky, toda gramática formal $G$ se define como una cuádruple:

$$G = (V_N, \Sigma, P, S)$$

Donde:

$\boldsymbol{V_N}$ identifica al conjunto finito de símbolos no terminales o variables, los cuales funcionan como marcadores de posición que deben ser reemplazados durante el proceso de derivación.

$\boldsymbol{\Sigma}$ representa el conjunto finito de símbolos terminales o alfabeto de la gramática, con la restricción fundamental de que este conjunto y el de no terminales sean disjuntos, tal que $V_N \cap \Sigma = \emptyset$.

$\boldsymbol{P}$ constituye el conjunto de reglas de producción, definidas formalmente como una relación binaria sobre $(V_N \cup \Sigma)^*$, cuya función principal es determinar las transformaciones permitidas entre cadenas de símbolos.

$\boldsymbol{S}$ se establece como el símbolo inicial o axioma de la gramática, debiendo pertenecer necesariamente al conjunto de no terminales ($S \in V_N$) para dar inicio a cualquier secuencia de derivación válida.

### B. Restricciones para gramáticas tipo 1

En este nivel de la jerarquía, las reglas de producción $P$ deben cumplir con el formato de sustitución contextual:

$$\alpha A\beta \to \alpha\gamma\beta$$

Donde la validez de la operación depende de las siguientes condiciones:

- El símbolo transformable $A$ debe ser obligatoriamente un elemento del conjunto de no terminales ($A \in V_N$).
- Los contextos izquierdo y derecho, representados por $\alpha$ y $\beta$ respectivamente, pueden ser cadenas vacías o compuestas por cualquier combinación de símbolos terminales y no terminales pertenecientes a $(V_N \cup \Sigma)^*$.
- El cuerpo de la producción $\gamma$ debe ser una cadena no vacía de símbolos, cumpliendo la condición $\gamma \in (V_N \cup \Sigma)^+$.
- La regla de no contracción establece que el peso o longitud de la cadena resultante debe ser mayor o igual a la original, expresado matemáticamente como $|\alpha A\beta| \leq |\alpha\gamma\beta|$, garantizando así que el proceso de análisis no sea infinito.

### 2.2.2 Transductores de estados finitos (FST) y redes aumentadas

Para la traducción y transformación de flujos simbólicos, se emplea la teoría de los transductores de estados finitos. A diferencia de un autómata aceptor tradicional, un transductor es una máquina que vincula un alfabeto de entrada con uno de salida mediante una relación racional.

Formalmente, basándose en la estructura propuesta por Gribkoff (2013), un transductor de estados finitos se define como una séptupla:

$$T = (Q, \Sigma, \Gamma, I, F, \delta, \sigma)$$

Donde:

$Q$ es el conjunto finito de estados internos de la máquina.

$\Sigma$ representa el alfabeto finito de entrada.

$\Gamma$ representa el alfabeto finito de salida.

$I$ es el conjunto de estados iniciales, donde $I \subseteq Q$.

$F$ es el conjunto de estados finales o de aceptación, donde $F \subseteq Q$.

$\delta$ constituye la función de transición de estados ($Q \times \Sigma \rightarrow Q$).

$\sigma$ es la función de salida que determina la cadena emitida en cada transición ($Q \times \Sigma \rightarrow \Gamma^*$).

Para manejar gramáticas que exceden la regularidad sin perder eficiencia lineal, se utilizan las redes de transición aumentadas (ATN). Estos modelos actúan como transductores recursivos que incorporan registros de almacenamiento y condiciones lógicas arbitrarias en los arcos de transición, permitiendo capturar estructuras profundas y transformaciones semánticas complejas.

### 2.2.3 Algoritmo de Shunting-yard y notación polaca inversa (RPN)

El análisis y la ejecución de expresiones matemáticas en sistemas de alto rendimiento se apoya en el algoritmo de Shunting-yard, diseñado por Dijkstra (1962). Este método iterativo emplea una estructura de pila para transformar expresiones de notación infija en representaciones postfijas inequívocas en tiempo lineal $O(n)$, como se muestra a continuación:

**Figura 2**

*Representación de una expresión infija en notación posfija (RPN)*

<table>
<tr><td>Notación infija</td><td colspan="11">5 + 39 / (7 + 2 * 3 ) - 6</td></tr>
<tr><td>Notación posfija (RPN)</td><td>5</td><td>39</td><td>7</td><td>2</td><td>3</td><td>*</td><td>+</td><td>/</td><td>+</td><td>6</td><td>-</td></tr>
</table>

*Nota*. Adaptado del apunte EWD28 de Dijkstra (1962).

La notación polaca inversa (RPN) resultante ofrece los siguientes beneficios técnicos:

- El algoritmo respeta intrínsecamente la precedencia y asociatividad de los operadores, lo que permite prescindir de delimitadores físicos (ej. paréntesis) en tiempo de ejecución.

- Permite que una máquina basada en pilas procese una expresión RPN en una sola pasada lineal, logrando así minimizar el movimiento de datos entre registros y memoria principal.

### 2.2.4 Representación de datos bajo IEEE 754 y NaN Boxing

La eficiencia en la gestión de tipos dinámicos depende de la arquitectura binaria de los datos en memoria. Acorde a la Universidad Nacional de Quilmes (2014) y Melançon y otros (2025), el estándar IEEE 754 para aritmética de punto flotante de 64 bits desglosa el espacio de bits con la siguiente estructura técnica:

**Tabla 3**

*Estructura de bits del estándar IEEE 754 (doble precisión)*

| Campo | Bits | Función técnica |
|---|---|---|
| Signo | 1 | Define la polaridad del valor real (positivo o negativo). |
| Exponente | 11 | Determina la magnitud.<br>$11111111111_2$ para NaN / infinitos |
| Mantisa | 52 | Almacena las cifras significativas o *payload* en caso de NaN. |

*Nota*. Elaboración propia basada en Melançon y otros (2025).

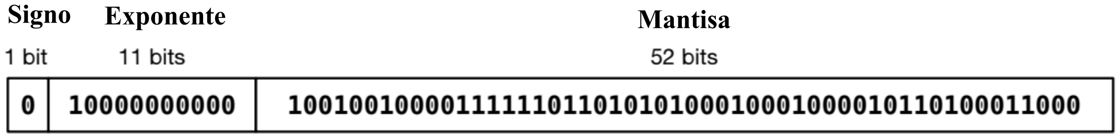
#### A. Técnica de NaN Boxing

Esta optimización aprovecha que un valor se clasifica como Quiet NaN si su exponente está compuesto totalmente por bits en 1 y el bit más significativo de la mantisa (bit 51) también es 1. Esta configuración deja 51 bits disponibles para almacenar información que no sea numérica sin que el procesador intente procesarlos como un número válido.

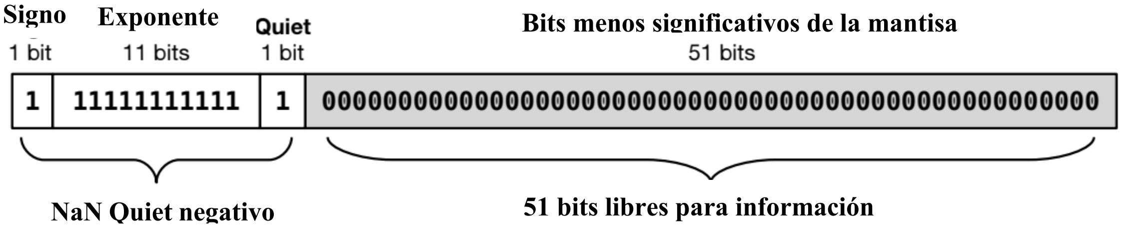
Visualmente, la distribución de un valor bajo NaN Boxing se organiza de la siguiente manera:

**Figura 3**

*Representaciones de números en punto flotante y NaN*



(a) Representación de $\pi$ según el estándar IEEE754.



(b) El NaN-boxing utiliza los 51 bits libres en valores NaN Quiet negativos para almacenar datos

*Nota*. Adaptado de *Float Self-Tagging* de Melançon y otros, 2025.

La implementación de este esquema permite las siguientes ventajas operativas:

- Aprovecha el espacio redundante. Las arquitecturas modernas de 64 bits solo direccionan hasta 48 bits, dejando los bits superiores libres para metadatos.
- El sistema utiliza los bits de la mantisa para codificar de forma compacta punteros a objetos, booleanos o enteros de 32 bits, diferenciándolos mediante una etiqueta o *tag* específica.
- Esta representación garantiza que cada valor ocupe exactamente 8 bytes, lo que optimiza el paso de parámetros por registros y mejora drásticamente el rendimiento de la memoria caché al mantener una densidad de datos constante.

### 2.2.5 Gestión de memoria

La latencia y el rendimiento de un sistema de ejecución están determinados por su modelo de gestión de memoria. Se distinguen dos enfoques fundamentales:

**Tabla 4**

*Comparativa de modelos de gestión de memoria en recolectores de tipo tracing*

| Criterio | Modelo de copying collection | Modelo de mark-and-sweep |
|---|---|---|
| Mecanismo de asignación | Emplea un esquema de *bump-pointer* que permite la asignación secuencial en un espacio contiguo de memoria. | Utiliza estructuras de *free-lists* para localizar y administrar bloques de memoria disponibles. |
| Complejidad | Garantiza un tiempo constante $O(1)$, al avanzar simplemente un *next pointer* (puntero de límite). | Presenta un tiempo variable, pues requiere recorrer la lista libre para hallar un bloque de tamaño $n$, adecuado. |
| Costo de recolección | El coste es proporcional únicamente a la memoria alcanzable (*live objects*), ignorando la basura. | El coste es proporcional a toda la memoria asignada, debido a la inspección en la fase de *sweep*. |
| Integridad del *heap* | Realiza una compactación intrínseca al mover los objetos desde el *from-space* hacia el *to-space*. | Es un método *in-place* que no mueve los objetos, lo que conlleva un riesgo elevado de fragmentación. |
| Localidad | Alta, debido a que los objetos vivos se reorganizan de forma contigua tras cada ciclo de copia. | Baja, dado que los objetos permanecen dispersos, lo que puede degradar el rendimiento del *caché*. |

*Nota*. Adaptado de *Lecture Notes on Garbage Collection*, por F. Pfenning, 2024, Carnegie Mellon University.

Como sostiene Pfenning (2024), los modelos basados en rastreo (*tracing*), simplifican la gestión al evitar las deficiencias intrínsecas del conteo de referencias frente a optimizaciones del compilador y la incapacidad de este último para resolver estructuras circulares. No obstante, el modelo de copia ofrece una ventaja competitiva adicional sobre otros métodos de rastreo (como *mark-and-sweep*) al optimizar la jerarquía de caché mediante la compactación de datos; esto permite mejoras en el rendimiento del mutador que la literatura técnica sitúa entre el 7% y el 15% frente a asignadores basados en listas libres (Berger et al., 2002).

### 2.2.6 Infraestructura de sistemas en Rust

Rust se establece como la tecnología fundamental para el desarrollo de infraestructuras de ejecución de alto rendimiento debido a su capacidad para garantizar la seguridad de memoria sin incurrir en la sobrecarga de un recolector de basura. Los pilares que sustentan su adopción en la ingeniería de lenguajes son:

**Tabla 5**

*Pilares de seguridad y rendimiento en la arquitectura de Rust*

| Pilar técnico | Mecanismo de control | Impacto en la ejecución |
|---|---|---|
| Modelo de propiedad | Sistema de *Ownership* y *Borrowing* validado estáticamente. | Eliminación de fugas de memoria y punteros colgados (*dangling pointers*). |
| Abstracciones de costo cero | Mapeo de construcciones de alto nivel a código máquina eficiente. | Rendimiento óptimo en la manipulación de registros, equiparable a C++. |
| Abstracciones seguras | Encapsulación de lógica compleja tras interfaces protegidas. | Aislamiento de operaciones críticas (como el acceso a memoria cruda). |

*Nota***.** Basado en *How do programmers use unsafe Rust?* (pp. 1-3), por V. Astrauskas et al., 2020.

Como investigan Astrauskas y otros (2020), el lenguaje opera bajo el principio de que el código potencialmente inseguro (*unsafe*) debe usarse de forma mesurada y estar siempre oculto tras una abstracción segura. Este diseño permite implementar estructuras de datos complejas que no pueden representarse bajo las reglas estrictas del compilador, asegurando que el cliente final interactúe con una interfaz libre de riesgos de memoria.

## 2.3. Conceptual

**AEFT (Augmented Finite State Transducer)**

Motor propio de DinoCode que extiende los transductores tradicionales para procesar gramáticas de adyacencia mediante un registro de estados previos.

**Array / Arreglo**

Estructura de datos que almacena múltiples elementos del mismo tipo en posiciones contiguas accesibles por índice entero (de Zwart et al., 2024).

**Arena**

Espacio de asignación rápida para datos constantes (Strings y BigInts), que optimiza el acceso y simplifica el rastreo al no permitir cambios (Berger et al., 2002).

**ASI (Automatic Semicolon Insertion)**

Mecanismo en JavaScript que intenta insertar automáticamente puntos y comas al final de sentencias, con reglas opacas que pueden causar comportamientos inesperados (Alam, 2025).

**AST (Abstract Syntax Tree)**

Representación arbórea que captura la estructura sintáctica jerárquica de un programa de forma abstracta, omitiendo detalles sintácticos superfluos (de Zwart et al., 2024).

**Asociatividad**

Regla que define cómo se agrupan operadores del mismo nivel de precedencia (izquierda-a-derecha o derecha-a-izquierda) (Ruehr, 2017).

**ATN (Augmented Transition Network)**

Extensión de transductores finitos que incorpora registros de almacenamiento y condiciones lógicas arbitrarias en los arcos de transición (Gribkoff, 2013).

**Automaton / Automata**

Máquina matemática abstracta que procesa símbolos conforme a reglas definidas, generando un comportamiento reconocible según transiciones de estado (Jurado Málaga, 2008).

**Benchmark**

Prueba que mide características de rendimiento (velocidad, memoria, latencia) comparándolas con referencias estándar o sistemas competidores (Alroobaea & Mayhew, 2014).

**BigInt**

Tipo de dato que representa enteros de precisión arbitraria sin límite de magnitud, permitiendo cálculos exactos con números extremadamente grandes (Granlund & GMP development team, 2026).

**Big O Notation**

Notación matemática que describe la complejidad asintótica de algoritmos en términos del tamaño de entrada (Pardo, 2016).

**Borrowing**

Mecanismo en Rust que permite a una función usar una referencia a un valor sin tomar su propiedad, garantizando su validez mediante análisis estático (Astrauskas et al., 2020).

**Bootstrap**

Proceso de inicialización de estructuras de tiempo de ejecución, permitiendo que sistemas se auto-configuren (Sibaja, 2017).

**Breakers**

Operadores postfijos que fuerzan una ruptura de continuidad en el motor de inferencia de DinoCode.

**Bump Allocator**

Estrategia simple de asignación de memoria donde un puntero se desplaza ("salta") secuencialmente al siguiente espacio libre (Pfenning, 2024).

**Bytecode**

Código intermedio de bajo nivel, independiente de la plataforma, que es ejecutado por una máquina virtual en lugar de un procesador nativo (Saleil & Feeley, 2018a).

**Closure / Clausura**

Función que captura variables del contexto donde fue definida, accediendo a ellas posteriormente incluso después de que ese contexto finalice (Lovelle, 2001).

**Carga Cognitiva**

Demanda mental que el usuario debe invertir para usar una interfaz o lenguaje de software efectivamente (Myers et al., 2016).

**Cohesión**

Medida de cuánto están relacionadas las responsabilidades dentro de un componente o módulo de software (Pressman, 2010).

**Compilador**

Programa que traduce el código fuente de un lenguaje de programación a código máquina o formato intermedio, optimizando su ejecución (Pressman, 2010).

**Component-based Software Engineering (ISBC)**

Enfoque que construye sistemas complejos mediante el ensamblaje de componentes autónomos y reutilizables (Pressman, 2010).

**Context-free Grammar (GLC)**

Lenguaje cuyas reglas de producción no dependen del contexto donde aparece un símbolo no-terminal (UNCPBA, 2009a).

**Context-sensitive Grammar (GSC)**

Lenguaje cuyas reglas de producción pueden depender del contexto izquierdo y derecho del símbolo a transformar (UNCPBA, 2009b).

**Copying Collection**

Algoritmo de recolección de basura que copia objetos vivos a una nueva región de memoria, compactando el heap automáticamente (Pfenning, 2024).

**Crate**

Unidad compilable mínima en Rust que puede ser una biblioteca o un ejecutable (Astrauskas et al., 2020).

**Dangling Pointer**

Referencia que apunta a una región de memoria que ha sido desasignada, causando un comportamiento indefinido si se desreferencia (Astrauskas et al., 2020).

**Delimitador**

Símbolo que marca el inicio o fin de una construcción sintáctica (comillas, paréntesis, llaves) (Lovelle, 2001).

**Derivación**

Proceso secuencial de aplicar reglas de producción partiendo del símbolo inicial hasta obtener una cadena terminal válida (Jurado Málaga, 2008).

**DinoCode**

Lenguaje de programación de propósito general, cuyo diseño prioriza la eficiencia estructural y la reducción de carga cognitiva mediante un paradigma de inferencia de intención determinista.

**DinoIDE**

Entorno de desarrollo integrado diseñado específicamente para escribir, probar y depurar código del lenguaje DinoCode (Quiroz, 2026).

**DinoRef**

Estructura de 64 bits que representa cualquier valor en DinoCode mediante la técnica de NaN Boxing, permitiendo un tipado dinámico compacto (Melançon et al., 2025).

**EBNF (Extended Backus-Naur Form)**

Es un metalenguaje utilizado para especificar la gramática de un lenguaje formal mediante reglas que definen cómo se combinan elementos mínimos para formar estructuras válidas (Tomassetti, 2017).

**Feedback**

Información que proporciona el sistema o los usuarios en respuesta a acciones específicas, esencial la usabilidad (Alroobaea & Mayhew, 2014).

**Finite Automaton (FA)**

Máquina teórica con número finito de estados que reconoce lenguajes regulares mediante transiciones determinísticas o no-determinísticas (Jurado Málaga, 2008).

**Finite State Machine (FSM)**

Modelo de cómputo que procesa entradas basado en estados discretos y transiciones bien definidas deterministas o probabilísticas (Gribkoff, 2013).

**Finite State Transducer (FST)**

Autómata finito que produce una salida por cada entrada procesada, transformando flujos de símbolos (Gribkoff, 2013).

**Forwarding Pointer**

Puntero especial de un objeto que registra su nueva ubicación después de ser movido durante la recolección de basura (Pfenning, 2024).

**Free List / Listas Libres**

Estructura que mantiene una lista de bloques de memoria libres disponibles para su asignación dinámica (Pfenning, 2024).

**From-space**

Región de la memoria que contiene los objetos activos y datos procesables antes de iniciar un ciclo de recolección (Pfenning, 2024).

**Gramática de adyacencia**

Modelo formal de análisis sintáctico donde la relación entre los elementos del lenguaje no depende exclusivamente de delimitadores físicos, sino de la proximidad física y la disposición espacial de los tokens en el flujo de entrada.

**Garbage Collection (GC)**

Mecanismo automático que identifica objetos que ya no son accesibles y libera la memoria que ocupan (Pfenning, 2024).

**HCI (Human-Computer Interaction)**

Disciplina de estudio que examina la relación entre usuarios y sistemas computacionales, enfocada en la usabilidad (Myers et al., 2016).

**Heap**

Región de memoria dinámica donde se asignan estructuras de datos complejas durante el tiempo de ejecución (Pfenning, 2024).

**Inferencia contextual**

Proceso determinista ejecutado por el transductor sintáctico para resolver ambigüedades mediante el análisis del estado de continuidad operativa y el contexto circundante, permitiendo la omisión de paréntesis y otros símbolos sin pérdida de precisión semántica.

**IEEE 754**

Estándar internacional para aritmética de punto flotante que define formatos de 32 y 64 bits con reglas de redondeo (Universidad Nacional de Quilmes, 2014).

**In-place**

Método de limpieza que procesa objetos en su ubicación original sin moverlos, lo que puede comprometer la integridad del heap por fragmentación (Pfenning, 2024).

**Intérprete**

Motor que traduce y ejecuta código línea por línea sin compilación previa a código máquina (Pressman, 2010).

**Interning**

Técnica que asegura que valores iguales (especialmente strings) compartan la misma representación en memoria (Lakshmanan, 2022).

**Kanban**

Metodología ágil que visualiza el flujo de trabajo en un tablero y limita el trabajo en curso (WIP) para optimizar la eficiencia al reorganizar prioridades (Anzules & Moya, 2024).

**Lenguaje de programación**

Sistemas de signos y reglas semánticas que permiten la construcción de algoritmos para la resolución de problemas computacionales (Vladimir, n.d.).

**Lexer**

Componente del compilador que realiza el análisis léxico, descomponiendo el código fuente en símbolos gramaticales (tokens) (Ángel, n.d.).

**Likert Scale**

Escala de medición de 5 puntos (típicamente) que mide el grado de acuerdo en encuestas: muy en desacuerdo a muy de acuerdo (Matas, 2018).

**Live Objects**

Bloques de datos en memoria que aún cuentan con al menos una referencia activa y accesible por el programa (Pfenning, 2024).

**Mnemónicos**

Códigos alfanuméricos simples (como MOV o ADD) que representan instrucciones binarias de bajo nivel (McCandless & Gregg, 2011).

**Mark-and-Sweep**

Algoritmo de recolección de basura que marca objetos alcanzables y luego barre la memoria eliminando los no marcados (Pfenning, 2024).

**NaN (Not a Number)**

Valor especial en IEEE 754 que representa un número indefinido o resultado de operación inválida (Melançon et al., 2025).

**NaN Boxing**

Optimización que aprovecha valores NaN del IEEE 754 para almacenar metadatos en bits disponibles de la mantisa (Melançon et al., 2025).

**Next pointer**

Puntero que almacena la dirección de memoria del siguiente nodo en estructuras lineales o listas de bloques libres (Pfenning, 2024).

**Offset**

Valor numérico que indica la posición exacta de una instrucción o dato relativo al inicio de un flujo de bytes o estructura (Saleil & Feeley, 2018b).

**Opcode**

Byte que identifica la operación específica que debe realizar el procesador o máquina virtual (McCandless & Gregg, 2011).

**Parsing**

Proceso de analizar sintácticamente un flujo de símbolos conforme a las reglas de una gramática formal (Lovelle, 2001).

**Payload**

Carga útil de datos dentro de una instrucción o referencia, independiente de los metadatos de control (Melançon et al., 2025)

**Pipeline**

Flujo secuencial de transformación del código fuente hasta su conversión en código ejecutable (Saleil & Feeley, 2018b).

**Pool de objetos**

Reserva de memoria pre-asignada para estructuras complejas y reutilizables (arrays y objetos), minimizando el costo de creación en el heap (Berger et al., 2002).

**Regla de oro**

Paradigma de diseño de lenguajes que postula que la estructura formal del código debe emerger directamente de la intención lógica y la organización visual del programador, eliminando la redundancia sintáctica en favor de una interpretación intuitiva por parte del compilador.

**Regex / Regular Expression**

Patrón textual que describe un conjunto de strings según reglas formales definidas, usado para búsqueda y validación (Sedgewick & Wayne, n.d.).

**RPN (Reverse Polish Notation)**

Notación postfija donde los operadores se escriben después de los operandos, permitiendo una evaluación con pila sin delimitadores (Dijkstra, 1962; Ruehr, 2017).

**Shunting-yard Algorithm**

Algoritmo diseñado por Dijkstra (1962) que transforma expresiones infijas en notación postfija (RPN) (Dijkstra, 1962; Ruehr, 2017).

**Tag**

Etiqueta de bits que identifica el tipo de dato (String, Array, Object, etc.) en NaN Boxing (Melançon et al., 2025).

**Token**

Es la unidad léxica mínima con significado propio (como una palabra clave, un identificador o un símbolo) en la que el analizador léxico divide el código fuente (Ángel, n.d.).

**To-space**

Espacio de destino donde el recolector copia exclusivamente los objetos vivos, compactándolos para eliminar la fragmentación (Pfenning, 2024).

**Tracing**

Proceso de rastreo de referencias desde las raíces (pila/globales) para identificar y marcar objetos accesibles en el heap (Pfenning, 2024).

**Virtual Machine / VM**

Abstracción de software que ejecuta instrucciones (bytecode) en un entorno independiente del hardware subyacente (Saleil & Feeley, 2018a).

**WIP (Work In Progress)**

Cantidad de trabajo que está siendo realizado simultáneamente en una gestión ágil como Kanban (Anzules & Moya, 2024).

**Workspace**

Colección de múltiples crates de Rust que comparten una configuración común y dependencias (Astrauskas et al., 2020).

## 2.4. Legal

En Ecuador, el marco legal aplicable al desarrollo de lenguajes de programación se rige por la Ley Orgánica de Economía Social del Conocimiento, Creatividad e Innovación (2016), que reconoce como propiedad intelectual a los programas de ordenador, incluidos el código fuente y objeto. Esta protección deriva de la Decisión 351 de la Comunidad Andina, que equipara los programas de ordenador a las obras literarias protegidas por derechos de autor. En consecuencia, el lenguaje desarrollado será automáticamente protegido bajo derechos de autor, tanto en Ecuador como en los países miembros de la Comunidad Andina (Cabascango-Bedoya, 2024).

De acuerdo con la Convención de Berna, ratificada por Ecuador, estos derechos incluyen atribución de autoría, integridad de la obra y los derechos patrimoniales, permitiendo determinar el régimen de licenciamiento del software, ya sea propietario o libre.

## 2.5. Georeferencial

El proyecto se desarrolla en el contexto educativo de la Universidad Estatal de Bolívar, Ecuador. El nivel académico de los estudiantes de programación permite realizar pruebas reales de uso, considerando que la carrera de Software se estructura en ciclos que cubren desde fundamentos de programación hasta asignaturas avanzadas.

# CAPÍTULO III
# METODOLOGÍA

## 3.1. Metodología de desarrollo de software

Para el desarrollo de este proyecto, se seleccionó la ingeniería de software basada en componentes (ISBC). Según Pressman (2010), este enfoque se clasifica como un modelo de proceso especializado que se fundamenta en la reutilización de entidades denominadas "componentes de software", permitiendo la construcción de sistemas complejos mediante el ensamblaje de piezas autónomas y prefabricadas.

### La ISBC en Rust

La ISBC se acopla con Rust debido a la estructura de *crates* y módulos del lenguaje. Rust permite el encapsulamiento de lógica mediante modificadores de visibilidad, asegurando que cada unidad sea funcional e independiente (Astrauskas et al., 2020).

**Tabla 6**

*Análisis comparativo de modelos de procesos de desarrollo de software*

| Modelo | Filosofía de diseño | Ventajas técnicas | Idoneidad |
|---|---|---|---|
| Cascada | Secuencial y rígido por etapas bien definidas. | Documentación exhaustiva. | Baja. No tolera ajustes en la gramática. |
| Incremental | Entrega de productos funcionales (incremento). | Validación progresiva. | Media. Menor énfasis en la autonomía de las piezas. |
| ISBC | Ensamblaje de entidades reutilizables. | Bajo acoplamiento. | Alta. Maneja la complejidad del sistema en componentes y se integra con el stack tecnológico. |

*Nota.* Elaboración propia basada en Pressman (2010).

## 3.2. Marco de trabajo para la gestión de tareas

La gestión del flujo de trabajo se rige por Kanban. Este marco de trabajo se centra en la visualización del flujo y la limitación del trabajo en curso (*Work In Progress* - WIP) para optimizar la eficiencia y reducir el desperdicio.

Según Anzules y Moya (2024), Kanban es especialmente efectivo en el desarrollo de software por su capacidad de adaptar prioridades rápidamente sin la sobrecarga administrativa que conllevan otros marcos como Scrum. Para el desarrollo de DinoCode, Kanban permite gestionar las tareas técnicas de forma fluida, evitando la rigidez de los Sprints de tiempo fijo y permitiendo enfocarse en la estabilidad de un componente antes de transicionar al siguiente.

**Tabla 7**

*Contraste entre marcos de gestión ágil para proyectos de software*

| Marco de trabajo | Enfoque | Flexibilidad | Aplicabilidad individual |
|---|---|---|---|
| Kanban | Flujo continuo por demanda. | Máxima. Re-priorización diaria. | Alta. Sin ceremonias innecesarias. |
| Scrum | Iteraciones de tiempo fijo (Sprints). | Media. Cambios limitados por Sprint. | Baja. Sobrecarga de roles y reuniones. |
| XP | Calidad técnica y ciclos cortos. | Alta. Basada en historias. | Baja. Requiere programación en parejas. |

*Nota.* Elaboración propia basada en Anzules y Moya (2024). Se evalúan marcos de trabajo acorde a su factibilidad para un desarrollo individual.

## 3.3. Fases de desarrollo

El desarrollo de DinoCode se estructura en cinco fases descritas a continuación:

### Fase 1: Análisis de requisitos y dominio

1. Analizar las limitaciones sintácticas en lenguajes predominantes para identificar oportunidades de mejora.
2. Definir los requisitos funcionales y no funcionales del lenguaje propuesto.
3. Definir la infraestructura común y las estructuras de datos transversales necesarias para el sistema.

### Fase 2: Especificación de interfaces

1. Diseñar la gramática formal del lenguaje y sus reglas de producción
2. Establecer las interfaces de entrada y salida para el lexer, el parser y la máquina virtual.

**Fase 3: Desarrollo y certificación de componentes**

1. Implementar la lógica interna de cada módulo de forma aislada
2. Realizar pruebas unitarias en Rust para asegurar la solidez técnica de cada componente.

**Fase 4: Ensamblaje**

1. Unificar los componentes certificados bajo una capa de orquestación común
2. Implementar el flujo de transformación desde el código fuente hasta la ejecución del bytecode.

**Fase 5: Validación del sistema y evaluación de resultados**

1. Someter el sistema a pruebas de estrés y rendimiento.
2. Publicar el prototipo funcional en un repositorio de GitHub para su acceso público.
3. Proveer un manual de operación (README) con las instrucciones de instalación
4. Distribuir ejemplos de código con algoritmos comunes y características propias del lenguaje.
5. Recolectar la percepción de los desarrolladores mediante un cuestionario digital.
6. Contrastar las métricas obtenidas con los objetivos de eficiencia y facilidad de uso.

## 3.4. Técnicas e instrumentos de recopilación de datos

### 3.4.1 Técnicas de recopilación de datos

**Revisión bibliográfica**

Se emplea una revisión de documentos técnicos, manuales de referencia y literatura especializada. Se utiliza tanto para la fundamentación teórica como para el estudio de las especificaciones de los lenguajes de programación.

**Observación experimental**

Se realiza una ejecución controlada de pruebas sobre diferentes artefactos tecnológicos y entornos de desarrollo. Se utiliza para registrar de forma directa el comportamiento de los compiladores, la interpretación de la sintaxis y el desempeño de las máquinas virtuales.

### Encuesta

Se aplica un cuestionario estructurado a una muestra de desarrolladores. Se utiliza para recolectar datos primarios sobre la percepción cualitativa de la claridad sintáctica, la usabilidad y la expresividad del lenguaje propuesto.

## 3.4.2 Instrumentos de recopilación de datos

### Matriz de análisis

Se emplea un formato de registro sistemático para organizar las variables observadas. Se utiliza como soporte para documentar de forma comparativa las inconsistencias sintácticas y asentar los resultados cuantitativos de rendimiento obtenidos durante la experimentación.

### Cuestionario digital

Se emplea un instrumento de recolección de datos distribuido mediante una plataforma en línea. Se utiliza para capturar de manera estandarizada la percepción sobre la eficiencia sintáctica a través de preguntas de escala (tipo likert) y recolectar observaciones cualitativas mediante preguntas de respuesta abierta.

### Monitor de rendimiento (Benchmarking)

Se emplea un motor de medición especializado de bajo nivel, desarrollado en el lenguaje Rust para garantizar precisión y mínima latencia. Se utiliza para capturar de forma síncrona el tiempo de ejecución y el pico máximo de consumo de memoria mediante un muestreo de procesos cada 10 ms, permitiendo obtener métricas cuantitativas objetivas y libres de la sobrecarga de la interfaz gráfica.

### Entorno de pruebas (DinoIDE)

Se utiliza una plataforma de desarrollo integrada como soporte técnico para la experimentación. Este entorno facilita la ejecución del código DinoCode desde una interfaz gráfica con resaltado de sintaxis y predicción de texto, funcionando como la herramienta mediadora para la captura de métricas y el feedback de los usuarios.

## 3.5. Población y muestra

La investigación requiere participantes con conocimientos técnicos para validar la funcionalidad y expresividad del lenguaje.

### 3.5.1 Muestreo y reclutamiento

Se utilizó un muestreo no probabilístico por conveniencia, conformado por perfiles técnicos voluntarios. Según Hernández Sampieri y otros (2014), este procedimiento es técnica y científicamente válido cuando el investigador requiere casos que posean conocimientos profundos o específicos sobre el objeto de estudio.

El proceso de reclutamiento se ejecutó de forma híbrida:

1. **Canal institucional**

   Con autorización de la Coordinación de la Carrera de Software de la Universidad Estatal de Bolívar (UEB), se difundió la convocatoria y el acceso al repositorio en GitHub mediante los grupos oficiales de WhatsApp de la facultad.

2. **Canal profesional**

   Convocatoria abierta en redes orientadas a la tecnología como LinkedIn y GitHub para captar desarrolladores externos con experiencia.

### 3.5.2 Descripción de la muestra

La muestra consta de 30 participantes, conformada por estudiantes y desarrolladores. Esta selección se realizó bajo un criterio de competencia técnica, asegurando que los sujetos posean los conocimientos necesarios para interactuar con la lógica del lenguaje propuesto.

Según AlRoobaea y Mayhew (2014), en el desarrollo de software complejo se define un umbral de seguridad de 16 ± 4 usuarios recomendado para sistemas dinámicos. Esto valida que una muestra de 30 usuarios permite alcanzar un descubrimiento promedio del 99% de los problemas de usabilidad.

**Tabla 8**

*Ficha técnica de la muestra del estudio de usabilidad*

| Variable | Descripción |
|---|---|
| Objeto de evaluación | Lenguaje de programación DinoCode |
| Nro total de participantes | 30 sujetos voluntarios |
| Sistemas Operativos | Windows (vía DinoIDE) y Linux (vía Terminal/CLI) |
| Instrumento de evaluación | Cuestionario digital de usabilidad |
| Criterio de selección | Conocimientos en programación y lógica |
| Canales de difusión | WhatsApp institucional, LinkedIn y GitHub |

*Nota.* Elaboración propia. Justificación de muestreo basada en Hernández-Sampieri y otros (2014) y tamaño de muestra según Alroobaea & Mayhew (2014).

# CAPÍTULO IV
# INGENIERÍA DEL PROYECTO

## 4.1. Análisis de lenguajes predominantes

Los lenguajes tradicionales han sido diseñados históricamente para facilitar la tarea del compilador o del intérprete; esto delega en el programador la responsabilidad de adaptarse a reglas que, en muchos casos, desafían la percepción visual natural.

### 4.1.1 Inconsistencia en la interpretación de literales

### A. El conflicto en rutas de archivos y expresiones regulares

En lenguajes de amplio uso, la barra invertida (\) cumple una función dual al actuar como separador de directorios en entornos Windows y, simultáneamente, como carácter de escape. Esta ambigüedad funcional no se restringe a la gestión de rutas de archivos, sino que se manifiesta en cualquier contexto que involucre metacaracteres, por ejemplo, las expresiones regulares (*Regex*).

Para mitigar las colisiones resultantes, los lenguajes han proliferado diversos modos de literales. La Tabla 9 detalla esta fragmentación, evidenciando que la solución común ha sido crear un catálogo de prefijos y delimitadores que el programador debe manejar constantemente.

**Tabla 9**

*Análisis de la fragmentación de literales y parches de mitigación*

| Lenguaje | Tipo de literal | Mejor escenario | Peor escenario | Riesgo técnico |
|---|---|---|---|---|
| Python | Estándar | `“ruta\\1”` | `“ruta\1”` | Corrupción silenciosa. Permite escapar cualquier carácter. |
| | Semi-puro | `r"ruta\1"` | `r"ruta\1\"` | Literal colgante. El delimitador aún puede escaparse. |
| | Interpolado | `f"{ruta}\\1"` | `f"{ruta}\1"` | Igual que el estándar. |

| | | | | |
|---|---|---|---|---|
| | Semi-puro + interpolado | `fr"{ruta}\1"` | `fr"{ruta}\1\"` | Igual que el semi-puro. |
| Ruby | Interpolado | `"#{ruta}\1"` | `"#{ruta}\t1”` | Corrupción solo por escapes válidos, de lo contario, preserva la barra y el carácter. |
| | Semi-puro (Simple) | `'ruta\t1'` | `'ruta\t1\'` | Literal colgante. El delimitador aún puede escaparse. |
| | Semi-puro (Sigil q) | `%q(ruta\t1)` | `%q(ruta\t1\)` | Igual que el simple. |
| | Semi-puro (Sigil Q) | `%Q(#{ruta}\t1)` | `%Q(#{ruta}\t1\)` | Igual que el simple. |
| C# | Estándar | `"ruta\\1"` | `"ruta\t1"` | Corrupción solo por escapes válidos, de lo contario, presenta un error de compilación. |
| | Interpolado | `$"{ruta}\\1"` | `$"{ruta}\t1"` | Igual que el estándar. |
| | Puro (Verbatim) | `@"ruta\t1\"` | `---` | Robusto. La barra invertida no tiene poder de escape. |
| | Verbatim + interpolado. | `$@"{ruta}\t1\"` | `---` | Igual que el puro. |
| C++ | Estándar | `"ruta\\1"` | `"ruta\t1"` | Corrupción solo por escapes válidos, de lo contario, es indefinido (no predecible). |
| | Puro | `R"(ruta\1\)"` | `---` | Robusto. La barra invertida no tiene poder de escape. |

| | | | | |
|---|---|---|---|---|
| Rust | Estándar | `"ruta\\1"` | `"ruta\t1"` | Corrupción solo por escapes válidos, de lo contario, presenta un error de compilación. |
| | Puro | `r#"ruta\1\"#` | --- | Robusto. La barra invertida no tiene poder de escape. |

*Nota*. Los campos marcados como "---" indican escenarios donde la sintaxis nativa impide la colisión entre el carácter de escape y el delimitador. El término "semi-puro" se refiere a literales donde el escape sigue activo para el delimitador de cierre.

Como se observa en la Tabla 9, la mayoría de los lenguajes industriales (con excepción de C#, C++ y Rust en sus modos específicos) presentan el fenómeno del literal colgante. Este error de diseño ocurre cuando el carácter de escape conserva su funcionalidad sobre el delimitador de cierre incluso en modos de texto puro, obligando al desarrollador a inspeccionar visualmente el final de cada cadena para evitar errores de sintaxis o corrupciones de datos.

Tomando a Python como modelo de análisis detallado en la Tabla 10, se evidencia cómo el lenguaje prioriza por defecto la secuencia de control sobre el contenido literal, forzando la adopción de prefijos especializados para preservar la integridad de los datos.

**Tabla 10**

*Análisis de corrupción de datos y mecanismos de corrección*

| Código en Python | Interpretación | Resultado |
|---|---|---|
| `ruta = 'C:\temp\new.txt'` | `C:     emp`<br>`ew.txt` | Corrupción de la ruta |
| `ruta = 'C:\\temp\\new.txt'` | `C:\temp\new.txt` | Ruta correcta |
| `ruta = r'C:\temp\new.txt'` | `C:\temp\new.txt` | Ruta correcta |
| `pattern = '\d+'` | `d+` | Corrupción de la expresión regular |
| `pattern = '\\d+'` | `\d+` | Regex correcto |
| `pattern = r'\d+'` | `\d+` | Regex correcto |

*Nota.* El intérprete procesa `\t` como un tabulador, `\n` como un salto de línea y `\d` como un carácter ‘d’.

## B. La fricción entre estética del código e integridad del dato

El desafío de las cadenas multilínea radica en la dificultad de alinear el texto con la estructura jerárquica del programa sin alterar la integridad del literal. Como se observa en la Tabla 11, los lenguajes actuales suelen imponer un compromiso técnico: el desarrollador debe optar por mantener la legibilidad del código o preservar la fidelidad del dato

**Tabla 11**

*Análisis de inconsistencia visual en bloques multilínea*

| Código | Interpretación | Problema detectado |
|---|---|---|
| # Python | “\nHola\n    Mundo” | • Contaminación. Captura los espacios del margen y el primer/último salto de línea como datos reales.<br>• Dependencia. Si se desea eliminar los espacios extra se debe utilizar librerías en tiempo de ejecución. |
| // Rust | “\nHola\n    Mundo” | • Contaminación y dependencia, igual que Python. |
| // Kotlin | “Hola” | • Contaminación y dependencia, igual que Python. Sin embargo, incluye un método nativo para limpiar los espacios en tiempo de ejecución. |

```
# Python

def f():
    msg = """
    Hola
    Mundo"""
```

```
// Rust

 let s = r#"
    Hola
    Mundo"#;
```

```
// Kotlin

msg = """
Hola""".trimIndent()
```

<table>
<tr>
<td><pre>
# Ruby

def con_sangria()
   # Usando ‘-’
   msg = <<-TEXT
   Hola
   TEXT
end

def sin_sangria()
  # Usando ‘~’
  msg = <<~TEXT
  Hola
  Mundo
  TEXT
end
</pre></td>
<td>“   Hola\n”<br><code>"Hola\nMundo\n"</code></td>
<td><ul><li>Rigidez. El delimitador de cierre (TEXT) siempre debe estar al inicio de línea.</li><li>Carga cognitiva. El programador debe escoger entre distintos símbolos y delimitadores. Además, tiene que pre-calcular la línea con menor margen (es la que dicta el margen del bloque, sin importar su orden)</li></ul></td>
</tr>
<tr>
<td><pre>
// C / C++

"Hola\n"
"Mundo"
</pre></td>
<td>“Hola\nMundo”</td>
<td><ul><li>Fragmentación. Requiere gestionar manualmente cada salto de línea y comillas.</li></ul></td>
</tr>
<tr>
<td><pre>
// C# / Java

msg = """
    Hola
    """;
</pre></td>
<td>“Hola”</td>
<td><ul><li>Dependencia posicional y por delimitador. Se le resta el margen de las comillas de cierre a todas las líneas del bloque en tiempo de compilación.</li><li>Carga cognitiva. El programador debe pre-calcular los márgenes para ajustar las comillas de cierre.</li></ul></td>
</tr>
</table>

*Nota*. La tabla demuestra que no existe un método donde el contenido sea independiente de la estética del código sin recurrir a funciones de limpieza o un esfuerzo cognitivo por parte del programador.

### C. La carga cognitiva de los parches sintácticos

El análisis anterior revela que los lenguajes actuales no ofrecen una solución nativa y limpia para las cadenas de texto puras y multilínea. El hecho de que un programador deba decidir entre múltiples tipos de literales dependiendo del contenido introduce una carga cognitiva innecesaria. Esta fragmentación obliga al desarrollador a realizar un cambio de contexto mental constante para prever cómo la máquina interpretará cada símbolo, en lugar de confiar en una sintaxis predecible y coherente.

### 4.1.2 Ambigüedad por operadores implícitos

En C, C++, AHK y Python se permite la concatenación de literales de cadena simplemente colocándolos uno al lado del otro. Aunque esto pretende reducir la necesidad del operador de concatenación, genera errores lógicos cuando se olvidan comas en estructuras de datos como listas o tuplas.

**Tabla 12**

*Análisis de fallas lógicas por concatenación implícita en Python*

| Código en Python | Interpretación | Resultado |
|---|---|---|
| `archivos = ["1.csv" "2.sql"]` | `['1.csv2.sql']` | Une dos nombres de archivos en uno solo por falta de coma |
| `print("Hola" " mundo")` | `Hola mundo` | Correcto, pero oculta la creación de una cadena intermedia por concatenación. |

*Nota.* La tabla demuestra cómo la falta de un operador visible en colecciones de datos puede llevar a la creación de elementos corruptos que el programador percibe como dos entidades separadas.

### 4.1.3 El caso Python

Uno de los problemas más insidiosos en Python es la sensibilidad del carácter de continuación de línea a los espacios en blanco posteriores. Si un programador inserta accidentalmente un espacio después de una barra invertida (`\`), el intérprete deja de reconocerlo como un marcador de continuidad y lanza un error sintáctico que es virtualmente invisible al ojo humano durante la revisión de código.

**Figura 4**

*Falla de continuidad visual por espacios de terminación en Python*

```
total_ventas = 1500 + \
               2300 + \
               450.50
```

*Nota.* En este ejemplo, si existe un espacio después del primer `\`, el intérprete fallará. El modelo mental del programador ve una suma continua, pero la máquina detecta un carácter inesperado invisible, rompiendo la sincronía entre la intención y la ejecución.

### 4.1.4 JavaScript y la ASI

JavaScript intenta ser un lenguaje de baja fricción mediante la inserción automática de puntos y comas (ASI). El objetivo es permitir que el programador omita los puntos y coma, delegando en el intérprete la tarea de inferir el final de las sentencias. Sin embargo, este proceso no es determinista desde la perspectiva del usuario, ya que las reglas de inserción son opacas y a menudo contraintuitivas (Alam, 2025).

#### A. El error de retorno silencioso

El fallo más crítico de la ASI ocurre cuando se introduce un salto de línea inmediatamente después de una palabra clave de control como `return`, `break` o `continue`. El motor de JavaScript inserta un punto y coma automáticamente, terminando la ejecución de la sentencia de forma prematura y dejando el código restante como una expresión aislada.

**Figura 5**

*Interrupción del flujo de retorno por reglas de ASI en JavaScript*

```
function crearUsuario() {
  return
  {
    nombre: "Dino"
  };
}
```

*Nota.* El programador intenta devolver un objeto con un formato visual claro. Sin embargo, JavaScript interpreta esto como `return;`, devolviendo *undefined.*

**B. Confusión de operadores y bloques**

La ASI también genera problemas de interpretación cuando una línea comienza con caracteres específicos.

**Tabla 13**

*Colisiones de intención sintáctica por ASI en JavaScript*

| Código | Interpretación | Resultado |
|---|---|---|
| `let nums = [1, 2]`<br>`[3, 4].forEach()` | `let nums = [1, 2][3, 4].forEach()` | Intenta acceder a un atributo `[3, 4]` dentro de `[1, 2]` |
| `let nums = [1, 2]`<br>`let back = nums`<br>`(nums).forEach()` | `let nums = [1, 2]`<br>`let back = nums(nums).forEach()` | Intenta llamar `nums` como función. |
| `i`<br>`++`<br>`j` | `i; ++j;` | Asume que el operador de incremento es de `j`, no de `i`. |

*Nota.* Estas fallas demuestran que la ASI no reduce la carga cognitiva, sino que la desplaza: el programador ya no tiene que escribir puntos y coma, pero ahora debe memorizar una lista compleja de excepciones para evitar errores silenciosos.

### 4.1.5 El caso Ruby

Ruby se define por su elegancia y su principio de mínima sorpresa, permitiendo una sintaxis que se asemeja al lenguaje natural. No obstante, su flexibilidad extrema en la omisión de paréntesis en llamadas a métodos crea zonas donde el intérprete y el humano divergen en su lectura del código.

### A. Conflictos de precedencia en bloques y argumentos

En Ruby, existen dos formas de pasar bloques a un método: mediante llaves `{}` o mediante las palabras clave `do...end`. La diferencia no es solo estética; tienen niveles de precedencia distintos. Cuando se omiten los paréntesis en llamadas anidadas, el destinatario del bloque cambia radicalmente según el delimitador usado.

**Figura 6**

*Ambigüedad de destino de bloque en Ruby sin paréntesis*

```
# El bloque se envía al método más cercano (metodo_2)
resultado = metodo_1 metodo_2 { puts "Soy de metodo_2" }

# El bloque se envía al método principal (metodo_1)
resultado = metodo_1 metodo_2 do
  puts "Soy de metodo_1"
end
```

*Nota.* La elección de un delimitador visual altera la jerarquía de ejecución. Esto viola el modelo mental del programador, quien percibe el bloque como una unidad lógica asociada a la acción más cercana, independientemente de si usa llaves o palabras clave.

### B. El riesgo de la omisión en métodos con múltiples argumentos

Cuando un método toma varios argumentos y uno de ellos es el resultado de otra llamada a un método, la falta de paréntesis genera un error de resolución. El intérprete no puede distinguir si el último argumento pertenece al método interno o al externo.

**Tabla 3**

*Inconsistencias en el paso de parámetros en Ruby*

| Código en Ruby | Resultado |
|---|---|
| `obj.metodo arg1, otro_metodo arg2, arg3` | Error de sintaxis. No sabe si `arg3` es de `metodo` o `otro_metodo` |
| `super` | Pasa todos los argumentos actuales implícitamente |
| `super()` | No pasa ningún argumento |

*Nota.* La tabla evidencia cómo la búsqueda de una sintaxis limpia puede resultar en una gramática que requiere que el programador actúe como un parser humano, calculando precedencias antes de escribir una línea.

### 4.1.6 El espacio como operador: Julia y Swift

Lenguajes más modernos como Julia y Swift han intentado resolver estas ambigüedades introduciendo una sensibilidad estricta al espacio horizontal, especialmente en el contexto de funciones y clausuras.

#### A. Julia y el conflicto entre funciones y matrices

En Julia, el espacio entre el nombre de una función y su paréntesis de apertura no es opcional si se quiere realizar una llamada. Esto se debe a que Julia utiliza el espacio como un operador de concatenación dentro de las definiciones de matrices.

**Figura 7**

*Conflicto de espaciado semántico en Julia*

```
# Interpreta 'rand' como un objeto y '(2, 3)' como una tupla
x = rand (2, 3)
```

*Nota.* Esta restricción es "horrenda" para algunos desarrolladores porque rompe la convención universal de que `f(x)` y `f (x)` son semánticamente equivalentes. Aunque, este comportamiento es principalmente atribuido al operador implícito de concatenación.

#### B. Swift y las clausuras posteriores

Swift introdujo la sintaxis de clausura posterior para permitir que el último argumento de una función (si es un bloque de código) se coloque fuera de los paréntesis. Si bien esto mejora la estética de APIs como SwiftUI, crea colisiones con las estructuras de control de flujo como `if` y `guard`.

**Tabla 14**

*Ambigüedades de clausuras posteriores en Swift*

| Código en Swift | Resultado |
|---|---|
| `guard lista.contains {`<br>`  $0.esValido`<br>`} else {`<br>`  return`<br>`}` | El compilador cree que el '`{`' de la clausura es el inicio del bloque `else`. No entiende dónde termina la condición y dónde empieza el cuerpo del `guard`. |
| `if check {$0 > 0}` | El compilador se confunde: ¿Es un `if` con una condición que es una clausura, o es un `if` cuya condición es `check` y el bloque es `{$0 > 0}`? |
| `let x = {print("Hola")}`<br>`x()`<br>`[1].map {...}` | Si se coloca una clausura inmediatamente después de una llamada que ya tiene una, el compilador intenta encadenarlas y a menudo pierde el hilo de qué valor está retornando cuál. |

*Nota.* El programador percibe la clausura como parte de la expresión de condición, pero el parser la ve como el inicio del cuerpo del bloque de control. Esta desconexión obliga al uso de paréntesis "de seguridad", invalidando el propósito original de la sintaxis simplificada.

### 4.1.7 Java vs. Python

Mientras que Python utiliza un enfoque funcional y declarativo, Java requiere una estructura jerárquica incluso para las tareas más simples. Esto aumenta la necesidad de escribir muchos caracteres para expresar una sola idea.

**Tabla 15**

*Comparativa de verbosidad y carga estructural en el filtrado de datos*

| Código | Carga cognitiva |
|---|---|
| `// Java` | El programador debe gestionar la infraestructura del lenguaje antes que la lógica de negocio. |

```
// Java
List<String> filtrados =
lista.stream()
   .filter(s -> s.startsWith("A"))
   .collect(Collectors.toList());
```

| Código | Carga cognitiva |
|---|---|
| `# Python`<br>`filtrados = [s for s in lista if s.startswith("A")]` | La sintaxis es declarativa y compacta. Se elimina el ruido visual. |

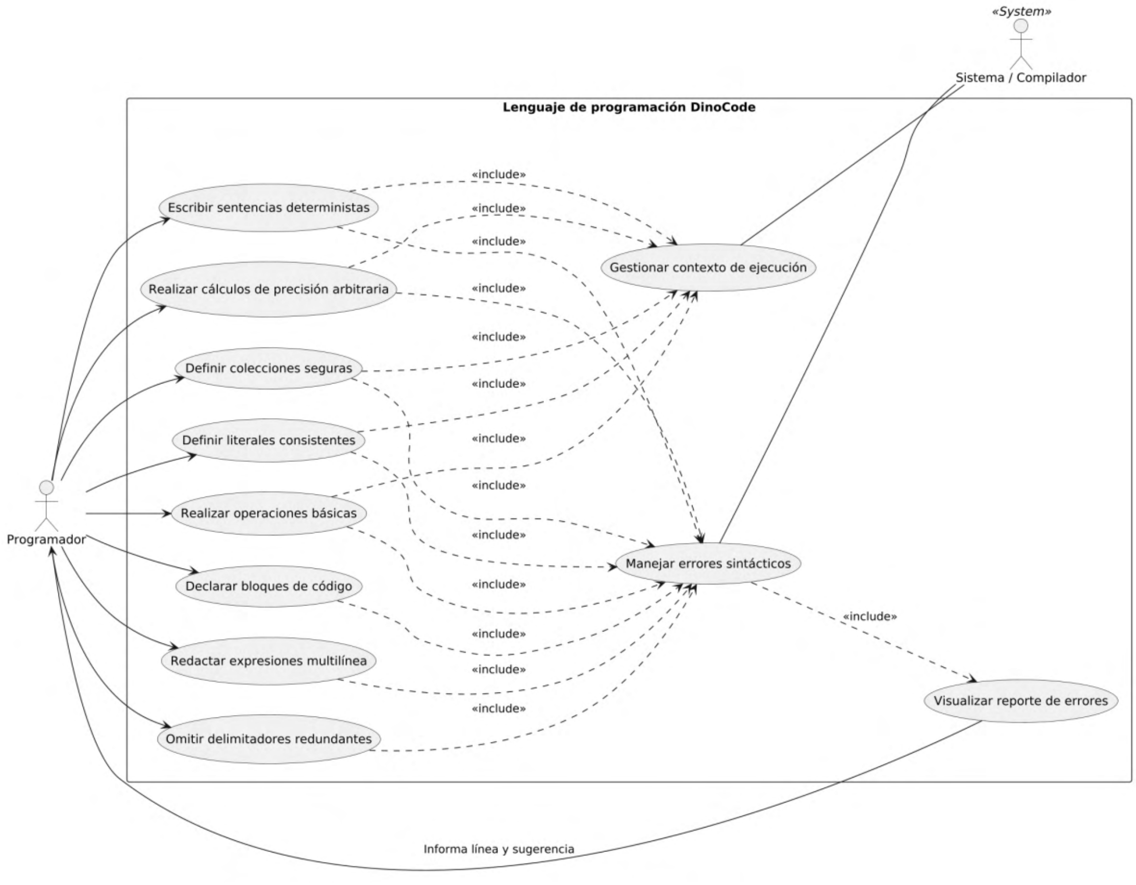
*Note.* El código de Java presenta un alto compromiso prematuro: el programador debe entender el concepto de Streams y Colectores antes de poder realizar un filtro básico. Python, en cambio, se alinea con el modelo mental de "seleccionar elementos que cumplan una condición".

## 4.2. Análisis de requerimientos

### 4.2.1 Diagrama de casos de uso

**Figura 8**

*Diagrama de casos de uso del lenguaje de programación DinoCode*



*Nota*. Elaboración propia mediante el lenguaje de modelado PlantUML.

### 4.2.2 Requerimientos funcionales

**Tabla 16**

*Requerimiento funcional de terminadores deterministas*

| Campo | Descripción |
|---|---|
| ID | RF-01 |
| Nombre | Escribir sentencias deterministas |
| Descripción | El sistema debe permitir al programador establecer el fin de una instrucción con reglas explícitas o de inferencia predecible. |
| Prioridad | Alta |
| Criterio de aceptación | El compilador no debe insertar terminadores que interrumpan el flujo lógico de la ejecución. |

**Tabla 17**

*Requerimiento funcional de literales consistentes*

| Campo | Descripción |
|---|---|
| ID | RF-02 |
| Nombre | Definir literales consistentes |
| Descripción | El sistema debe garantizar al programador que las cadenas de texto en línea y multilínea (con o sin sangría) sean interpretadas consistentemente en todas sus formas (rutas de archivos, expresiones regulares, plantillas para interpolación de datos y literales puros). |
| Prioridad | Alta |
| Criterio de aceptación | El manejo de literales no debe incurrir en la contaminación de datos por sangría, dependencias en tiempo de ejecución, ni en el abuso de delimitadores o símbolos. |

**Tabla 18**

*Requerimiento funcional de expresiones multilínea*

| Campo | Descripción |
|---|---|
| ID | RF-03 |
| Nombre | Redactar expresiones multilínea |
| Descripción | El sistema debe permitir al programador extender una operación lógica por varias líneas de código sin necesidad de marcadores visuales que sean sensibles a caracteres invisibles. |
| Prioridad | Alta |
| Criterio de aceptación | El sistema debe interpretar la continuidad del flujo basándose en el estado de la expresión actual, ignorando espacios en blanco accidentales al final de la línea. |

**Tabla 19**

*Requerimiento funcional de colecciones seguras*

| Campo | Descripción |
|---|---|
| ID | RF-04 |
| Nombre | Definir colecciones seguras |
| Descripción | El sistema debe impedir la unión accidental de elementos en estructuras de datos (listas o diccionarios) debido a la ausencia de separadores explícitos. |
| Prioridad | Media |
| Criterio de aceptación | El sistema debe ser capaz de diferenciar entidades separadas dentro de una colección basándose en su estructura, no solo en la puntuación. |

**Tabla 20**

*Requerimiento funcional para declarar bloques de código multilínea*

| Campo | Descripción |
|---|---|
| ID | RF-05 |
| Nombre | Declarar bloques de código |
| Descripción | El sistema debe proveer al programador de un mecanismo para jerarquizar bloques de código de forma legible. |
| Prioridad | Media |
| Criterio de aceptación | El sistema debe interpretar bloques de código anidados como cuerpos de funciones, condicionales y bucles. |

**Tabla 21**

*Requerimiento funcional para omitir delimitadores redundantes*

| Campo | Descripción |
|---|---|
| ID | RF-06 |
| Nombre | Omitir delimitadores redundantes |
| Descripción | El sistema debe permitir al programador omitir paréntesis en contextos claros como condicionales, llamadas a funciones y métodos de objetos para reducir el ruido visual. |
| Prioridad | Media |
| Criterio de aceptación | El compilador debe resolver la precedencia de operadores y el alcance de las expresiones en condicionales y llamadas sin requerir paréntesis obligatorios, siempre que no exista ambigüedad sintáctica. |

**Tabla 22**

*Requerimiento funcional para realizar operaciones básicas*

| Campo | Descripción |
|---|---|
| ID | RF-07 |
| Nombre | Realizar operaciones básicas |
| Descripción | El sistema debe permitir al programador realizar operaciones numéricas, lógicas y la definición de estructuras como arreglos y objetos. |
| Prioridad | Media |
| Criterio de aceptación | El sistema debe ejecutar correctamente operaciones aritméticas estándar, evaluaciones booleanas y la creación/acceso a objetos definidos por el usuario. |

**Tabla 23**

*Requerimiento funcional para realizar cálculos de precisión arbitraria*

| Campo | Descripción |
|---|---|
| ID | RF-08 |
| Nombre | Realizar cálculos de precisión arbitraria |
| Descripción | El sistema debe permitir al programador realizar cálculos con enteros de precisión arbitraria para evitar desbordamientos en operaciones con números extremadamente grandes. |
| Prioridad | Baja |
| Criterio de aceptación | El sistema debe soportar operaciones aritméticas con enteros que superen el límite de los enteros nativos sin pérdida de precisión ni errores de representación. |

### 4.2.3 Requerimientos no funcionales

**Tabla 24**

*Requerimiento no funcional para la minimización de la carga cognitiva*

| Campo | Descripción |
|---|---|
| ID | RNF-01 |
| Atributo | Minimización de la carga cognitiva |
| Descripción | La sintaxis debe ser diseñada para que el programador no necesite realizar procesos de "parsing mental" para prever el comportamiento del sistema ante símbolos comunes. |
| Prioridad | Crítica |

**Tabla 25**

*Requerimiento no funcional para garantizar predictibilidad visual*

| Campo | Descripción |
|---|---|
| ID | RNF-02 |
| Atributo | Predictibilidad visual |
| Descripción | El código fuente debe procesarse semánticamente de la misma forma en que el programador lo percibe visualmente durante la lectura. |
| Prioridad | Alta |

**Tabla 26**

*Requerimiento no funcional para garantizar seguridad de memoria y eficiencia técnica*

| Campo | Descripción |
|---|---|
| ID | RNF-03 |
| Atributo | Seguridad de memoria y eficiencia técnica |
| Descripción | El núcleo del compilador/intérprete debe implementarse utilizando el lenguaje Rust para garantizar la gestión segura de la memoria y prevenir errores de bajo nivel sin sacrificar el rendimiento. |
| Prioridad | Alta |
| Criterio de aceptación | El motor de DinoCode debe ejecutarse sobre una infraestructura que elimine fallos de segmentación mediante el modelo de ownership y abstracciones de costo zero de Rust. |

## 4.3. Arquitectura del sistema

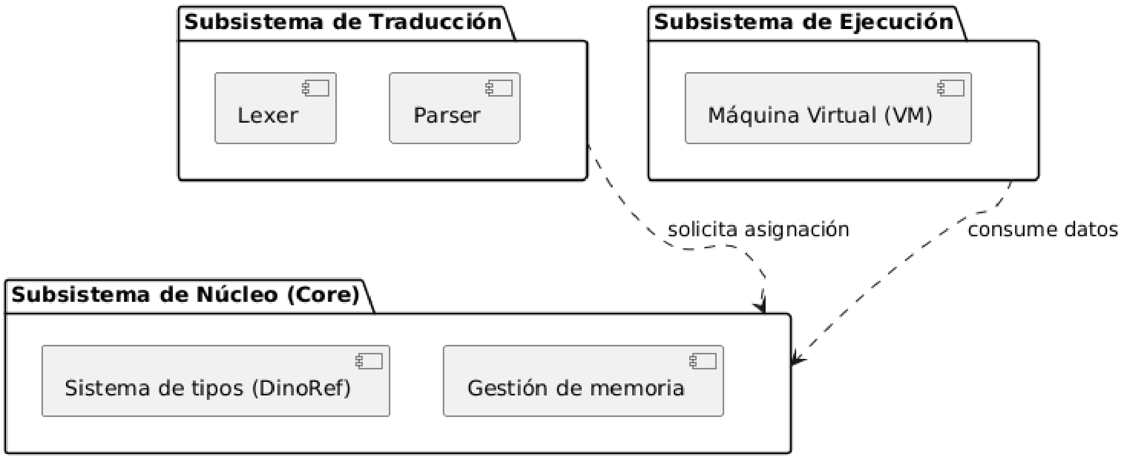
DinoCode implementa una arquitectura modular de alta cohesión fundamentado en la separación de responsabilidades. Esta estructura garantiza el aislamiento por componentes, donde cada módulo gestiona una faceta específica del ciclo de vida de la ejecución. Bajo esta configuración, el sistema se categoriza como un lenguaje de 4.5ª generación (4.5gl), representando una evolución técnica frente a la taxonomía clásica de la Tabla 1. Este enfoque híbrido integra la eficiencia operativa de los lenguajes de cuarta generación con la inferencia de intención característica de la quinta generación.

### 4.3.1 Vista lógica

A nivel macro, el sistema se divide en tres bloques de responsabilidad. El núcleo (Core) actúa como una fuente de verdad para la gestión de datos y estructuras de bajo nivel, mientras que los subsistemas de traducción y ejecución operan sobre él de forma desacoplada.

**Figura 9**

*Diagrama de paquetes de la arquitectura lógica de DinoCode*



Como se detalla en la Figura 9, la arquitectura garantiza que el subsistema de núcleo sea independiente. Esta decisión de diseño permite que la máquina virtual pueda ejecutar código pre-compilado o generado dinámicamente, siempre que se respeten los contratos de memoria definidos en el núcleo.

### A. Módulos del subsistema de núcleo

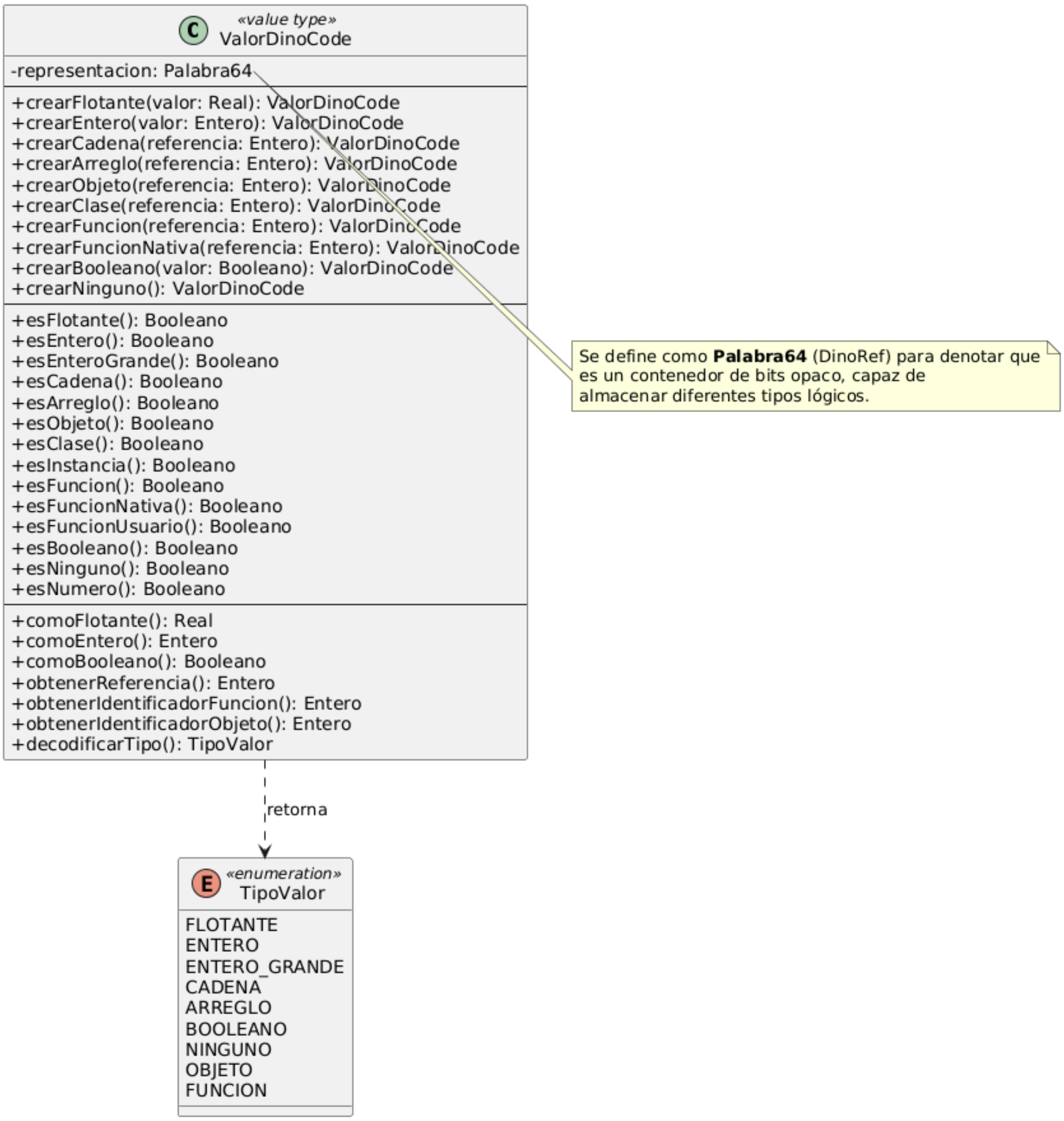
El núcleo es el componente más crítico del sistema. Su diseño garantiza que los datos tengan una representación única en memoria.

### Módulo de tipos

Este módulo define la unidad mínima de información (DinoRef). Se basa en una representación unificada de 64 bits mediante la técnica de NaN Boxing, permitiendo que los tipos primitivos se manejen directamente en los registros de la CPU.

**Figura 10**

*Diagrama de clases del módulo de tipos*

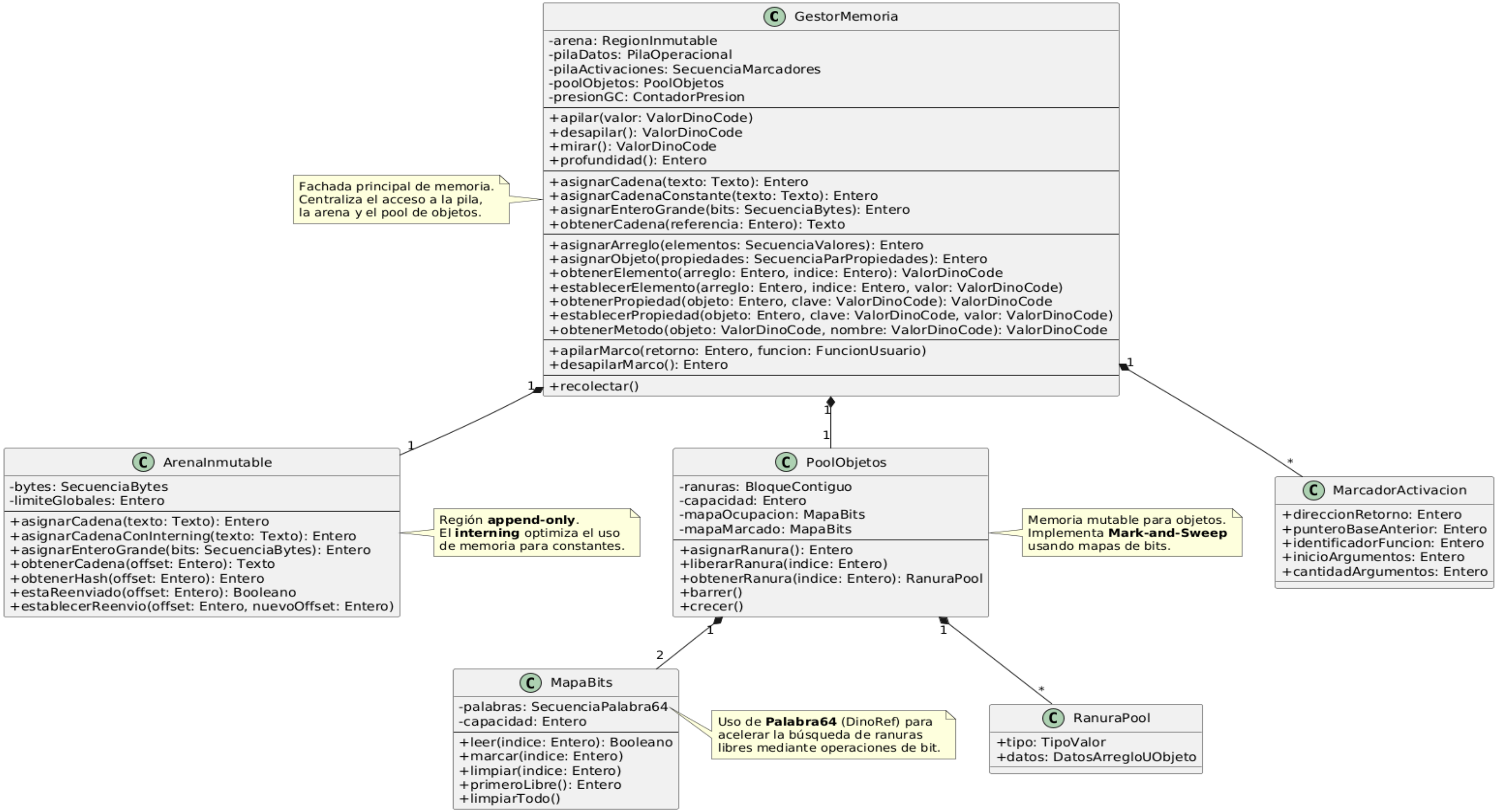


### Módulo de gestión de memoria

Diseñado como un sistema híbrido, este módulo coordina el almacenamiento físico. La Arena Inmutable se encarga de las constantes detectadas en la traducción y de los tipos inmutables generados dinámicamente en la ejecución, mientras que el Pool de Objetos gestiona las estructuras mutables creadas en tiempo de ejecución.

**Figura 11**

*Diagrama de clases del módulo de memoria*



Fachada principal de memoria.
Centraliza el acceso a la pila,
la arena y el pool de objetos.

Región **append-only**.
El **interning** optimiza el uso
de memoria para constantes.

Memoria mutable para objetos.
Implementa **Mark-and-Sweep**
usando mapas de bits.

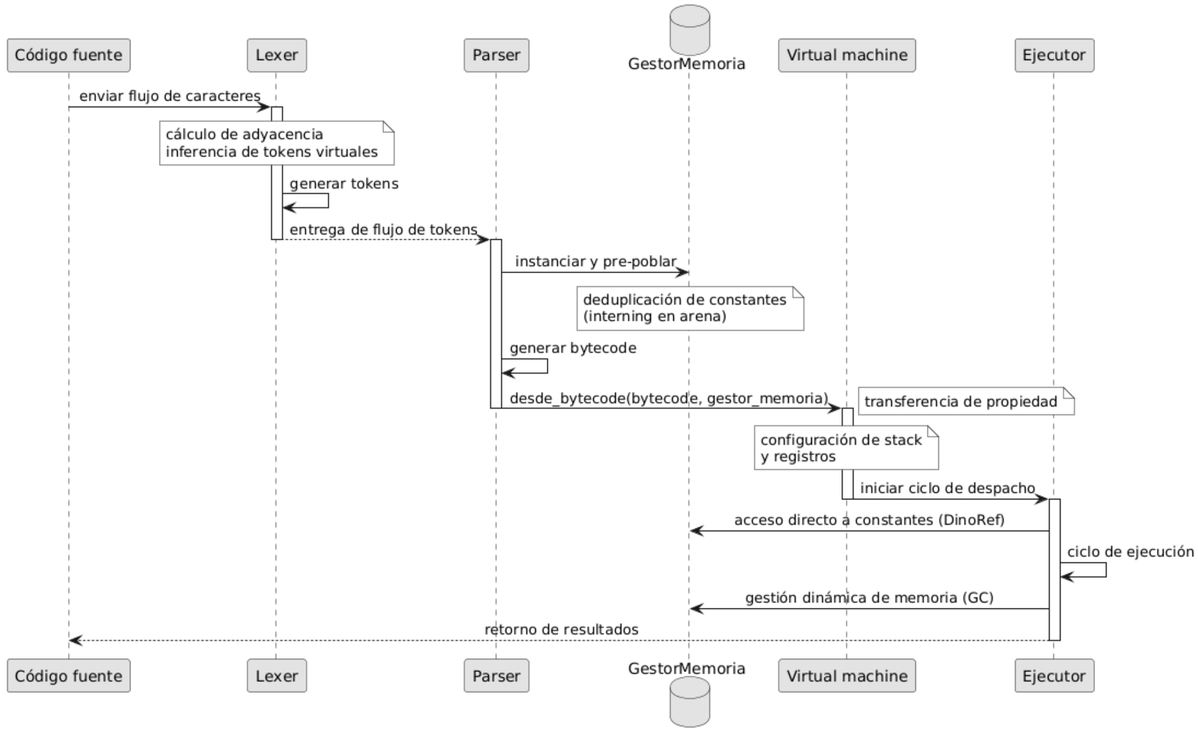
Uso de **Palabra64** (DinoRef) para
acelerar la búsqueda de ranuras
libres mediante operaciones de bit.

### 4.3.2 Vista de proceso

El *pipeline* de DinoCode destaca por su modelo de transferencia de propiedad, donde los componentes autónomos colaboran en un flujo de un solo paso.

**Figura 12**

*Diagrama de secuencia de la interfaz de comunicación entre los componentes del intérprete*



Como ilustra la Figura 12, el diseño del pipeline permite que el parser utilice el gestor de memoria para depositar constantes de forma inmediata. Al terminar la traducción, la VM recibe el contexto de memoria ya "caliente", lo que reduce la latencia de arranque.

### 4.3.3 Vista de desarrollo

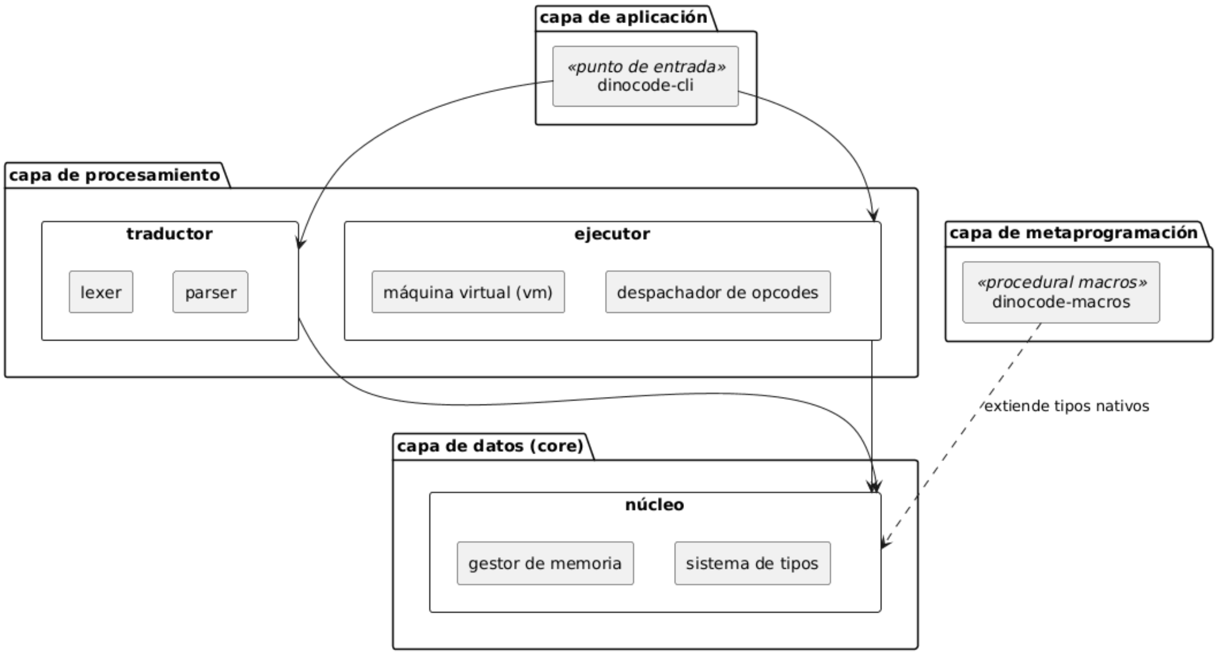
Siguiendo el enfoque de ISBC, DinoCode se estructura en componentes autónomos distribuidos en un espacio de trabajo en Rust (*workspace*), lo que facilita la reutilización y permite integrar funciones nativas mediante metaprogramación.

#### 4.3.3.1 Estructura de componentes y capas de software

La arquitectura física se organiza en capas donde los componentes de nivel superior utilizan los servicios de las capas inferiores. Esta estructura garantiza que el núcleo del lenguaje permanezca agnóstico a las interfaces de usuario o a los métodos de traducción específicos, protegiendo la integridad de los datos.

**Figura 13**

*Diagrama de capas de la arquitectura del sistema*



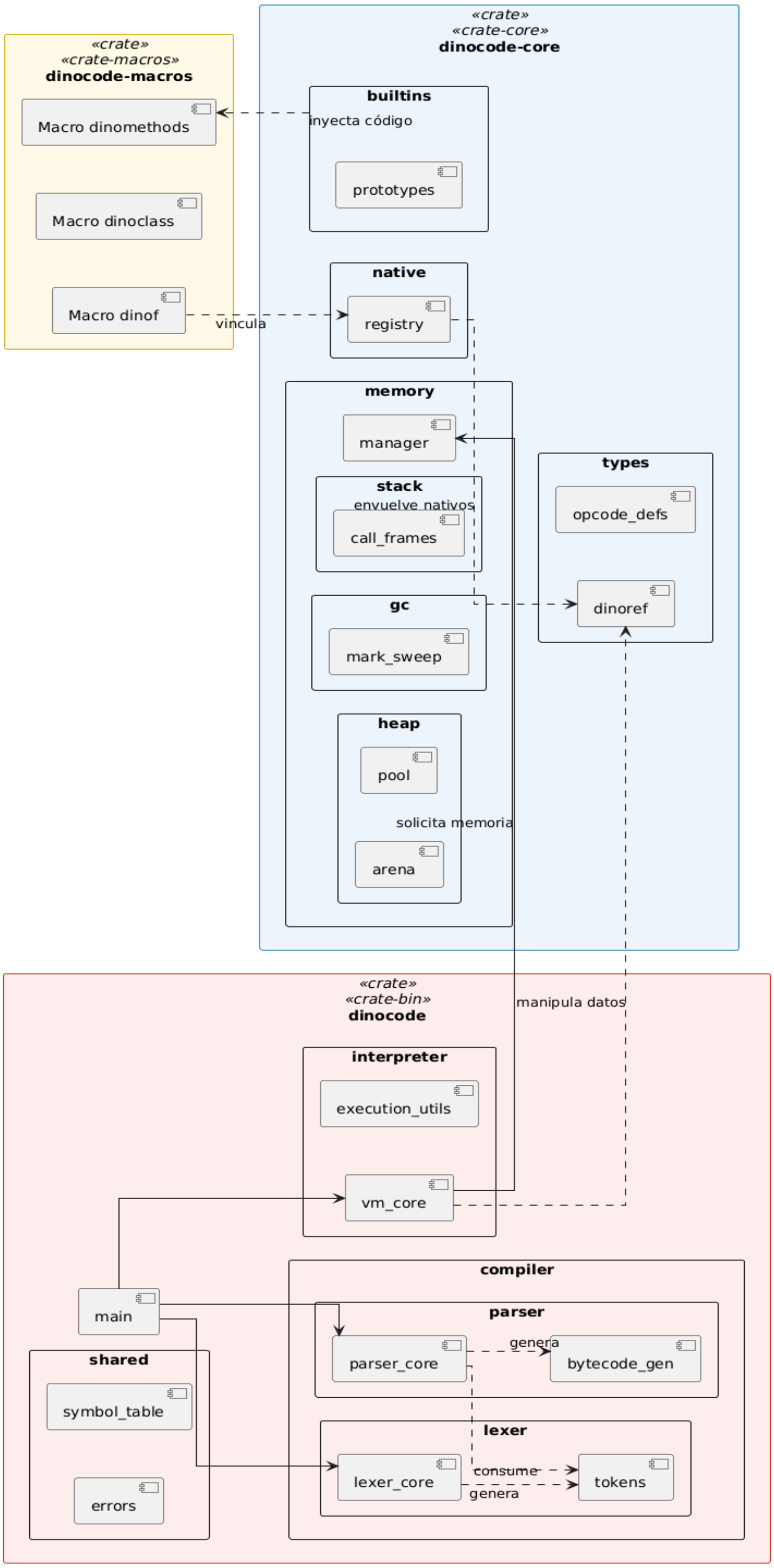
*Nota*. Cualquier cambio en la gramática dentro de la capa de procesamiento no afecta la integridad de la representación de los datos.

#### 4.3.3.2 Estructura física de módulos y dependencias

La siguiente figura detalla la jerarquía interna de los crates de Rust y cómo se relacionan los módulos de bajo nivel con el motor de ejecución. Esta organización permite que cada pieza sea tratada como una unidad independiente, facilitando su prueba y mantenimiento.

**Figura 14**

*Diagrama de componentes y organización jerárquica del sistema*



## 4.4. Especificación formal de la sintaxis de DinoCode

La gramática de DinoCode se define formalmente utilizando la notación EBNF (*Extended Backus-Naur Form*). Esta especificación detalla las reglas de producción que dictan cómo los componentes léxicos se combinan para formar sentencias y expresiones válidas dentro del lenguaje, abstrayendo los detalles de implementación del transductor.

### 4.4.1 Definición de tokens y léxico base

La unidad fundamental de la sintaxis de DinoCode es el token. El analizador léxico reconoce identificadores con soporte para caracteres extendidos, permitiendo una semántica natural con soporte de español, así como una amplia gama de representaciones numéricas.

**Figura 15**

*Reglas de producción EBNF para identificadores y literales numéricos*

```
identificador ::= [a-zA-Z_$ñáéíóúÁÉÍÓÚ][a-zA-Z0-9_$ñáéíóúÁÉÍÓÚ]*

numero_decimal  ::= [0-9]+('_'[0-9]+)*

numero_hex      ::= '0x'[0-9a-fA-F]+('_'[0-9a-fA-F]+)*

numero_binario  ::= '0b'[01]+('_'[01]+)*

numero_flotante ::= [0-9]+('.'[0-9]+)?([eE][+-]?[0-9]+)?

bigint          ::= (numero_decimal | numero_hex | numero_binario) 'n'

literal ::= numero_decimal | numero_hex | numero_binario |
numero_flotante

          | bigint | string_literal | booleano | 'none'
```

Como se observa en la Figura 15, los literales numéricos permiten el uso de guiones bajos (_) como separadores visuales para mejorar la legibilidad. Los números de precisión arbitraria (*BigInts*) se distinguen sintácticamente mediante el sufijo `n`.

### 4.4.2 Palabras reservadas, operadores y delimitadores

El lenguaje define un conjunto estricto de palabras clave y símbolos que estructuran la lógica de control y las operaciones aritmético-lógicas.

**Figura 16**

*Especificación de símbolos, operadores y palabras reservadas*

```
palabras_reservadas ::= 'true' | 'false'|'none'|'and'|'or'|'not'|
                      'as' | 'is' | 'in' | 'if' | 'elif' | 'else' |
                      'while'|'for' | 'return' | 'break' | 'continue'


operadores ::= '+' | '-' | '*' | '/' | '//' | '%' | '**'|'++' | '--' |
             '=' | '+=' | '-=' | '*=' | '/=' | '//=' | '.=' | '<-' |
             '==' | '!=' | '=~' | '>' | '<' | '>=' |'<='|'&&'| '||' |
             '!' | '&' | '|' | '^' | '~' | '<<' | '>>' | '.' | ':' |
             '?' | '::' | 'in' | 'as' | 'is'


delimitadores ::= '(' | ')' | '[' | ']' | '{' | '}' |'"'|"'"|','| ';'
```

*Nota*. DinoCode introduce operadores específicos como `<-` para la asignación interactiva con entrada de usuario, `::` para la declaración de clases, y el símbolo `>` que actúa de forma dual como operador relacional y como redireccionador de bloques literales.

### 4.4.3 Estructura del programa y declaraciones

Un programa en DinoCode es una secuencia de declaraciones. El lenguaje prescinde de las llaves tradicionales (`{ }`) para delimitar ámbitos; en su lugar, la gramática define la estructura de bloques basándose en tokens lógicos de indentación (indent y dedent).

**Figura 17**

*Gramática para la estructura global del programa y sentencias de control*

```
programa ::= {declaracion}


declaracion ::= declaracion_variable
              | declaracion_funcion
              | declaracion_clase
```

```
                | sentencia_control

                | expresion_fin


declaracion_variable ::=identificador operador_asignacion expresion_fin

                         | acceso_array operador_asignacion expresion_fin

                         | acceso_miembro operador_asignacion expresion_fin

                         | identificador expresion_redireccion

                         | acceso_array expresion_redireccion

                         | acceso_miembro expresion_redireccion


declaracion_funcion ::= ':' identificador [lista_parametros] bloque

declaracion_clase   ::= '::' identificador bloque


lista_parametros ::= {identificador espacio} identificador

bloque           ::= 'indent' {declaracion} 'dedent'


sentencia_control ::= 'if' expresion bloque {'elif' expresion bloque} ['else'
bloque]

                    | 'while' expresion bloque

                    | 'for' identificador 'in' expresion bloque
```

Esta estructura obliga a que la topografía visual del código fuente coincida estrictamente con la jerarquía semántica, mejorando la legibilidad inherente de las sentencias de control.

### 4.4.4 Jerarquía de expresiones

La gramática de expresiones está estructurada en capas sucesivas para forzar la asociatividad sin necesidad de definir la precedencia en el nivel de implementación sintáctica estricta.

**Figura 18**

*Jerarquía de derivación de expresiones y precedencia operativa*

```
expresion ::= expresion_asignacion

expresion_asignacion     ::= expresion_logica [operador_asignacion
expresion_asignacion]

expresion_logica         ::= expresion_or {('&&' | '||') expresion_or}

expresion_or             ::= expresion_and {'or' expresion_and}

expresion_and            ::= expresion_igualdad {'and'
expresion_igualdad}

expresion_igualdad       ::= expresion_comparacion {('==' | '!=' |
'=~') expresion_comparacion}

expresion_comparacion    ::= expresion_concatenacion {('>' | '<' |
'>=' | '<=' | 'in' | 'not in') expresion_concatenacion}

expresion_concatenacion  ::= expresion_aditiva {'.' expresion_aditiva}

expresion_aditiva        ::= expresion_multiplicativa {('+' | '-')
expresion_multiplicativa}

expresion_multiplicativa ::= expresion_potencia {('*' | '/' | '//' |
'%') expresion_potencia}

expresion_potencia       ::= expresion_unaria {'**' expresion_unaria}

expresion_unaria         ::= [operador_unario] expresion_primaria
```

Para formalizar la manera en la que los operadores son evaluados matemáticamente en expresiones complejas, la sintaxis respeta la siguiente tabla de precedencia, ordenada de mayor a menor prioridad:

**Tabla 27**

*Jerarquía de precedencia de operadores en DinoCode*

| Nivel | Operadores | Asociatividad | Reglas especiales | ¿Es unario? |
|---|---|---|---|---|
| 16 | as<br>is | Izquierda | Operadores de tipo | No |
| 15 | ++<br>-- | Derecha | Incremento<br>Decremento | Sí/No |
| 14 | **<br>^ | Derecha | Potencia | No |
| 13 | !<br>+<br>- | Derecha |  | Sí |
| 12 | * / % // | Izquierda | Aritméticos multiplicativos | No |
| 11 | +<br>- | Izquierda | Aritméticos aditivos | No |
| 10 | << >> & \| | Izquierda | Operadores de bit a bit | No |
| 9 | . | Izquierda | Acceso a miembros | No |
| 8 | < <=<br>> >=<br>In | Izquierda | Comparación | No |
| 7 | == !=<br>=~ | Izquierda | Igualdad y pattern matching | No |
| 6 | && | Izquierda | Conjunción lógica | No |
| 5 | \|\| | Izquierda | Disyunción lógica | No |
| 2 | =<br>+=<br>-=<br>*=<br>/=<br>%=<br>//=<br>.=<br><- | Derecha | Operadores de asignación | No |

### 4.4.5 Estructuras de datos y objetos

DinoCode provee una sintaxis declarativa para la construcción y acceso a colecciones de datos, introduciendo flexibilidad en la definición de propiedades de objetos.

**Figura 19**

Gramática para la estructura de datos y objetos

```
acceso_array      ::= expresion_primaria '[' expresion ']'

creacion_array    ::= '[' [lista_elementos] ']'

lista_elementos   ::= expresion {espacio expresion}


creacion_objeto   ::= '{' [lista_propiedades] '}'

lista_propiedades ::= propiedad {espacio propiedad}

propiedad         ::= identificador expresion

                    | identificador ':' expresion
```

*Nota*. La regla de propiedad en la creación de objetos demuestra que los dos puntos (:) son opcionales en la declaración de diccionarios, reduciendo el ruido visual.

### 4.4.6 Sintaxis de invocación y métodos implícitos

Una de las características gramaticales de la propuesta es la omisión opcional de paréntesis en las llamadas a funciones, siempre que el contexto garantice la ausencia de ambigüedad.

**Figura 20**

*Reglas de producción para llamadas a funciones y el operador dollar call*

```
llamada_funcion ::= expresion_primaria '(' [lista_argumentos] ')'

                  | expresion_primaria [lista_argumentos_implicitos]

                  | identificador [lista_argumentos_implicitos]

                  | metodo_implicito

                  | dollar_call
```

metodo_implicito ::= identificador '.' identificador [lista_argumentos_implicitos]

dollar_call ::= '$' '(' expresion_primaria [lista_argumentos_implicitos] ')' lista_argumentos ::= expresion {',' expresion}

lista_argumentos_implicitos ::= expresion {espacio expresion}

Nota. El llamado *Dollar Call* se introduce como una herramienta gramatical explícita para forzar el encapsulamiento y evaluación previa de una expresión antes de pasarla como argumento, lo cual permite mntener una sintaxis visualmente idéntica a los llamados implícitos dentro de expresiones ambiguas.

### 4.4.7 Cadenas de texto y bloques de captura (*Templates*)

La sintaxis separa semánticamente las cadenas de texto según su delimitador para evitar la corrupción por secuencias de escape no deseadas.

**Figura 21**

*Especificación formal para literales de cadena y bloques de captura multilínea*

```
string_literal ::= '"' {caracter | interpolacion | escape_doble} '"'
                 | "'" {caracter} "'"


escape_doble  ::= '\' '"' | '\' '\' | '\' 'n' | '\' 't' | '\' 'r'

interpolacion ::= '$' identificador
                | '${' expresion '}'


template_string   ::= '>' expresion_destino 'newline' bloque_indentado
                    | 'lstring' {interpolacion | string_literal} 'rstring'

bloque_indentado  ::= {linea_indentada}

expresion_destino ::= identificador | acceso_array | acceso_miembro
```

Las reglas de derivación expuestas en la Figura 21 establecen tres modos de tratamiento de literales:

1. **Comillas dobles (")**

   Habilitan la interpolación de variables o expresiones y evalúan caracteres de escape.

2. **Comillas simples (')**

   Representan literales puros, útiles para rutas de archicos o expresiones regulares, donde cualquier secuencia de escape es ignorada. Adicionalmente, tanto en dobles como en simples, la duplicación del delimitador (ej. "”) se interpreta gramaticalmente como un carácter escapado y no como un cierre de cadena.

3. **Bloques *template* (>)**

   Permiten capturar un conjunto de líneas indentadas y asignarlo directamente a una expresión destino (como una variable o la propiedad de un objeto), limpiando carácteres que estén fuera de la indentación automáticamente y habilitando únicamente la interpolación.

Esta especificación EBNF define las reglas gramaticales de DinoCode de forma estática. No obstante, la omisión de delimitadores físicos introduce colisiones sintácticas que un análisis puramente gramatical no puede resolver por sí mismo. Para garantizar el determinismo en la traducción, el sistema delega la resolución final al motor de inferencia, detallado en la sección 4.6.

## 4.5. Diseño del transductor léxico

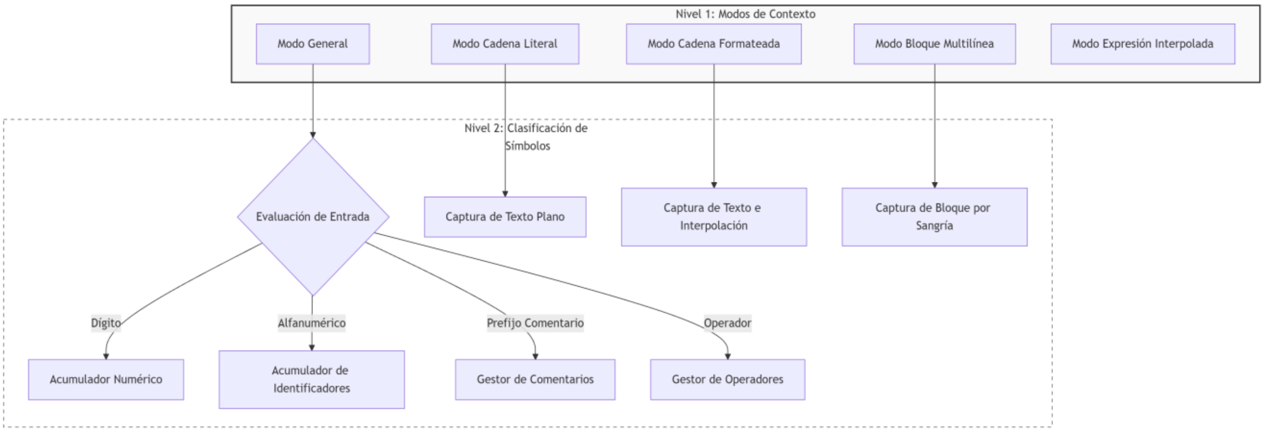
### 4.5.1 Contextos y acumuladores

Antes de detallar el flujo de datos, es necesario establecer la distinción jerárquica que gobierna el componente. El diseño del analizador léxico se fundamenta en una estructura que separa la interpretación del entorno de la recolección de datos. Como se ilustra en la Figura 22, el sistema opera en dos niveles de abstracción: los modos de contexto y los mecanismos de clasificación.

El modo de contexto actúa como un estado de control superior que define la gramática activa en un momento dado (por ejemplo, si el sistema interpreta instrucciones estándar o el contenido de una cadena de texto). Por su parte, los mecanismos de clasificación funcionan como acumuladores transitorios que agrupan los caracteres según su naturaleza técnica mientras se permanece en un modo determinado. Esta separación garantiza que un mismo símbolo pueda ser interpretado de formas distintas dependiendo de la región del código fuente en la que se encuentre.

**Figura 22**

*Contextos y acumuladores de símbolos.*

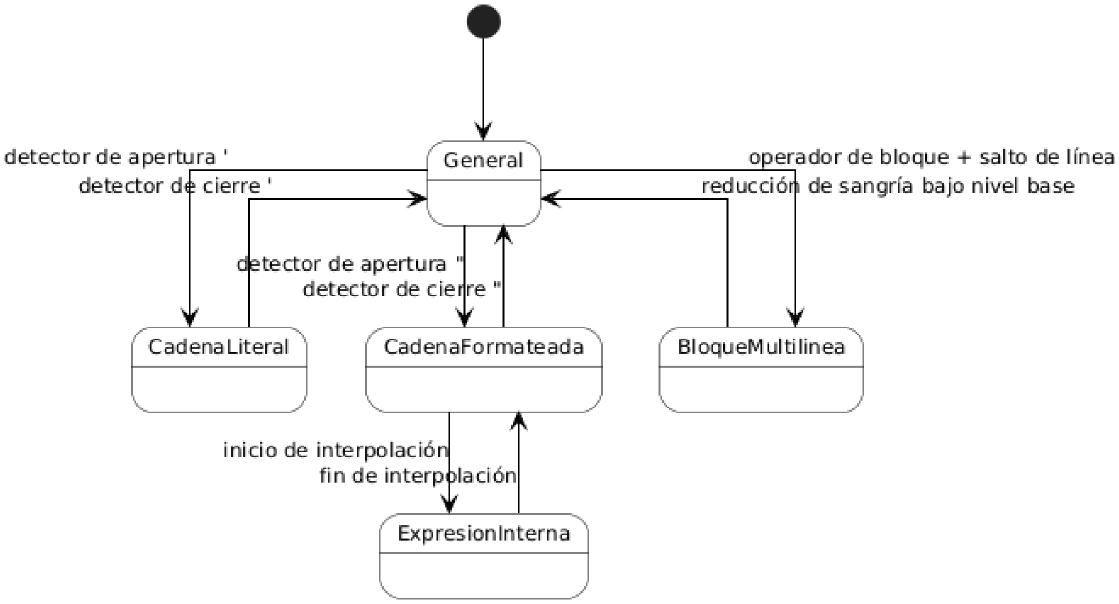


## 4.5.2 Dinámica de estados y recursividad

La transición entre los diferentes contextos no es lineal, sino que responde a un modelo de pila que permite la recursividad. La Figura 23 describe el autómata de estados finitos que regula el flujo del analizador. Mediante operaciones de apilado y desapilado de estados, el sistema puede procesar estructuras complejas, como expresiones aritméticas anidadas dentro de una cadena de texto formateada.

**Figura 23**

*Autómata de estados finitos para la gestión de contextos recursivos.*



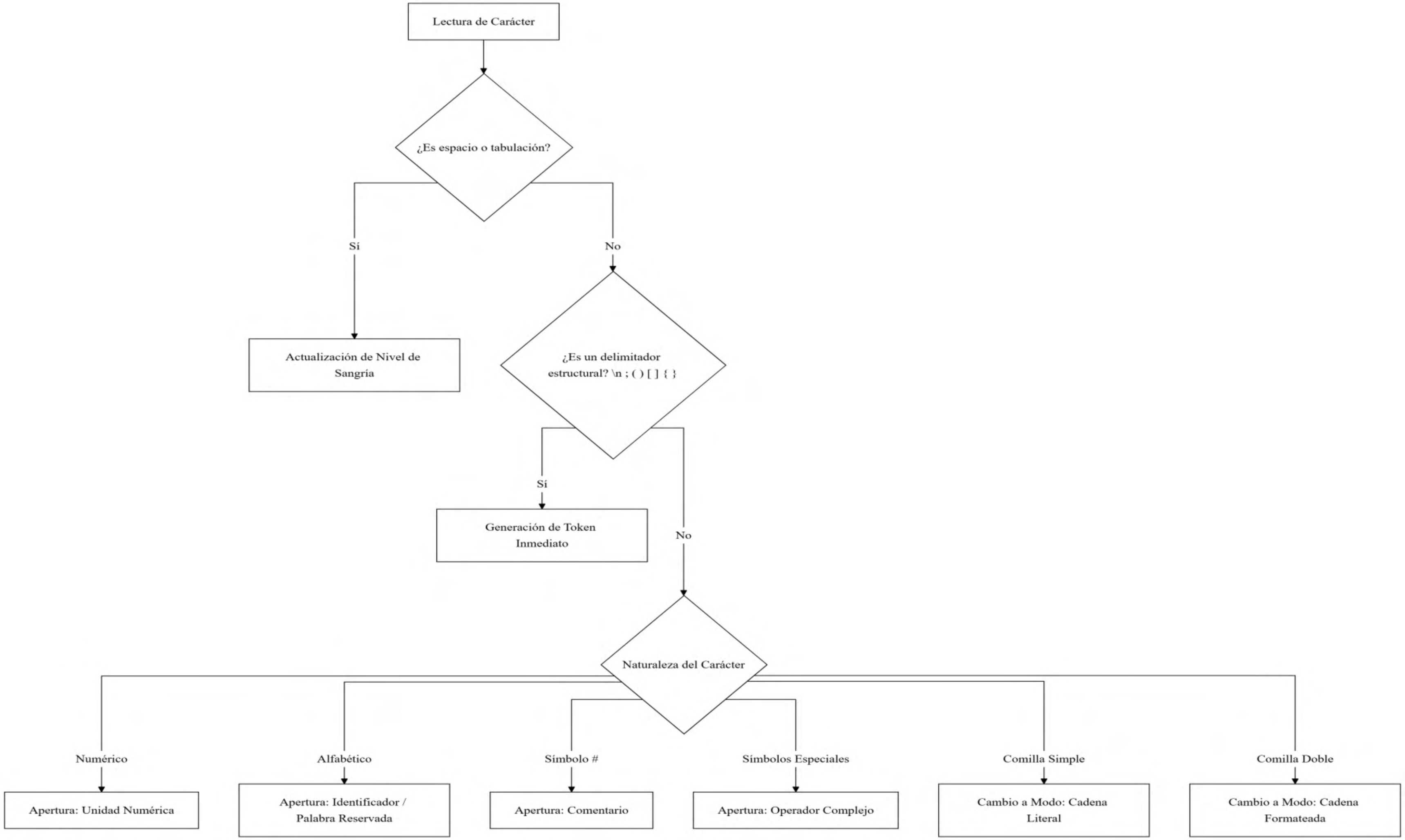
### 4.5.3 Reglas de procesamiento y emisión

#### 4.5.3.1 Algoritmo de clasificación y apertura de unidades léxicas

El núcleo operativo del analizador reside en la clasificación inmediata de cada carácter de entrada cuando el sistema se encuentra en el modo general. La Figura 24 detalla la lógica de bifurcación que permite al sistema identificar si un carácter inicia una nueva unidad léxica o si debe ser tratado como un delimitador estructural. Esta fase es crítica, ya que pre-clasifica la entrada antes de invocar los mecanismos especializados de gestión de memoria y resolución de tokens.

**Figura 24**

*Algoritmo de clasificación de entrada y apertura de unidades léxicas*

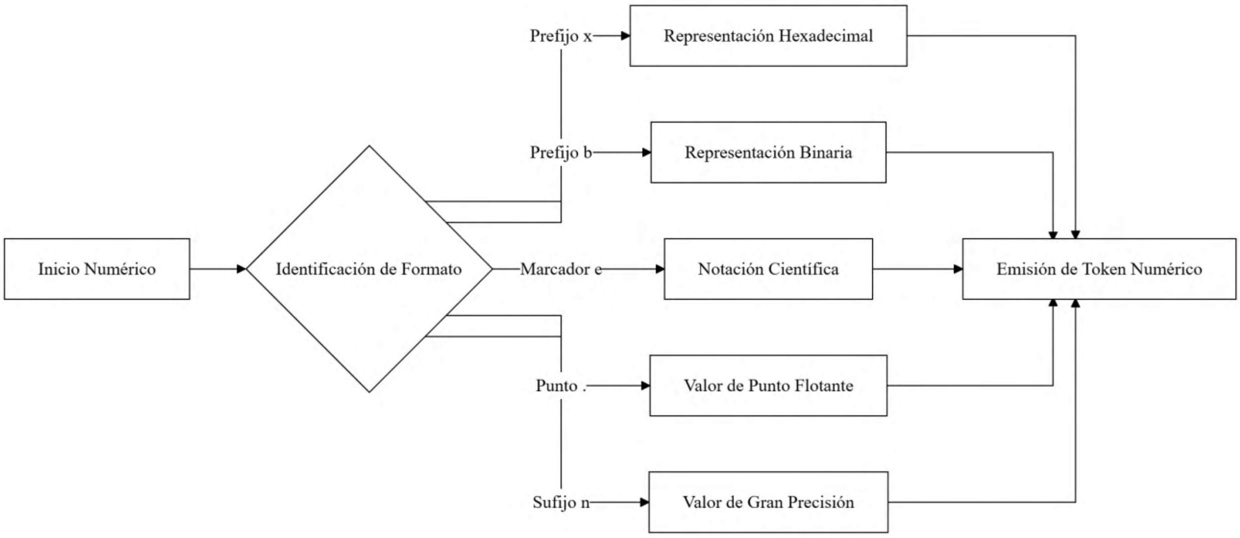


#### 4.5.3.2 Gestión de literales numéricos complejos

Debido a que el lenguaje soporta múltiples representaciones numéricas, el sistema implementa una lógica de refinamiento tras la detección inicial del dígito. Como se observa en la Figura 25, el sistema analiza marcadores de formato adicionales para distinguir entre bases numéricas (como hexadecimal o binaria), notación científica o tipos de precisión arbitraria. Este procedimiento asegura que los literales sean transformados en unidades con el tipado exacto requerido por las fases posteriores del compilador.

**Figura 25**

*Diagrama de flujo para la resolución de literales numéricos complejos*



## 4.6. Motor de inferencia y gramática de adyacencia

### 4.6.1 Introducción a la gramática de adyacencia

El diseño de DinoCode propone una ruptura paradigmática con el modelo de análisis sintáctico tradicional. Desde la formulación de la jerarquía de Chomsky, los lenguajes de programación han dependido de las gramáticas libres de contexto (GLC) para definir su estructura, lo que impone una segregación estricta entre un análisis léxico amnésico y un análisis sintáctico jerárquico (Lovelle, 2001; Vladimir, n.d., pp. 81–86). Esta independencia del contexto obliga el uso de delimitadores explícitos para resolver ambigüedades, ignorando la información semántica latente en la distribución espacial y física del texto fuente.

DinoCode introduce el concepto de gramática de adyacencia, implementada mediante un transductor de estados finitos enriquecido (AEFT, por sus siglas en inglés: augmented finite state transducer). A diferencia de las gramáticas sensibles al contexto (GSC) convencionales, este modelo restringe la sensibilidad a una vecindad inmediata y persistente (Pardo, 2016; UNCPBA, 2009b). El transductor opera como una función de transición extendida que no solo reconoce *lexemas*, sino que transforma el flujo de entrada basándose en un registro de estados previos.

Formalmente, el comportamiento del transductor en un instante $n$ se define mediante la función de transición:

$$\delta(q, T_n, \Phi_{n-1}) \rightarrow (q', T'_n, \Phi_n)$$

Donde:

$q$ representa el estado interno del autómata

$T_n$ el lexema capturado

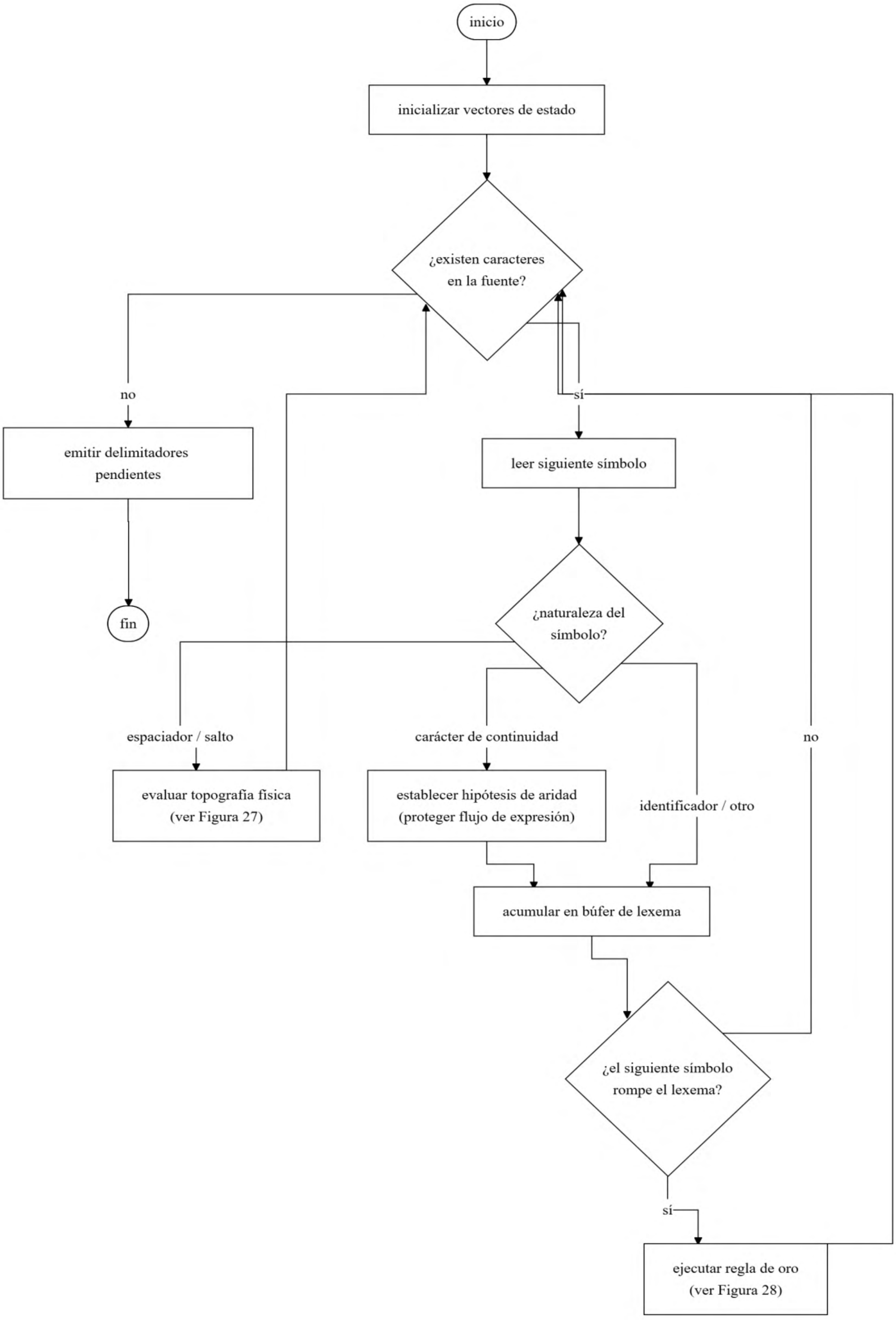
$\Phi_{n-1}$ el vector de indicadores volátiles del token anterior

$T'_n$ el token enriquecido resultante

Este enfoque permite que la topografía del código sea procesada como un flujo de información continua, donde el estado del sistema condiciona la interpretación de cada símbolo subsiguiente.

**Figura 26**

*Ciclo de control del* AEFT



### 4.6.2 La Regla de Oro: el paradigma de la inferencia de intención

El postulado fundamental que rige este sistema es la Regla de Oro. Se define como el conjunto de decisiones e inferencias heurísticas que el transductor ejecuta para acoplar la estructura gramatical a la intención del programador. Bajo este paradigma, la sintaxis es una propiedad emergente de la disposición física y la continuidad lógica. El sistema asume la responsabilidad de deducir la cohesión o ruptura de las expresiones, permitiendo que la forma física del código refleje la intención lógica sin intervención de delimitadores físicos redundantes.

Es imperativo destacar que la carga semántica de esta 'intención' recae exclusivamente en las fases de lexing y parsing. El transductor sintáctico actúa como un intérprete de voluntad, absorbiendo la complejidad y transformándola en instrucciones atómicas. Esta arquitectura garantiza que los módulos interiores, específicamente la máquina virtual, no se contaminen con lógica de inferencia, operando bajo un criterio de eficiencia pura.

Para materializar la Regla de Oro de forma determinista, el transductor emplea el vector de indicadores volátiles $\Phi$, el cual sintetiza la inercia del flujo. La relación de dependencia entre el token actual y el contexto se describe a través de los componentes de $\Phi_n$:

**A. Continuidad lógica ($C_o \in \{0,1\}$)**

Señala si el flujo gramatical quedó abierto por operadores o símbolos de continuidad. Donde $C_o = 1$ denota continuidad lógica.

**B. Aridad operativa ($U_n \in \{0,1\}$)**

Indica la aridad del operador. Donde $U_n = 1$ denota aridad unaria.

**C. Adyacencia física ($A_f \in \{0, d, LF\}$)**

Mide y clasifica la separación topográfica. Donde 0 es contacto físico, $d$ representa una separación horizontal (espacios o tabuladores) y $LF$ una separación vertical (salto de línea).

### 4.6.3 Semántica adaptativa

La efectividad de la Regla de Oro reside en su capacidad para anticipar o reinterpretar los resultados de la inferencia de continuidad basándose en patrones comunes del pensamiento lógico.

#### 4.6.3.1 Principios de adaptabilidad aplicados a la inferencia de continuidad

El sistema utiliza una serie de principios adaptables y extensibles descritos a continuación:

### A. Inferencia de aridad operativa

En los sistemas tradicionales, la ambigüedad de un operador se resuelve tras su captura completa. En DinoCode, se emplea un mecanismo de inferencia prematura. Al detectar el primer carácter del símbolo, se evalúa el siguiente predicado:

$$U_n = C_o \vee (L = 0)$$

Donde:

$L$ es la longitud total de la colección de tokens

Mientras se consumen caracteres para identificar el operador completo, el estado $U_n = 0$ garantiza que la continuidad del flujo se mantenga protegida hasta que el token sea insertado.

Algo a destacar, es que este modelo permite definir la aridad de los operadores unarios y binarios desde la fase de análisis léxico, ya que $U_n$ es la hipótesis de aridad que es validada y adjuntada como *metadato* una vez se capture todo el operador.

### B. Cohesión por contacto

El contacto físico ($A_f = 0$) actúa como una fuerza de unificación de tipos. Un ejemplo crítico son los símbolos de apertura (paréntesis, llaves y corchetes). Estos lexemas, pueden representar agrupación, acceso o instanciación, lo que los vuelve ambiguos. Para solucionarlo, se aplica una regla de cohesión inmediata que afecta la continuidad operativa antes de su procesamiento sintáctico. Sea $L$ un símbolo de apertura, su impacto en el vector de estado se define como:

$$\Phi_n(C_o) = \begin{cases} 1 & \text{si } A_f = 0 \\ \Phi_{n-1}(C_o) & \text{si } A_f \in \{d, LF\} \end{cases}$$

Esta fórmula describe cómo el contacto directo fuerza la continuidad, mientras que la separación física mantiene el estado previo, permitiendo que la interpretación dependa de si el flujo ya era continuo o si se encontraba en una ruptura lógica.

## C. Supresión por redundancia

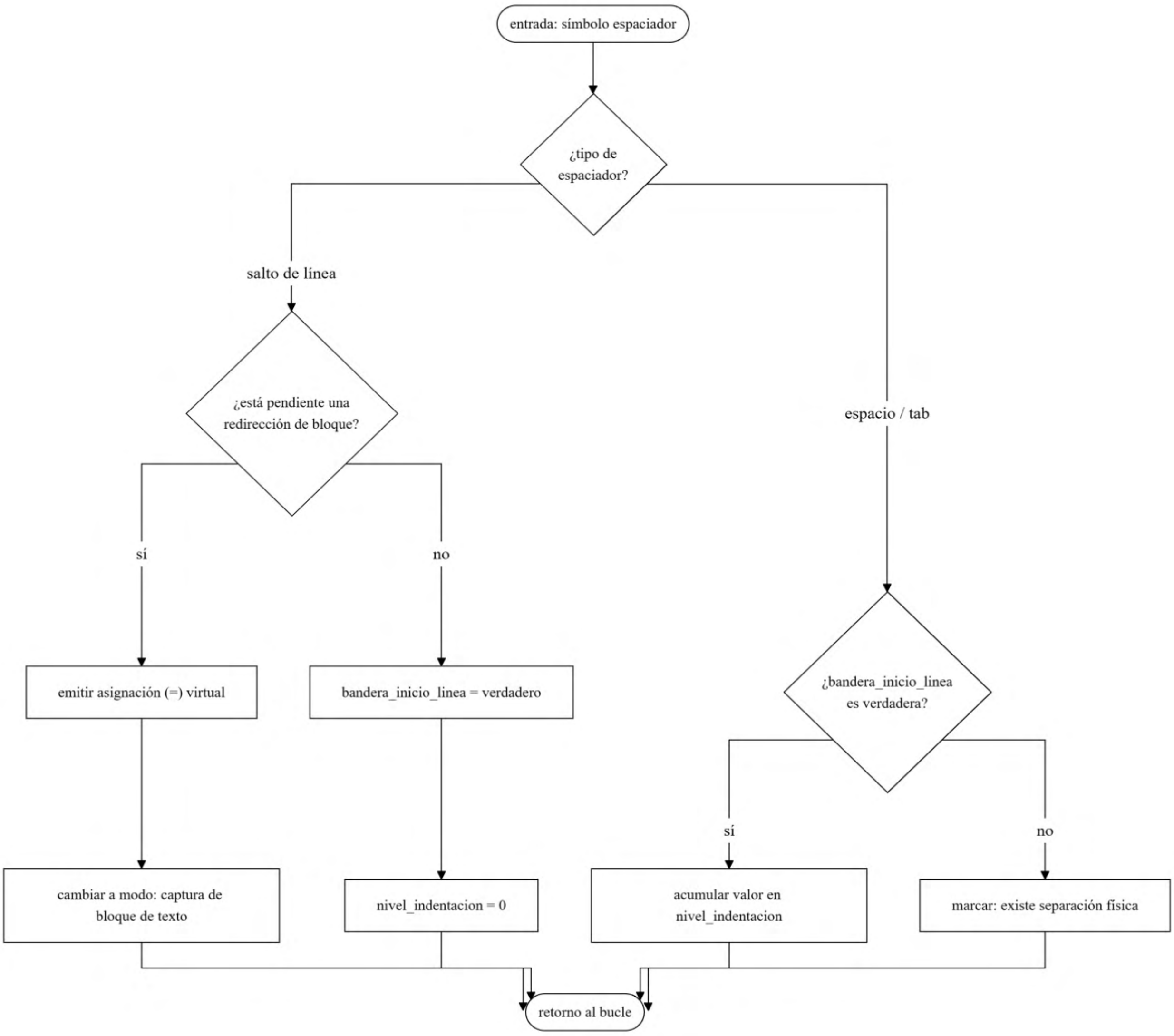
El AEFT actúa como un motor de inyección de delimitadores virtuales ($\delta_v$). La emisión de un delimitador virtual es una función de la ruptura de continuidad frente a la topografía actual. Un principio rector es la inhibición deliberada en símbolos de clausura:

$$\text{emitir } \delta_v \Leftrightarrow \neg(C_o \vee \neg U_n \vee T_n \in \{\text{clausuras}\})$$

Si el token entrante es un delimitador de cierre y no existe continuidad operativa, el sistema suprime la emisión de $\delta_v$. Al asumir que el propio símbolo de clausura actúa como frontera topográfica, se garantiza que la representación sea semánticamente densa y estructuralmente limpia.

**Figura 27**

*Algoritmo de transición para estados topográficos y redirección*



#### 4.6.3.2 Reglas de emisión de delimitadores virtuales ($\delta_v$)

Sea:

$D$ la profundidad de anidamiento por delimitadores de agrupación explícitos

$\delta_v$ el delimitador virtual resultante

Entonces $\delta_v \in \{, ; \}$ sí y solo si:

1. **Separación de expresión**

$$\delta_v \to ',' \Leftrightarrow \neg C_o \wedge (D > 0 \vee A_f \in \{d\}).$$

Cuando la continuidad se rompe ($\neg C_o$) pero hay una agrupación pendiente ($D > 0$) o se detecta una separación horizontal ($A_f \in \{d\}$), el sistema inserta una separación de expresión virtual.

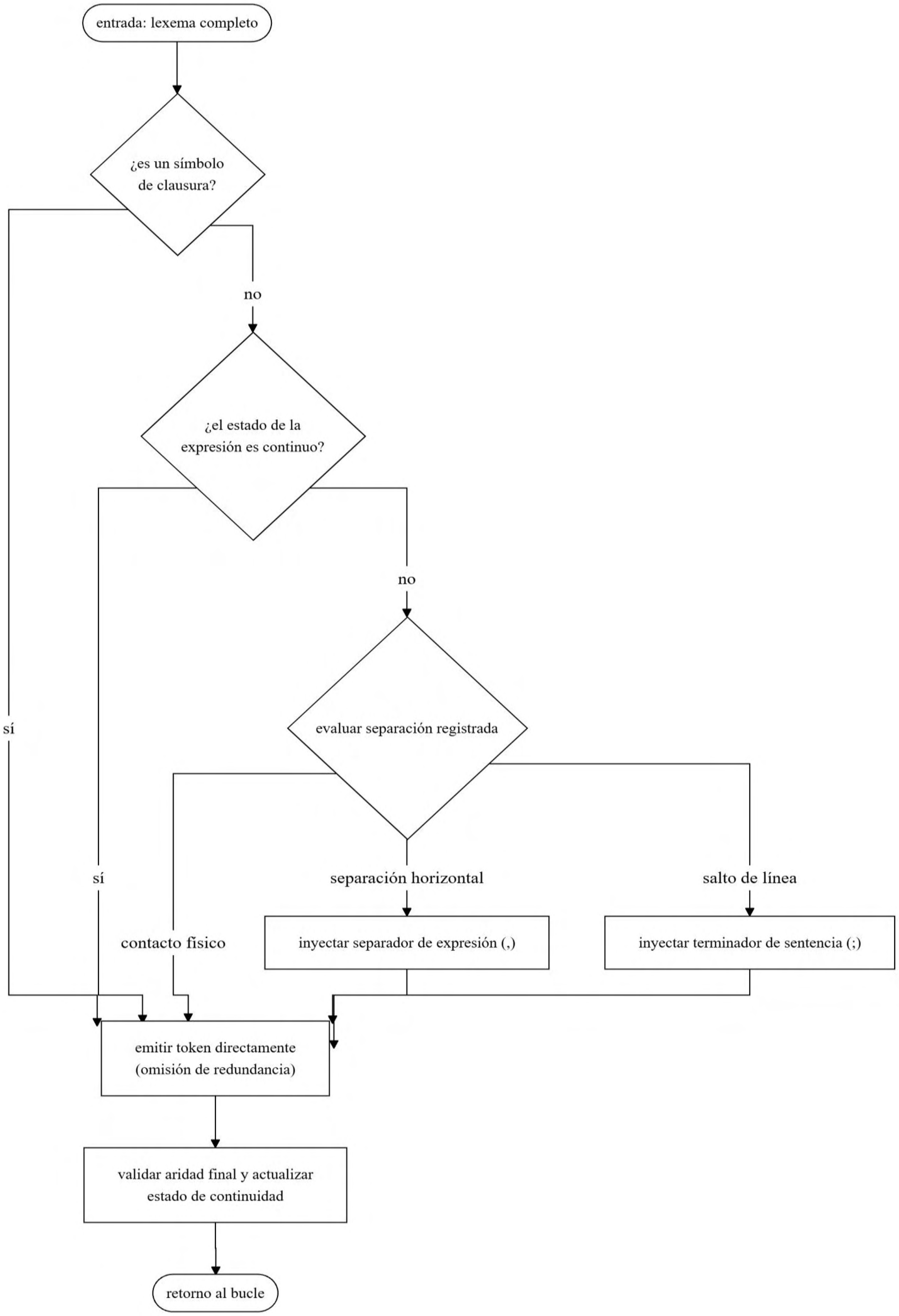
2. **Terminador de sentencia**

$$\delta_v \to ';' \Leftrightarrow \neg C_o \wedge D = 0 \wedge A_f \in \{LF\}.$$

Cuando la continuidad se rompe ($\neg C_o$) y coincide con un salto de línea ($A_f \in \{LF\}$) fuera de cualquier agrupación ($D = 0$), el sistema emite un final de sentencia.

**Figura 28**

*Algoritmo de inferencia y omisión de la regla de oro.*



### 4.6.3.3 Mecanismos de rectificación y llamados implícitos a funciones

El motor de DinoCode reinterpreta los resultados de la inferencia de continuidad basándose en patrones comunes del pensamiento del programador:

#### A. El postulado de la invocación implícita

Cuando un identificador es seguido por una separación física, está fuera de una agrupación y actualmente no tiene continuidad ($\neg C_o \wedge D = 0 \ \wedge A_f \in \{d, LF\}$), el sistema asume una intención de invocación. Científicamente, esto genera una ruptura de continuidad artificial, donde se fuerza el estado $C_o = 0$ aunque exista un operador subsiguiente. Esta transición habilita que el siguiente token sea interpretado como un argumento unario. Por ejemplo:

**Figura 29**

*Inferencia de intención de invocación*

```
identificador  - 1
```

*Nota*. El sistema infiere que -1 es un valor negativo y no una operación de resta sobre el identificador.

#### B. Rectificación por superposición

Existen escenarios donde la regla espacial entraría en conflicto con el modelo mental del programador. Para resolverlo, ciertos tokens poseen la capacidad de sobrescribir el estado previo mediante una rectificación de la ruptura:

- **Operadores especiales**

  Si el token subsiguiente es un operador de asignación o acceso jerárquico (propiedades o métodos), el sistema anula la ruptura de la invocación implícita y restaura la continuidad operativa $\Phi_{n-1}(C_o)$. Por ejemplo:

**Figura 30**

*Anulación de la invocación implícita por operadores especiales*

```
identificador  =       expresión
```

*Nota*. El sistema infiere que la intención es la asignación y no el llamado implícito.

- **Operadores de ruptura (*Breakers*)**

  Los operadores postfijos fuerzan un estado $C_o = 0$. Este mecanismo permite que expresiones independientes coexistan en una misma línea sin necesidad de comas físicas, delegando la separación a la ruptura de continuidad detectada por el motor.

**Figura 31**

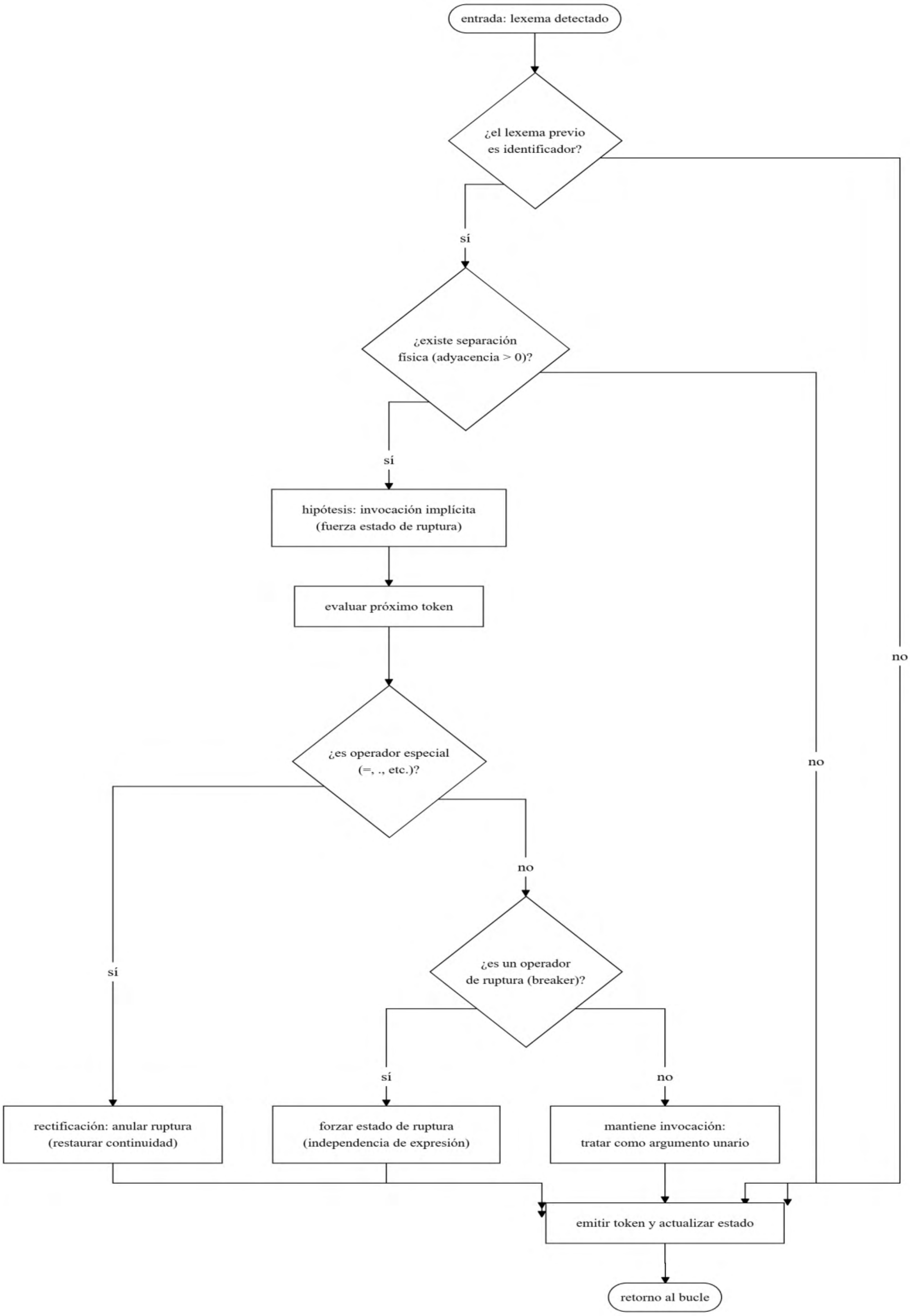
*Ruptura de continuidad operativa por Breakers*

```
i++              j--
```

*Nota*. El sistema infiere que son dos expresiones independientes, a pesar de que existe un operador intermedio y están en la misma línea.

**Figura 32**

*Algoritmo de rectificación semántica y arbitraje de invocación implícita.*



## 4.7. Diseño del transductor sintáctico

Aunque el término “analizador sintáctico” es común, la implementación de DinoCode se define con mayor precisión como un transductor sintáctico. Esto se debe a que el componente no se limita a validar la estructura de la entrada, sino que realiza una transformación activa y concurrente: convierte una secuencia de símbolos gramaticales en un flujo de instrucciones operativas (bytecode) sin estados intermedios. En este sentido, el parser actúa como un transductor de alto nivel donde cada reducción de una regla gramatical produce una emisión inmediata hacia la máquina virtual.

### 4.7.1 Principios fundamentales del parser

El transductor sintáctico de DinoCode se fundamenta en tres principios de diseño que definen su comportamiento:

- **Principio de separación de contextos**

  El parser mantiene una clara distinción entre el contexto sintáctico actual (función, clase, condicional) y los mecanismos de construcción activos, permitiendo la gestión modular de las estructuras anidadas.

- **Principio de precedencia explícita**

  Cada operador tiene definida su precedencia y asociatividad mediante reglas formales que determinan el orden exacto de evaluación, incluyendo casos especiales como operadores unarios y asignación.

- **Principio de construcción incremental**

  Las estructuras sintácticas se construyen incrementalmente mediante marcos de ejecución que permiten la recursividad y el anidamiento sin pérdida del contexto original.

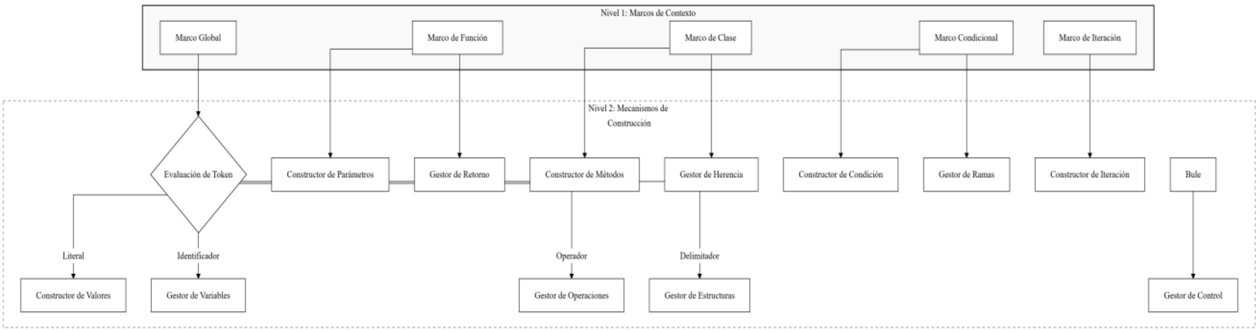
### 4.7.2 Contextos y marcos de ejecución

Antes de detallar el flujo de procesamiento, es imperativo establecer la distinción jerárquica que gobierna el componente parser. El diseño del transductor sintáctico se fundamenta en una estructura que separa la interpretación del contexto sintáctico de la gestión de marcos de construcción. Como se ilustra en la Figura 33, el sistema opera en dos niveles de abstracción: los marcos de contexto y los mecanismos de construcción.

El marco de contexto actúa como un estado de control superior que define la estructura sintáctica activa en un momento dado (por ejemplo, si el sistema está construyendo una función, un condicional o un objeto). Por su parte, los mecanismos de construcción funcionan como acumuladores transitorios que agrupan las instrucciones según su naturaleza estructural mientras se permanece en un marco determinado. Esta separación garantiza que una misma construcción pueda ser interpretada de formas distintas dependiendo del contexto sintáctico en el que se encuentre.

**Figura 33**

*Contexto y marcos de construcción.*

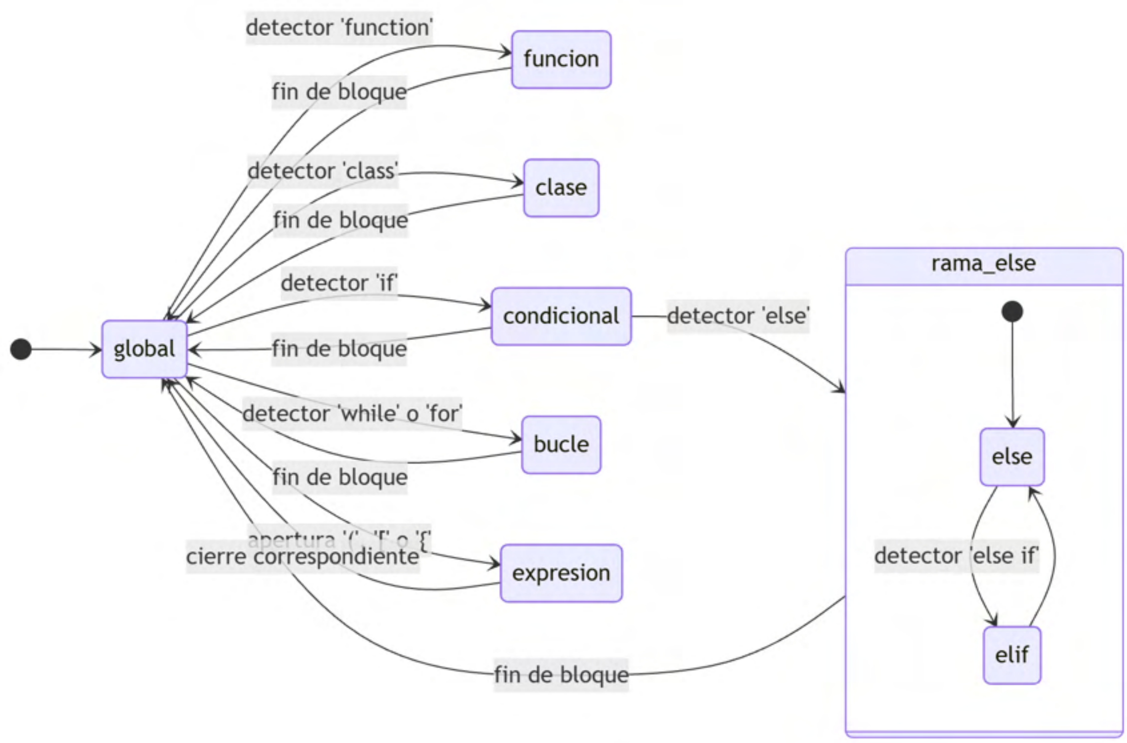


### 4.7.3 Dinámica de estados y recursividad estructural

La transición entre los diferentes contextos no es lineal, sino que responde a un modelo de pila que permite la recursividad. La Figura 34 describe el autómata de estados finitos que regula el flujo del parser. Mediante operaciones de apilado y desapilado de marcos, el sistema puede procesar estructuras complejas, como funciones anidadas dentro de clases o condicionales anidados dentro de bucles, garantizando que el analizador recupere con exactitud el estado previo una vez resuelta la sub-estructura.

**Figura 34**

*Autómata de estados finitos para la gestión de contextos recursivos.*



### 4.7.4 Algoritmo de procesamiento principal

El núcleo operativo del parser reside en el procesamiento secuencial de tokens mediante un algoritmo determinista. La Figura 35 y Figura 36 detalla el flujo completo desde la entrada hasta la generación de bytecode.

**Figura 35**

*Ciclo principal y clasificación inicial de tokens*

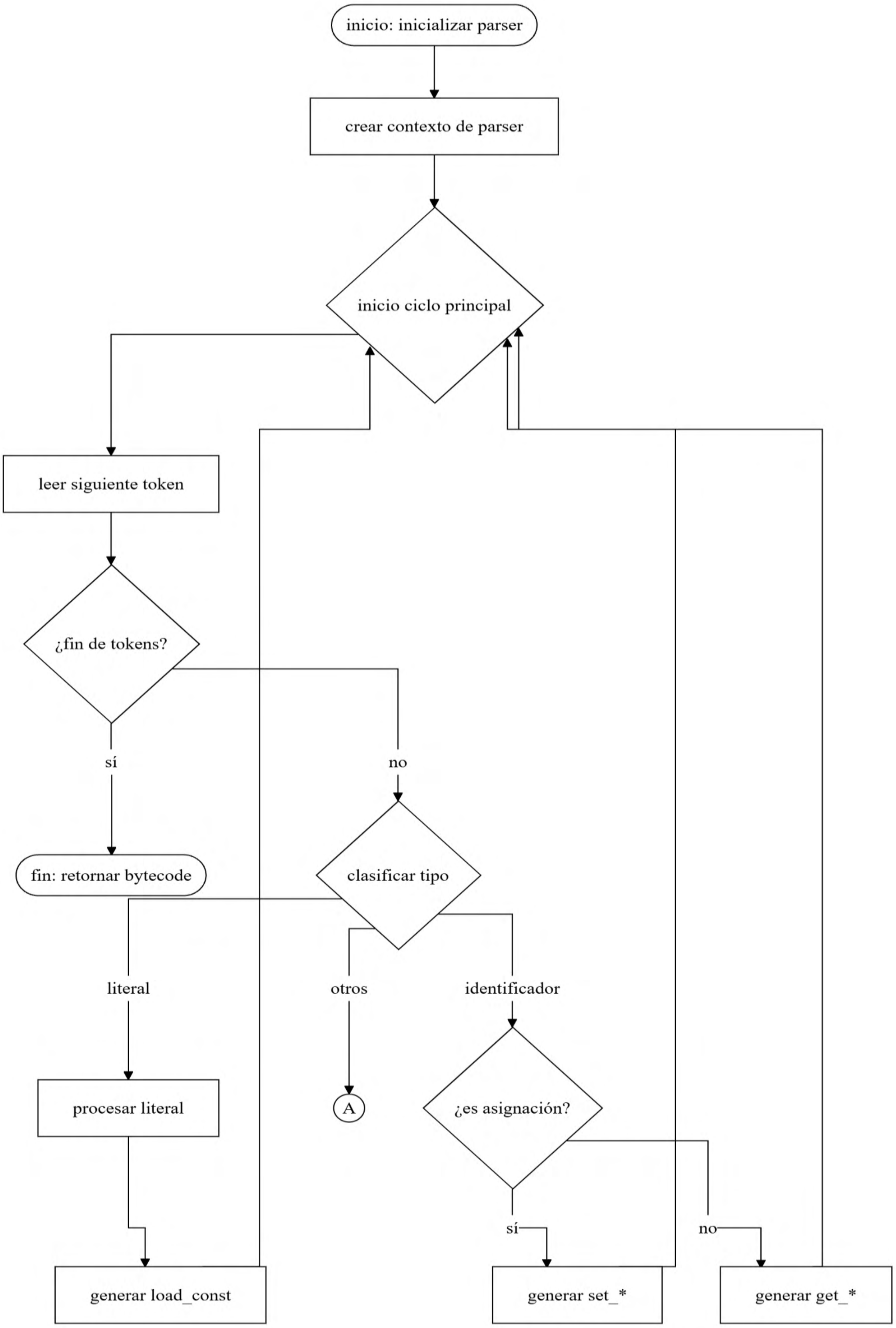
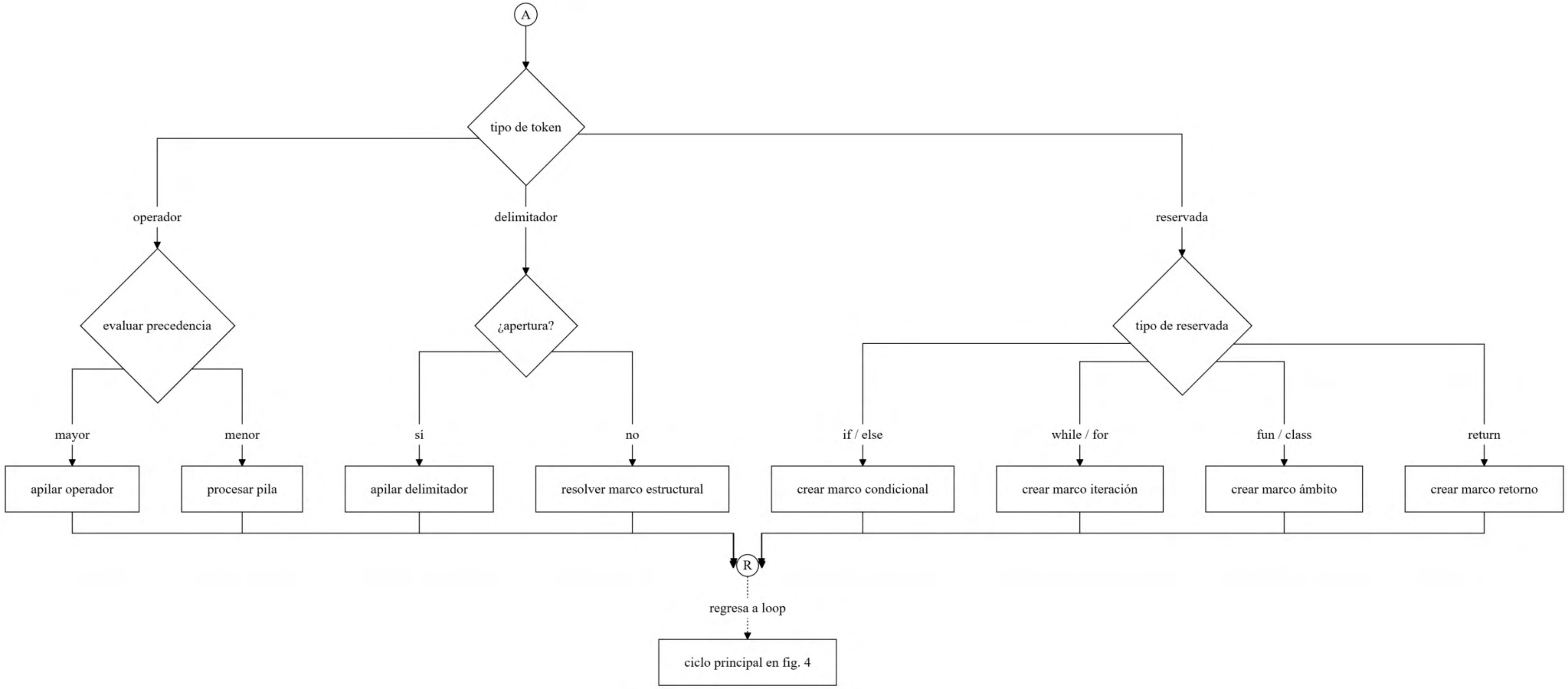


**Figura 36**

*Procesamiento de operadores, delimitadores y control*

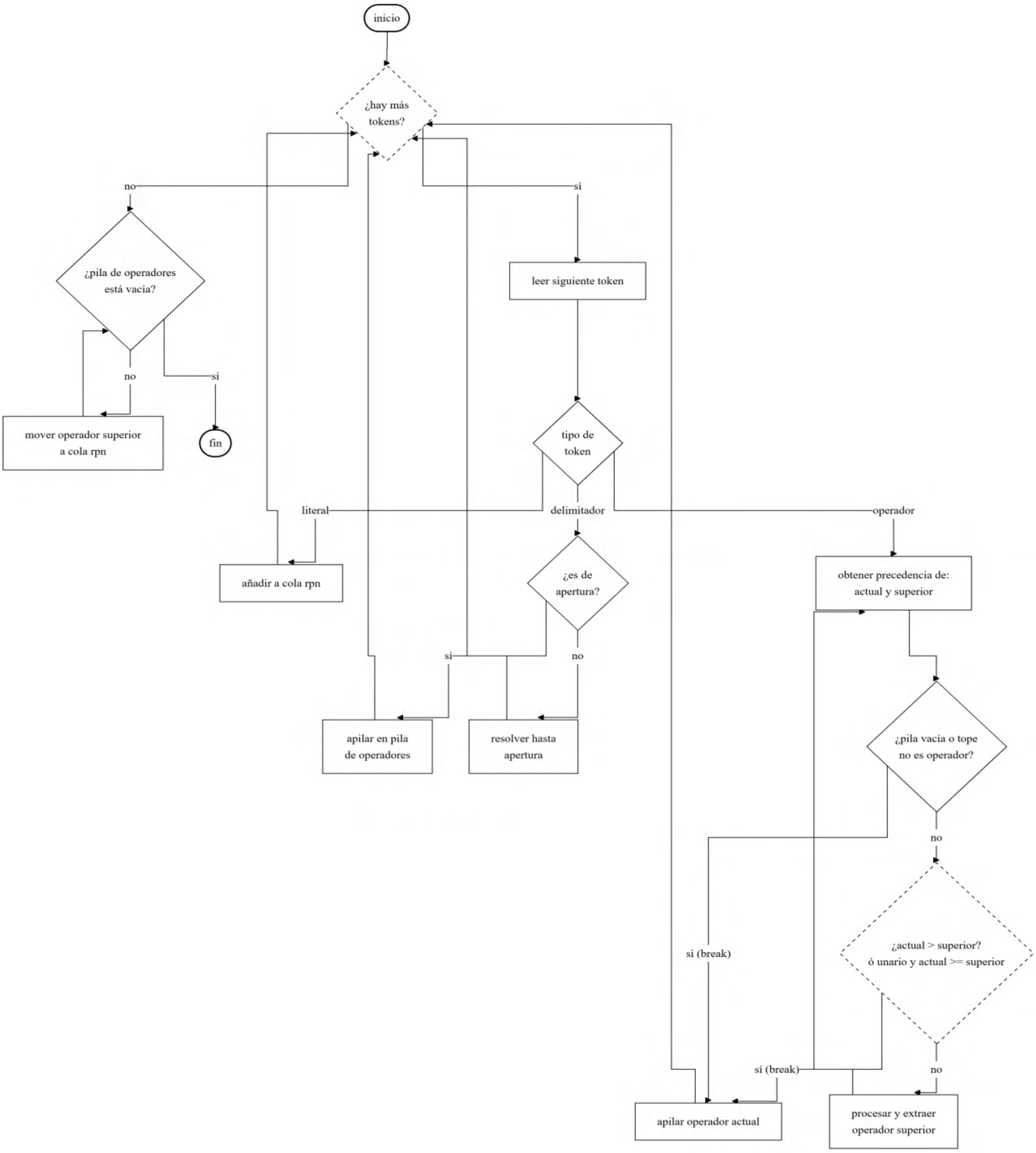


### 4.7.5 Algoritmo de resolución de precedencia

La Figura 37 muestra el algoritmo específico para la resolución de precedencia de operadores, un componente que garantiza el orden correcto de evaluación de expresiones complejas.

**Figura 37**

*Algoritmo de resolución de precedencia de operadores*



### 4.7.6 Gestión de ámbitos y punteros de pila

El parser implementa un sistema de gestión de ámbitos que permite el anidamiento de variables y funciones mediante punteros.

#### 4.7.6.1 Sistema de mapeo de ámbitos

Mantiene un sistema dual de mapeo que permite una gestión optimizada de variables en diferentes niveles de ámbito:

- **Mapeo global a local**: Cada variable tiene un índice global único que permite la deduplicación de nombres
- **Mapeo de ámbito actual**: Cada ámbito mantiene un vector de índices globales que están visibles en ese nivel

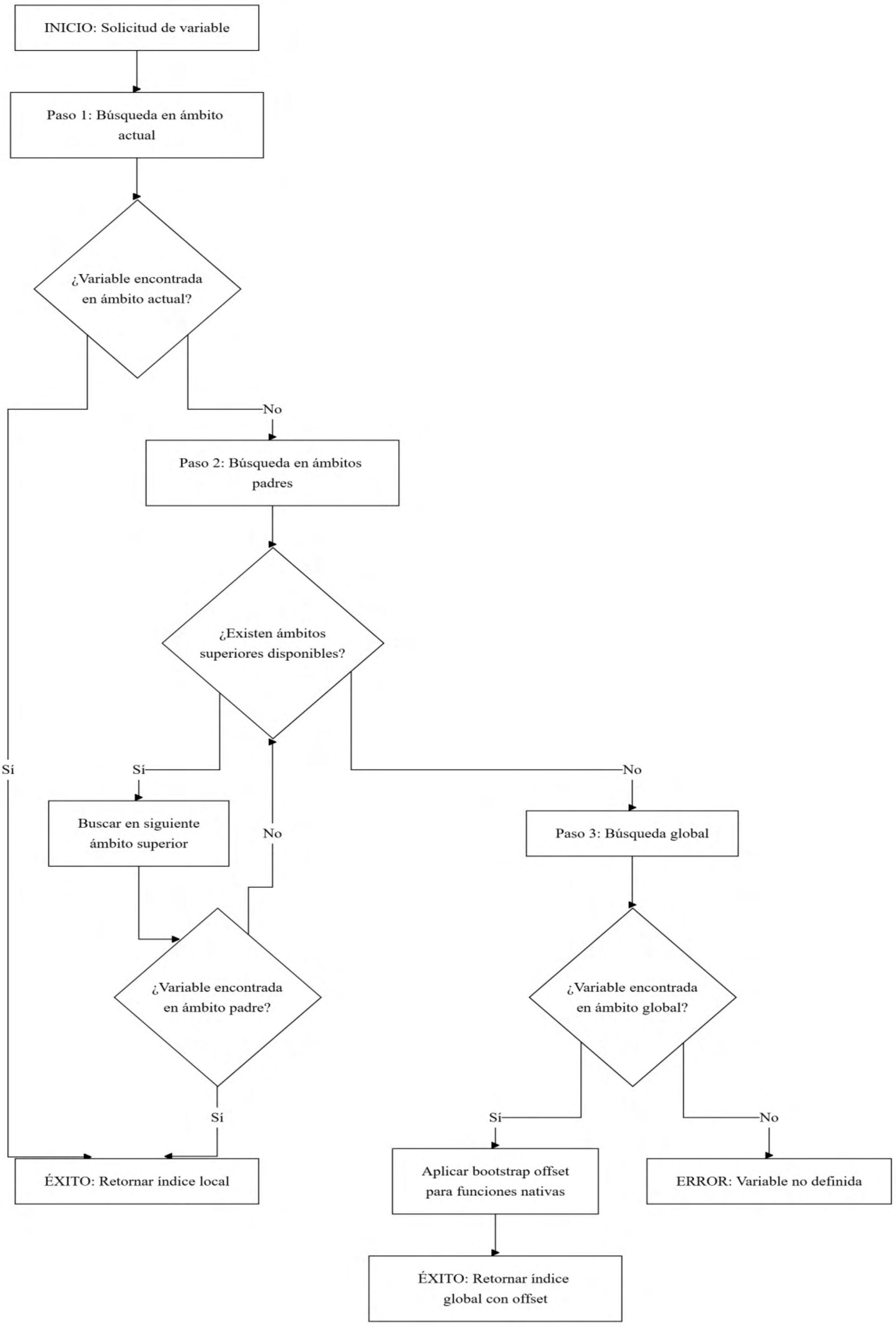
**Tabla 28**

*Estructura de mapeo de ámbitos en ValuePool*

| Componente | Tipo de estructura | Descripción | Propósito |
|---|---|---|---|
| Nombres globales | Vector de cadenas | Registro único de nombres de variables | Deduplicación de identificadores |
| Mapeo nombre-índice | Tabla hash | Asociación nombre → índice global | Búsqueda por nombre |
| Pila de ámbitos | Vector de vectores | Jerarquía de ámbitos anidados | Gestión de visibilidad de variables |
| Mapeo local-global | Tabla hash por ámbito | Asociación índice local → global | Resolución rápida en el ámbito actual |

**Figura 38**

*Algoritmo de resolución de ámbitos*



#### 4.7.6.2 Gestión de variables temporales

Para operaciones complejas como el bucle `for`, el sistema asigna variables temporales con nombres únicos `(@temp_0, @temp_1, @temp_2, ...)`

**Tabla 29**

*Características de variables temporales*

| Característica | Mecanismo de implementación | Objetivo de diseño |
|---|---|---|
| Identificación única | Prefijo especial y un contador | Evitar conflictos con identificadores de usuario |
| Gestión automática | Sistema de asignación especializado | Garantizar la creación y registro consistentes |
| Liberación garantizada | Limpieza por ámbito | Prevenir fugas de memoria |

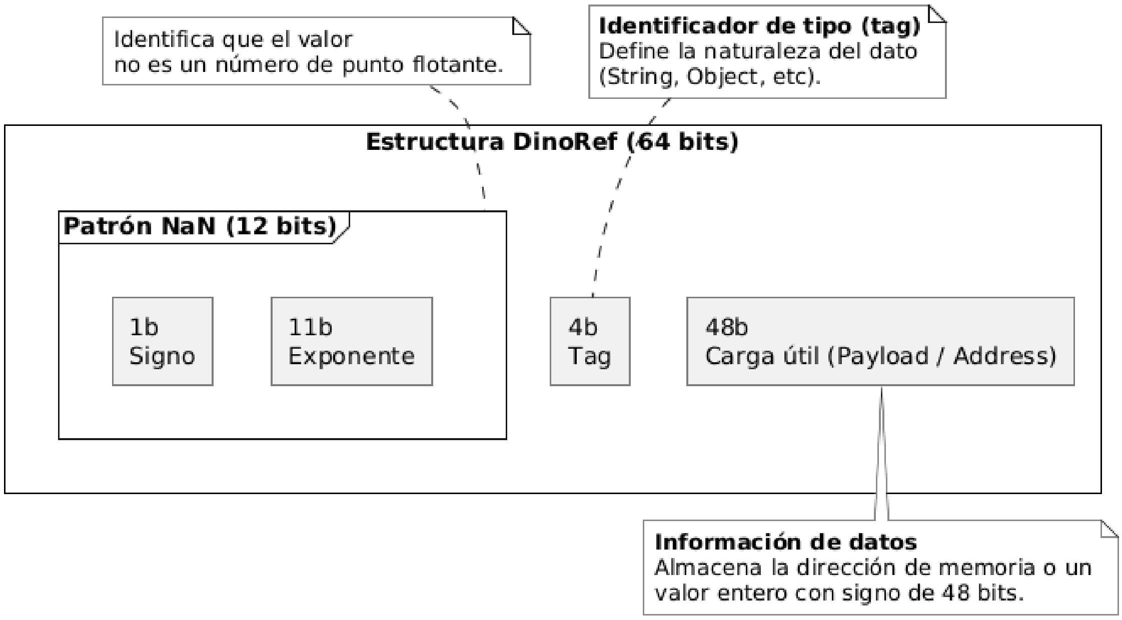
## 4.8. Especificación técnica

### 4.8.1 Sistema de representación de valores: DinoRef

En DinoCode, un DinoRef emplea un esquema de NaN boxing basado en rangos. Se establece un umbral en **`0xFFF0000000000000`**, cualquier patrón de bits inferior a este valor se interpreta como un número de punto flotante nativo. Los valores superiores se segmentan en sub-rangos que identifican tipos complejos (como cadenas, arreglos u objectos) mediante etiquetas de tipo (tags), reservando el rango superior (**`0xFFF8...`**) para la representación de números enteros de 48 bits. Esta arquitectura permite un despacho de tipos mediante comparaciones de magnitud, optimizando el rendimiento del intérprete al reducir los saltos condicionales.

**Figura 39**

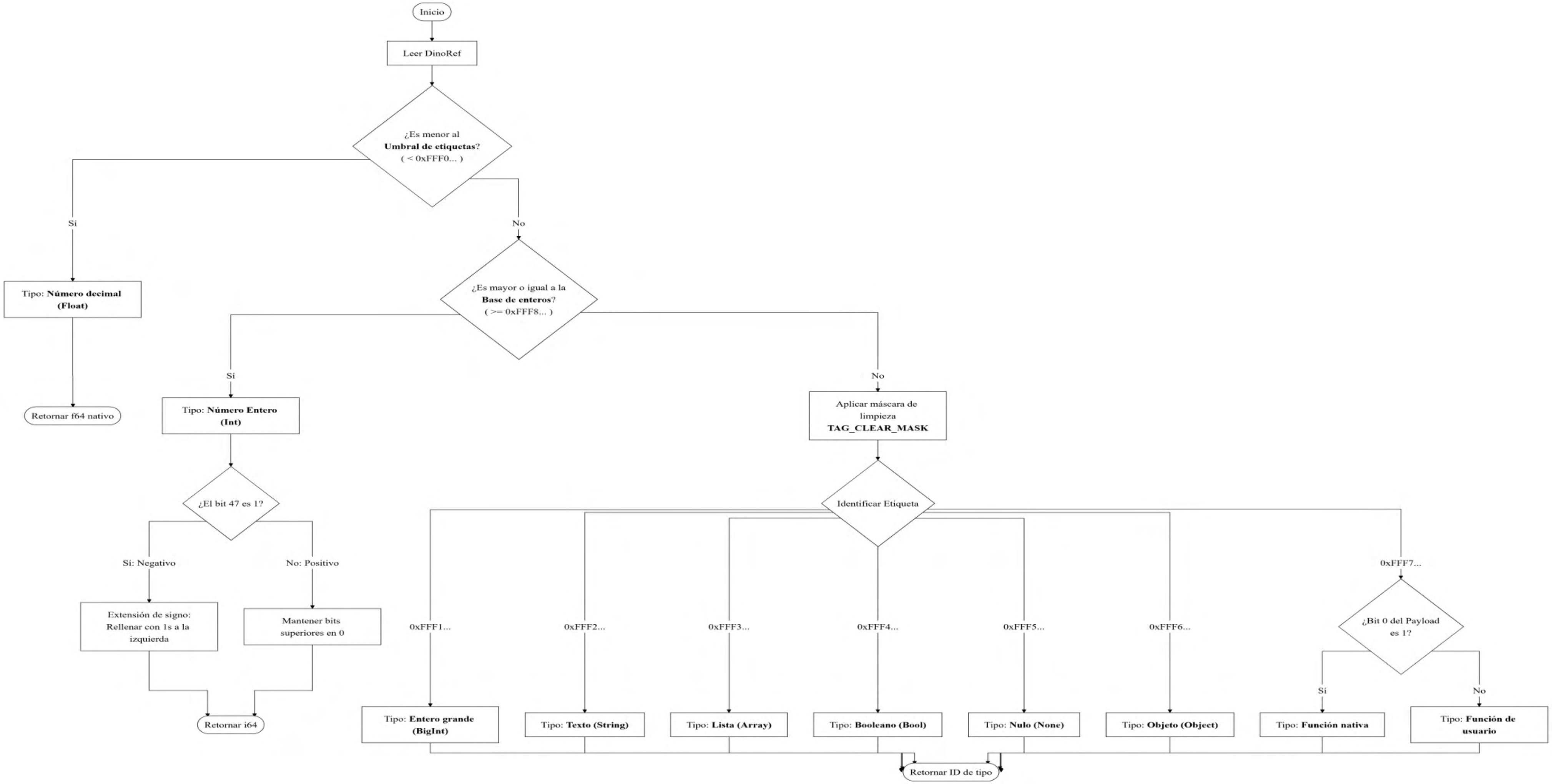
*Diagrama de la segmentación de bits para la estructura DinoRef*



La Figura 40 expone el flujo de reconocimiento de tipos usando un DinoRef estándar:

**Figura 40**

*Diagrama de flujo para la clasificación de datos con DinoRef*



A continuación, se detalla el uso de cada espacio de bits:

**Tabla 30**

*Especificación completa del sistema NaN Boxing en DinoCode*

| Tipo | Tag (Hex) | Rango (Hex) | Payload | Bits | Descripción |
|---|---|---|---|---|---|
| Float | N/A | 0x0000_0000_0000_0000<br>0xFFEF_FFFF_FFFF_FFFF | IEEE 754 | 64 | Flotantes de doble precisión |
| BigInt | 0xFFF1 | 0xFFF1_0000_0000_0000<br>0xFFF1_FFFF_FFFF_FFFF | Offset | 32 | Offset a arena de big integers |
| String | 0xFFF2 | 0xFFF2_0000_0000_0000<br>0xFFF2_FFFF_FFFF_FFFF | Offset | 32 | Offset a arena de strings |
| Array | 0xFFF3 | 0xFFF3_0000_0000_0000<br>0xFFF3_FFFF_FFFF_FFFF | Offset | 32 | Offset al pool de arrays |
| Bool | 0xFFF4 | 0xFFF4_0000_0000_0000<br>0xFFF4_0000_0000_0001 | Value | 1 | 0 = false<br>1 = true |
| None | 0xFFF5 | 0xFFF5_0000_0000_0000 | N/A | 0 | Valor nulo |
| Object | 0xFFF6 | 0xFFF6_0000_0000_0000<br>0xFFF6_FFFF_FFFF_FFF8 | Offset | 31 | Offset al pool de objetos |
| Class | 0xFFF6 | 0xFFF6_0000_0000_0001<br>0xFFF6_FFFF_FFFF_FFFF | Offset | 31 | Object + bit de clase |
| Function | 0xFFF7 | 0xFFF7_0000_0000_0000<br>0xFFF7_FFFF_FFFF_FFF8 | ID | 31 | ID de función de usuario |
| Native | 0xFFF7 | 0xFFF7_0000_0000_0001<br>0xFFF7_FFFF_FFFF_FFFF | ID | 31 | Function + bit de tipo nativa |
| Integer | 0xFFF8 | 0xFFF8_0000_0000_0000<br>0xFFFF_FFFF_FFFF_FFFF | Value | 48 | Enteros de 48 bits con signo |

### 4.8.2 Gestión de memoria híbrida

DinoCode emplea una estrategia de gestión de memoria diferenciada según la mutabilidad semántica de los datos: la arena inmutable para datos que no cambian después de su creación (cadenas de texto, big integers) y el pool de objetos mutables para datos que pueden modificarse en tiempo de ejecución (arreglos, diccionarios). Esta separación permite aplicar estrategias de recolección de basura distintas y optimizadas para cada clase de dato.

#### 4.8.2.1 Arena de datos inmutables

La arena es un vector de bytes (`Vec<u8>`) que crece únicamente por adición al final. Toda asignación consiste en reservar `n` bytes al final del vector y retornar el desplazamiento donde comienza la nueva región.

**Tabla 31**

*Distribución de bytes para cadenas de texto*

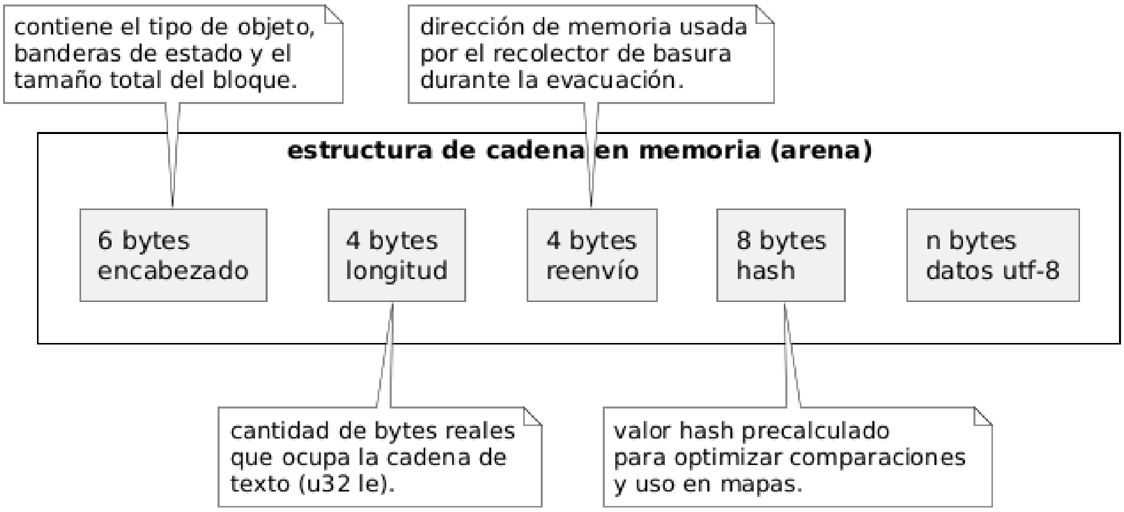
| Bytes | Contenido |
|---|---|
| [0..6) | Cabecera:<br>tipo (1 byte)<br>banderas - *flags* (1 byte)<br>tamaño de datos (4 bytes) |
| [6..10) | Longitud de la cadena en bytes (4 bytes) |
| [10..14) | Puntero de reenvío (*forwarding pointer*), usado durante la compactación GC (4 bytes) |
| [14..22) | Hash de la cadena (8 bytes, para interning) |
| [22..22+n) | Bytes UTF-8 de la cadena |

**Interning de cadenas constantes**

Las cadenas que aparecen en el código fuente pasan por el internador de cadenas. El internador calcula el hash de la cadena y consulta una tabla hash interna antes de asignar; si la cadena ya existe en la arena, retorna el desplazamiento existente sin asignarla nuevamente. Esto garantiza que dos literales de cadena idénticos compartan una única representación en memoria, reduciendo el consumo de la arena y acelerando las comparaciones por igualdad de referencia.

**Figura 41**

*Distribución del encabezado de cadena en la arena*



#### 4.8.2.2 Pool de objetos mutables

El pool de objetos gestiona las ranuras de almacenamiento para arreglos y objetos. El pool es un bloque de memoria contiguo de $N$ ranuras. Para un pool con capacidad inicial de $N$ ranuras, el bloque se asigna una sola vez al inicio; el crecimiento se realiza mediante `realloc` con duplicación de capacidad cuando el pool se satura.

El pool mantiene dos bitmaps principales:

- **Mapa de bits:** Indica qué ranuras están actualmente asignadas. El bit es 1 si la ranura está ocupada.
- **Mapa de marcado:** Utilizado durante la fase de marcado del recolector de basura. El bit es 1 si el objeto en la ranura fue alcanzado durante el trazado del recolector.

Cada bitmap almacena sus bits en un vector de DinoRef (`Vec<u64>`). Para un pool de $N$ ranuras, el número de DinoRef necesarias es $\lceil N/64 \rceil$. Las operaciones de lectura/escritura de un bit en el índice $k$ se realizan mediante:

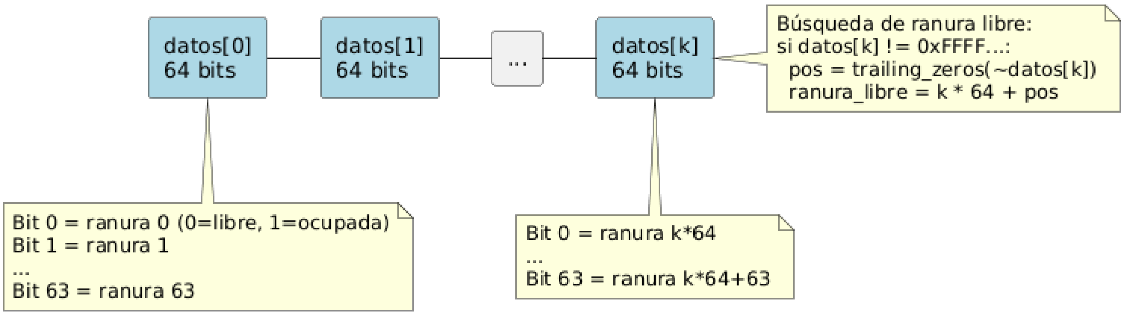
$$\text{DinoRef} = k \div 64$$

$$\text{bit\_dentro\_del\_ref} = k \bmod 64$$

En lugar de una inspección lineal bit a bit, el sistema optimiza la búsqueda de espacio libre evaluando DinoRef completas. Si una referencia presenta un valor distinto al máximo representable ($2^{64} - 1$), se identifica la existencia de al menos una ranura vacía. La posición del primer bit disponible se obtiene mediante el conteo de ceros a la derecha sobre el complemento de la referencia. Esta técnica aprovecha las instrucciones de conteo de ceros del hardware, lo que reduce la complejidad de asignación de $O(N)$ a $O(N/64)$ en el peor de los casos.

**Figura 42**

*Estructura del bitmap de 64 bits en el pool de objetos*



#### 4.8.2.3 Ciclo de recolección de basura

El recolector de basura de DinoCode es un híbrido que aplica estrategias diferenciadas a cada región de memoria. El punto de entrada es la fase de marcado y barrido, que se ejecuta cuando los contadores de presión de la arena o del pool superan sus umbrales configurados.

El ciclo se divide en tres fases secuenciales:

**Figura 43**

*Fases iniciales del ciclo de recolección de basura: detección y marcado*

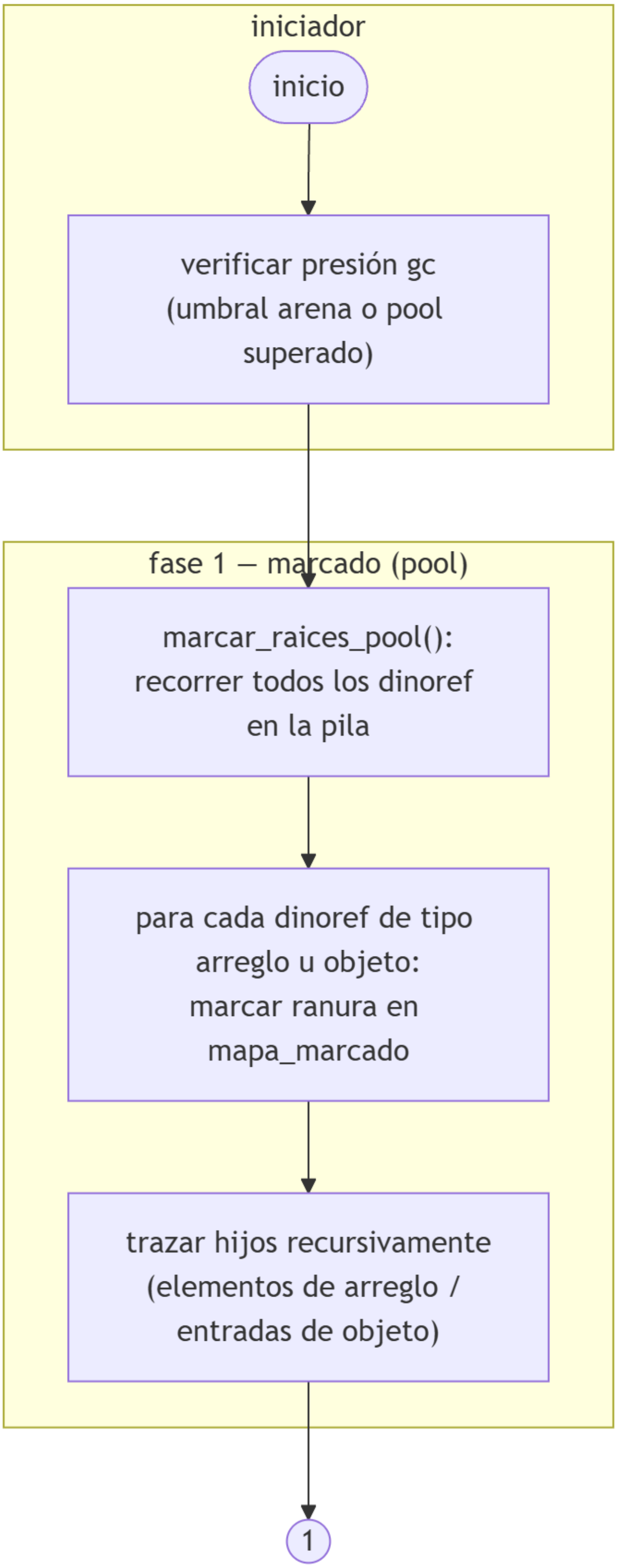
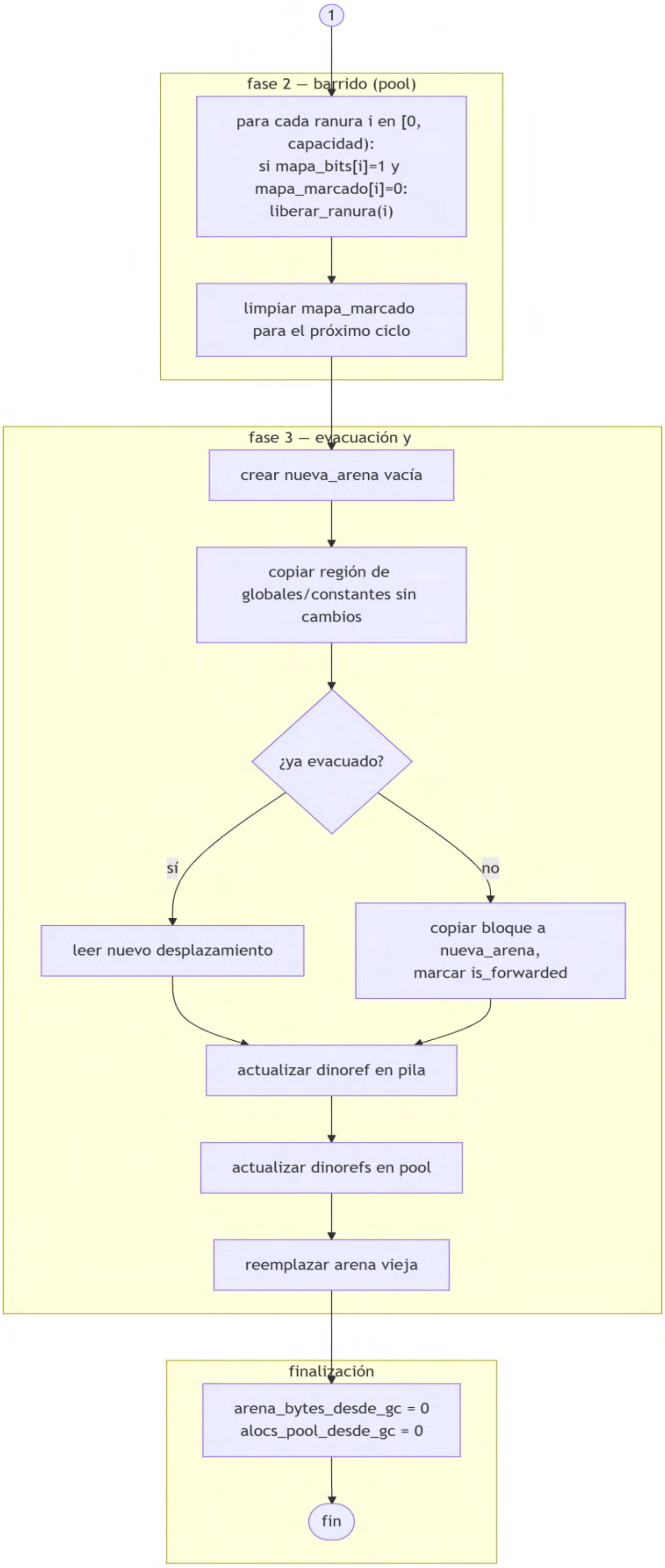


**Figura 44**

*Fases finales del ciclo de recolección de basura: barrido y compactación de arena.*



Respecto a la Figura 43 y Figura 44, la evacuación de la arena implementa una variante del algoritmo de copiado (copying collector), en la cual el espacio de destino (nueva arena) se construye de forma incremental durante el recorrido de las raíces del sistema. En este proceso, el empleo de un puntero de reenvío (*forwarding pointer*) resulta fundamental para garantizar la consistencia de los datos; si una misma cadena es referenciada desde múltiples posiciones en la pila, este mecanismo asegura que el objeto se desplace y copie una sola vez hacia el nuevo segmento de memoria (Pfenning, 2024).

La detección de cadenas ya evacuadas se realiza mediante la inspección del bit de flag del encabezado de la cadena en la arena antigua. Una vez confirmada la evacuación, el nuevo desplazamiento del dato se recupera también de su cabecera antigua (bits de forwarding pointer), permitiendo la actualización de todas las referencias huérfanas.

Respecto al rendimiento del algoritmo, la complejidad temporal del ciclo completo se define mediante la expresión:

$$O(R + P + A)$$

Donde:

$R$ representa el número de entradas en la pila de datos

$P$ el número de ranuras del pool

$A$ la extensión total en bytes de la arena.

Por otro lado, la complejidad espacial se establece como $O(A)$, debido a la necesidad de reservar un espacio equivalente para la nueva arena durante la fase de compactación de los datos supervivientes.

### 4.8.3 Sistema de interoperabilidad nativa

#### 4.8.3.1 Registro de funciones mediante macros procedurales

La interoperabilidad entre el código nativo escrito en Rust y el ambiente de ejecución de DinoCode se implementa mediante un sistema de macros procedurales de atributo. Este mecanismo elimina el costo de escritura de código de adaptación manual y garantiza que el registro de funciones sea verificado en tiempo de compilación.

El componente dinocode-macros expone cuatro macros principales:

- **#[dinof]**: Registra una función de Rust como función nativa de DinoCode.

- **#[dinoclass]**: Registra una estructura de Rust como prototipo de clase DinoCode.
- **#[dinomethods]**: Registra automáticamente todos los métodos de un bloque de implementación de Rust como métodos de la clase.

**Figura 45**

*Ejemplo del registro de una función nativa de Rust en DinoCode*

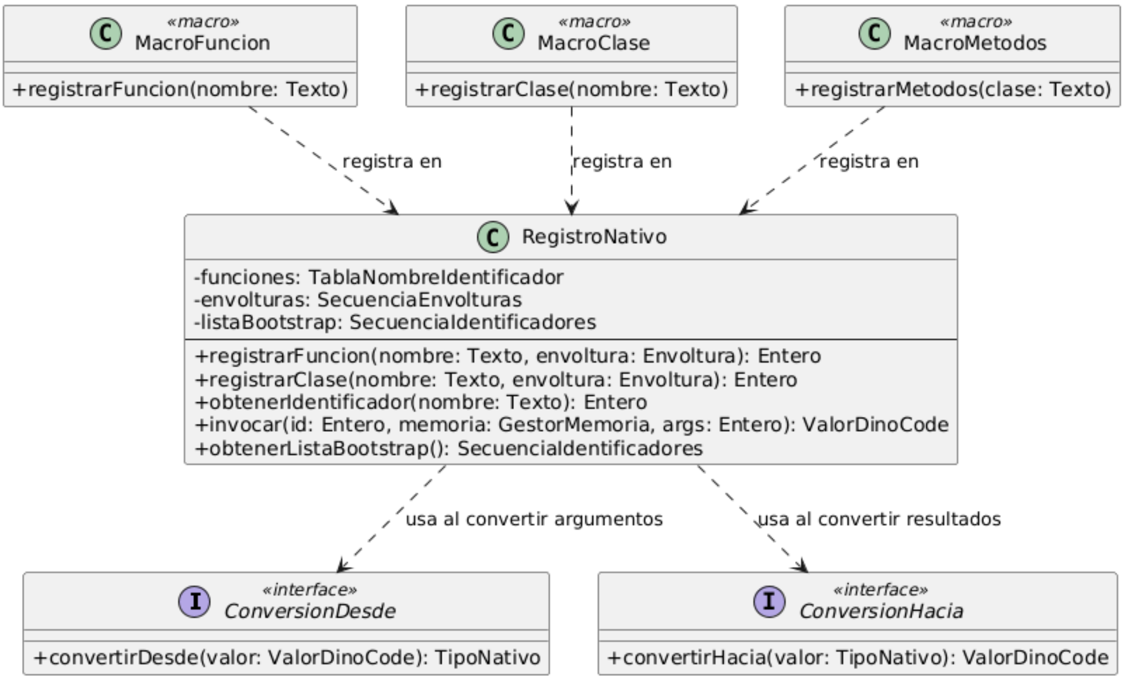
```
#[dinof]
fn add(a: i64, b: i64) -> i64 {
    a + b
}
```

*Nota*. La función `add` podrá ser llamada desde DinoCode en tiempo de ejecución.

A continuación, se muestra la estructura operativa y relacional de la macro:

**Figura 46**

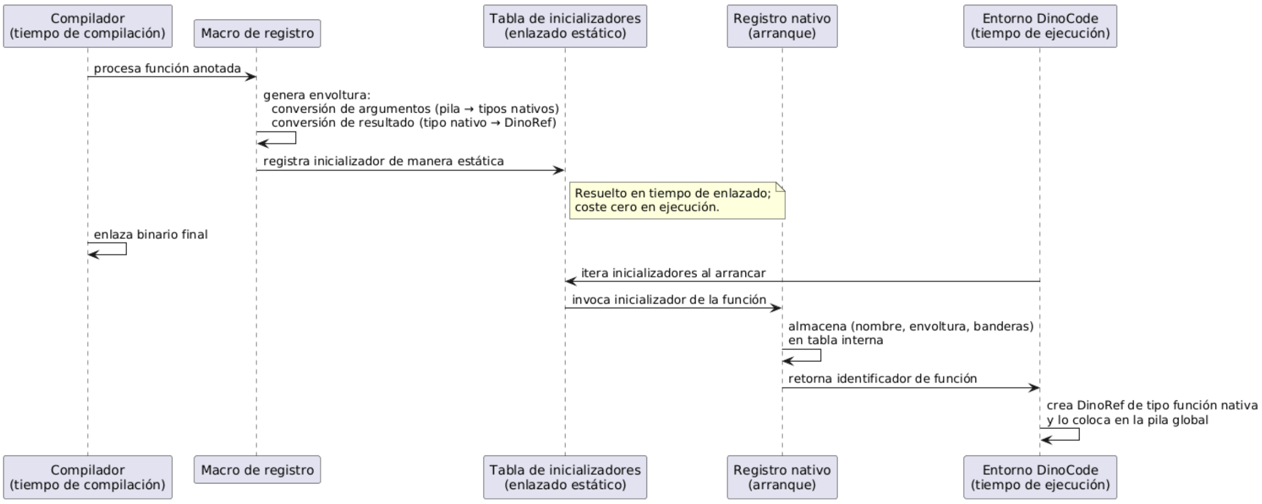
*Diagrama de clases del sistema de interoperabilidad nativa*



El registro de funciones es estático en tiempo de compilación y la conversión de tipos ocurre a través de un acceso directo por índice en la pila de ejecución, sin asignaciones adicionales y sin invocaciones de funciones virtuales. La siguiente figura muestra la secuencia de registro e inicialización de las funciones nativas en DinoCode:

**Figura 47**

*Diagrama de secuencia del registro de una función nativa con #[dinof]*



i

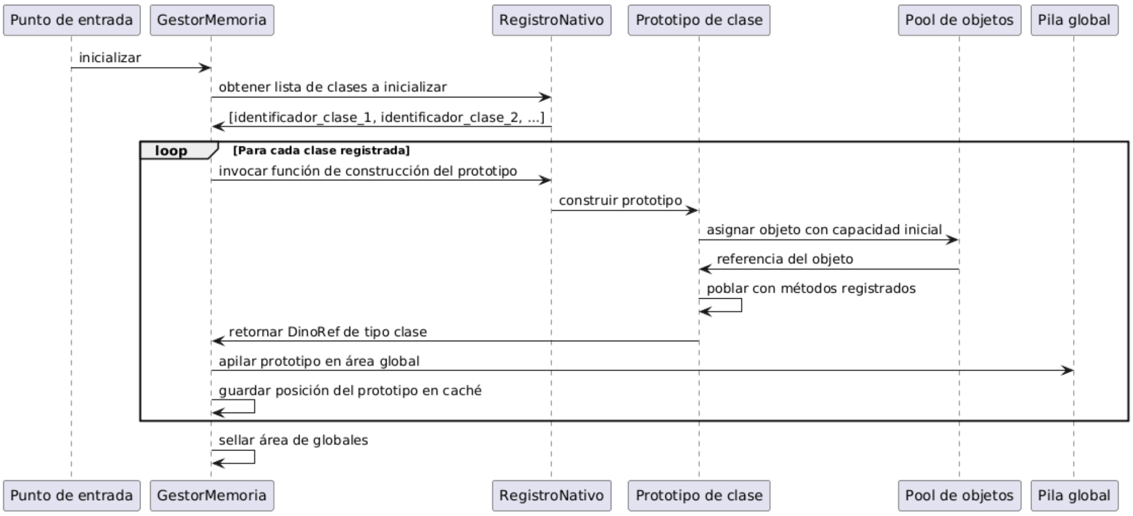
La estructura

#### 4.8.3.2 Registro de clases nativas

El macro `#[dinoclass]` extiende el registro a estructuras complejas (clases) mediante la creación de prototipos y el uso de un Bootstrap, como se muestra en la siguiente figura:

**Figura 48**

*Secuencia de bootstrap de una clase nativa*



### 4.8.4 Operaciones con punteros de pila

El sistema utiliza dos punteros principales para la gestión de la pila:

- **BP (Base Pointer)**: Apunta al inicio del marco de la función actual, permitiendo acceso a parámetros y variables locales
- **SP (Stack Pointer)**: Apunta al tope actual de la pila, indicando dónde se almacenará el siguiente valor

**Tabla 32**

*Operaciones principales con punteros de pila*

| Operación | Comportamiento BP | Comportamiento SP | Descripción |
|---|---|---|---|
| Crear marco | Apunta a inicio de argumentos | Se desplaza para espacio local | Establece nuevo contexto de función |
| Restaurar marco | Recupera puntero anterior | Regresa a posición previa | Finaliza contexto de función |
| Acceso variable | Base + desplazamiento | Sin cambio | Localiza variables locales y parámetros |
| Apilar valor | Sin cambio | Incrementa en una unidad | Almacena dato en pila |

### 4.8.5 Conjunto de instrucciones de la máquina virtual

La máquina virtual de DinoCode es una máquina de pila con instrucciones de ancho variable. Cada instrucción comprende un byte de *opcode* y, para la mayoría de las instrucciones, operandos adicionales. El diseño del conjunto de instrucciones agrupa los opcodes por función:

**Tabla 33**

*Conjunto completo de instrucciones de la máquina virtual DinoCode*

| Categoría | Hex | Opcode | Descripción |
|---|---|---|---|
| Control especial | 0x00 | NOP | Sin operación. |
|  | 0x01 | HALT | Detiene la ejecución de la máquina virtual. |

| | | | |
|---|---|---|---|
| Carga de datos | 0x10 | LOAD_CONST | Carga una constante desde el pool de valores usando su índice. |
| | 0x16 | GET_LOCAL | Obtiene el valor de una variable local usando su índice en la pila. |
| | 0x17 | GET_GLOBAL | Obtiene el valor de una variable global por nombre/índice. |
| | 0x18 | SET_LOCAL | Asigna un valor a una variable local existente. |
| | 0x19 | SET_GLOBAL | Asigna un valor a una variable global. |
| | 0x1A | DROP_LOCAL | Elimina una variable local del seguimiento de referencias. |
| | 0x1B | DROP_GLOBAL | Elimina una variable global del seguimiento de referencias. |
| | 0x13 | TRUE | Apila la constante booleana verdadera. |
| | 0x14 | FALSE | Apila la constante booleana falsa. |
| | 0x15 | NONE | Apila el valor nulo (None). |
| Control de flujo | 0x20 | JUMP | Salto incondicional relativo. |
| | 0x21 | JUMP_IF | Salta a una dirección si el tope de la pila es evaluado como verdadero. |
| | 0x22 | JUMP_IF_NOT | Salta a una dirección si el tope de la pila es evaluado como falso. |
| | 0x23 | RETURN | Retorna None. |
| | 0x24 | RETURN_REF | Retorna una referencia. |
| | 0x25 | RETURN_SELF | Retorna la referencia al objeto actual (self) en un método. |
| | 0x26 | CALL | Invoca una función o método, consumiendo los argumentos de la pila. |
| Gestión de pila | 0x30 | POP | Descarta el elemento superior de la pila. |
| Operaciones binarias | 0x40 | ADD | Suma aritmética. |
| | 0x41 | SUB | Resta aritmética. |

| | | | |
|---|---|---|---|
| | 0x42 | MUL | Multiplicación aritmética. |
| | 0x43 | DIV | División decimal. |
| | 0x44 | FLOOR_DIV | División entera. |
| | 0x45 | MOD | Módulo (resto de la división). |
| | 0x46 | POW | Exponenciación matemática. |
| | 0x47 | EQ | Igualdad de valores o referencias (==). |
| | 0x48 | NE | Desigualdad (!=). |
| | 0x49 | GT | Mayor que (>). |
| | 0x4A | LT | Menor que (<). |
| | 0x4B | GE | Mayor o igual que (>=). |
| | 0x4C | LE | Menor o igual que (<=). |
| | 0x4D | AND | Conjunción lógica |
| | 0x4E | OR | Disyunción lógica |
| | 0x4F | BIT_AND | AND a nivel de bits (&). |
| | 0x50 | BIT_OR | OR a nivel de bits (` |
| | 0x51 | BIT_XOR | XOR a nivel de bits (^). |
| | 0x52 | DOT | Concatenación. |
| Operaciones unarias | 0x60 | NOT | Negación lógica (not). |
| | 0x61 | NEG | Negación numérica / inversión de signo (-). |
| | 0x62 | BIT_NOT | Complemento a nivel de bits (~). |
| Conversión de tipos | 0x63 | TO | Conversión explícita (*to*). |
| | 0x64 | IS_TYPE | Comprobación semántica de tipo dinámico (is). |
| Estructuras agrupadas | 0x70 | MAKE_ARRAY | Asigna un arreglo en el pool de memoria. |
| | 0x71 | MAKE_OBJECT | Asigna un objeto en el pool de memoria. |
| | 0x72 | GET_MEMBER | Lee la propiedad de un objeto mediante indexación. |
| | 0x73 | SET_MEMBER | Escribe una propiedad o inyecta una nueva en un objeto. |

|  |  |  |  |
|---|---|---|---|
|  | 0x74 | STR_BUILD | Ejecuta la interpolación y concatenación múltiple. |
|  | 0x75 | GET_METHOD | Obtiene la referencia a un método y enlaza contextualmente al objeto. |
|  | 0x76 | MAKE_CLASS | Instancia y registra un prototipo de clase en el entorno. |
| Asignación de variables | 0x80-0x85 | *_ASSIGN_VAR | Opcode dinámico para reasignación compuesta (ej. +=, *=). |
|  | 0x86-0x89 | *_FIX_VAR | Pre y post incrementos/decrementos unarios (ej. x++, --y). |
| Asignación flecha | 0x8A-0x8B | ARROW_ASSIGN* | Asignación y lectura de un dato por teclado. |
| Asignación de miembros | 0x90-0x99 | *_MEMBER | Reasignación e incrementos directos de propiedades en objetos. |
| Excepción | 0xFF | UNKNOWN | Instrucción desconocida, aborta como mecanismo de defensa. |

*Nota.* La notación * indica la agrupación de un set continuo de opcodes semánticamente similares por razones de espacio en las categorías especializadas 0x80 - 0x90.

La instrucción CALL (0x26) despacha la invocación de una función. El intérprete consulta el tipo de la función mediante el bit de bandera nativa para decidir si delega a la tabla de nativas o si genera un nuevo marco en la pila de llamadas para una función definida por el usuario. Esta distinción tiene coste temporal equivalente a una instrucción simple condicional.

## 4.9. Análisis de resultados

### 4.9.1 Análisis comparativo de rendimiento entre DinoCode y Python

Este estudio presenta un análisis comparativo de rendimiento entre DinoCode, el lenguaje propuesto, y Python, un lenguaje interpretado de amplio uso en la industria. Para garantizar la precisión de las métricas y mitigar posibles fluctuaciones del sistema operativo, se implementó un motor de medición especializado desarrollado en Rust. Esta herramienta permite capturar de forma síncrona el tiempo total de ejecución y el pico de consumo de memoria mediante la monitorización directa de los procesos.

Para asegurar una comparativa equitativa, el intérprete de Python se ejecutó utilizando las banderas de optimización `-S` (omitir la carga del módulo *site*) y `-OO` (eliminar *docstrings* y optimizar el bytecode), reduciendo así su sobrecarga inicial al mínimo posible.

**Tabla 34**

*Resultados cuantitativos promedio de las métricas de rendimiento evaluadas*

| Categoría de prueba | Tiempo Python (ms) | Tiempo DinoCode (ms) | Reducción de tiempo | Memoria Máx. Python (KB) | Memoria Máx. DinoCode (KB) |
|---|---|---|---|---|---|
| Aritmética básica | 179.85 | 28.73 | 84.02% | 19,612.00 | 2,896.00 |
| Funciones | 166.32 | 14.50 | 91.28% | 20,520.00 | 2,876.00 |
| Strings | 168.07 | 15.29 | 90.90% | 20,964.00 | 2,748.00 |
| Fibonacci | 174.53 | 21.11 | 87.90% | 20,380.00 | 2,756.00 |
| Calculadora | 164.26 | 14.37 | 91.25% | 20,748.00 | 2,848.00 |
| Prueba de estrés | 164.90 | 32.37 | 80.37% | 20,400.00 | 2,760.00 |

Los resultados demuestran una eficiencia superior por parte de DinoCode en todas las categorías evaluadas. Mientras que Python registró un tiempo de ejecución promedio de 169.65 ms, DinoCode completó las mismas tareas en un promedio de 21.06 ms, lo que representa una reducción general del 87.58% en el tiempo de procesamiento.

Esta disparidad se fundamenta en la arquitectura del motor de ejecución. A diferencia de Python, que requiere la construcción y posterior recorrido de un Árbol de Sintaxis Abstracta (AST), DinoCode utiliza una compilación dirigida por sintaxis de una sola pasada. En este modelo, el parser consume los tokens y emite bytecode de forma lineal y directa. Este flujo elimina capas de abstracción intermedias, permitiendo que la Máquina Virtual de DinoCode inicie la ejecución de instrucciones con una latencia significativamente menor.

Respecto al consumo de memoria, DinoCode mantiene una huella de recursos consistentemente baja, promediando los 2,800 KB, en contraste con los 20,000 KB requeridos por la configuración mínima de Python. Esta eficiencia es resultado directo de la implementación del núcleo en Rust y la gestión de memoria optimizada mediante referencias internas (DinoRefs).

**Consideración sobre la madurez del Runtime**

Es imperativo señalar que estos resultados deben analizarse con cautela. Aunque DinoCode supera a Python en velocidad y ligereza en estas pruebas, existe una diferencia sustancial en la densidad de ambos entornos. El intérprete de Python es un entorno maduro que, incluso con optimizaciones, carga subsistemas complejos de gestión de basura y utilidades de sistema que DinoCode, en su estado actual de desarrollo, aún no incorpora. Por tanto, la ventaja de DinoCode reside tanto en su modelo de compilación lineal como en la naturaleza minimalista de su bootstrap, lo que lo posiciona como un motor de ejecución de alta velocidad para tareas de propósito específico.

### 4.9.2 Análisis de los resultados de la encuesta

Se analizan los datos recolectados de la muestra de 30 participantes, estructurando los hallazgos en métricas de usabilidad y potencial de adopción del lenguaje DinoCode.

#### 4.9.2.1 Caracterización de la muestra

La validez del estudio de usabilidad depende de la diversidad técnica de los evaluadores. La Tabla 35 detalla la distribución de los sujetos de prueba según su experiencia previa en programación.

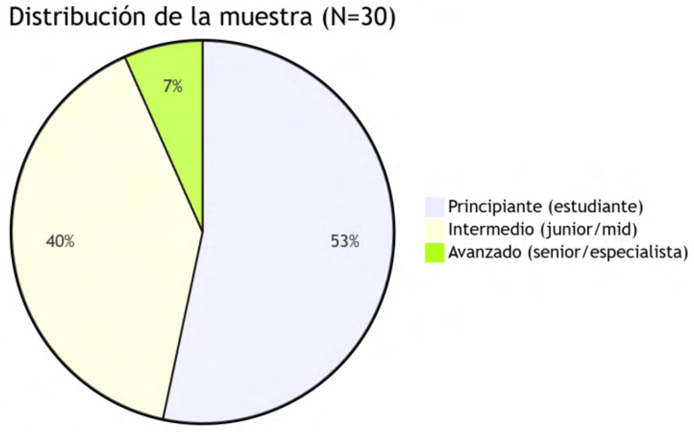
**Tabla 35**

*Distribución de la muestra por nivel de competencia técnica*

| Nivel de experiencia | Frecuencia (n) | Porcentaje (%) | Perfil técnico |
|---|---|---|---|
| Principiante | 16 | 53.33% | Estudiantes de pregrado en ingeniería y computación. |
| Intermedio | 12 | 40.00% | Desarrolladores con experiencia laboral junior o mid-level. |
| Avanzado | 2 | 6.67% | Desarrolladores senior con dominio de múltiples paradigmas. |

**Figura 49**

*Composición porcentual de la muestra según nivel de experticia*



*Nota*. El análisis de la muestra indica una concentración del 93.33% en perfiles principiantes e intermedios, lo cual permite evaluar con precisión la reducción de la curva de aprendizaje en usuarios en formación.

#### 4.9.2.2 Evaluación de dimensiones de usabilidad y sintaxis

Para medir el impacto de las innovaciones sintácticas de DinoCode, se aplicó una escala de Likert (1-5). La Tabla 36 presenta el análisis de las cuatro variables principales del estudio.

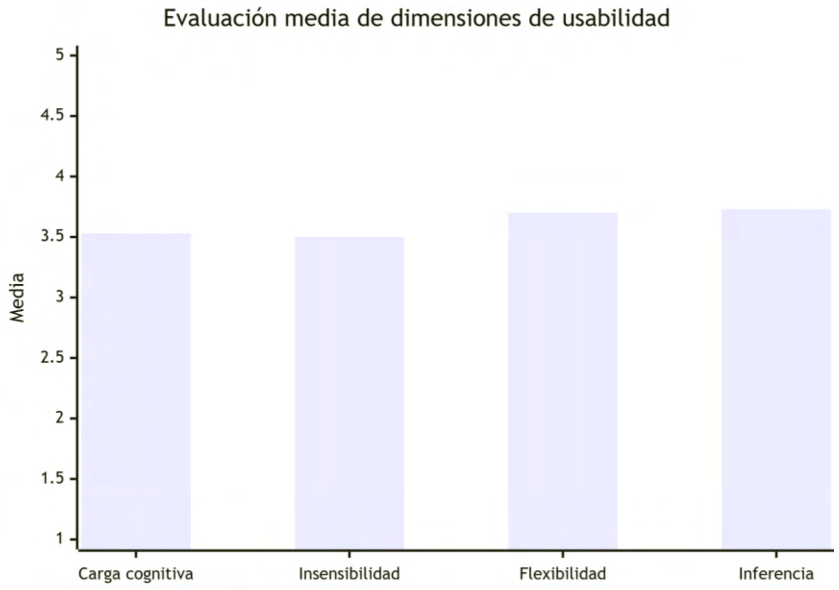
**Tabla 36**

*Métricas de desempeño sintáctico y carga cognitiva*

| Variable | Media ($\bar{x}$) | Interpretación del resultado |
|---|---|---|
| Carga cognitiva | 3.53 | La supresión de símbolos redundantes reduce la fatiga visual. |
| Insensibilidad a mayúsculas | 3.50 | Mitigación de errores por inconsistencia en mayúsculas de identificadores. |
| Libertad de sintaxis | 3.70 | Preferencia por la flexibilidad en el uso de delimitadores. |
| Inferencia de intención | 3.73 | Alta precisión percibida en el motor de interpretación. |

**Figura 50**

*Comparativa de promedios de aceptación por dimensión técnica*



Nota. El análisis cuantitativo destaca la inferencia de intención como la característica con mayor puntaje ($\bar{x} = 3.73$), sugiriendo que la arquitectura lógica del lenguaje compensa efectivamente la ausencia de delimitadores físicos sin sacrificar la claridad funcional para el desarrollador.

#### 4.9.2.3 Análisis de la adopción tecnológica

Un indicador clave de la aceptación de un nuevo lenguaje es la disposición del usuario a incorporarlo en su flujo de trabajo. La Tabla 37 analiza las posturas de los participantes si DinoCode tuviera un ecosistema de librerías completo.

**Tabla 37**

*Proyección de uso y adopción de DinoCode*

| Postura de adopción | n | % | Justificación del usuario |
|---|---|---|---|
| Adopción directa | 14 | 46.67% | Simplicidad y rapidez para scripts rápidos. |
| Adopción condicionada | 14 | 46.67% | Supeditado a la complejidad técnica del proyecto. |
| Resistencia técnica | 2 | 6.66% | Preferencia por lenguajes tradicionales. |

*Nota*. El análisis de los datos indica una aceptación positiva del 93.34% (sumando adopción directa y condicionada). esto demuestra que la propuesta tecnológica de DinoCode es percibida como una herramienta eficiente para optimizar tiempos de desarrollo en tareas de scripting y proyectos de complejidad moderada.

#### 4.9.2.4 Síntesis cualitativa de la Regla de Oro

La inferencia de intención, eje central de este proyecto, fue evaluada mediante respuestas abiertas para identificar beneficios no cuantificables. La Tabla 38 categoriza estas percepciones.

**Tabla 38**

*Dimensiones de beneficio en la inferencia de intención*

| Categoría de impacto | Beneficio identificado | Observación cualitativa destacada |
|---|---|---|
| Flujo de trabajo | Reducción de la fricción sintáctica | "No se pierde el impulso creativo peleando con la sintaxis." |
| Comunicación | Legibilidad natural | "El código es comprensible incluso para no programadores." |
| Psicología | Reducción de estrés | "El entorno es menos intimidante para quienes inician." |
| Mantenimiento | Claridad lógica | "Facilita la detección de errores en la lógica del negocio." |

*Nota*. A través del análisis cualitativo se infiere que la flexibilidad del lenguaje actúa como un catalizador de la productividad. La Regla de Oro no solo mejora la comunicación técnica, sino que mitiga las barreras psicológicas de entrada para usuarios nuevos.

# CONCLUSIONES

El estudio comparativo de lenguajes industriales (Python, JavaScript, Swift, entre otros) permitió identificar una disonancia crítica entre la percepción visual del programador y la interpretación sintáctica de la máquina. Se determinó que patologías de diseño como la corrupción de datos por operadores implícitos, los errores derivados de la inserción automática de punto y coma (ASI) y la fragilidad de las cadenas multilínea, subyacen en gramáticas que ignoran la topografía del código. Este hallazgo validó la oportunidad de mejora de DinoCode al transformar la adyacencia y el espacio de "ruido visual" en operadores de control deterministas.

Se diseñó con éxito una arquitectura basada en el paradigma de la Regla de Oro, donde la disposición física de los elementos dicta la semántica operativa. A diferencia de lenguajes como Julia o Python, DinoCode resuelve ambigüedades estructurales mediante una gramática de adyacencia procesada en un solo paso. El diseño de templates integrados permite la captura de bloques multilínea con fidelidad topográfica absoluta y un costo de procesamiento cero, al ser resueltos estáticamente por el lexer sin requerir limpieza de memoria en tiempo de ejecución.

La construcción del prototipo funcional en Rust demostró una eficiencia prometedora en entornos interpretados, logrando una reducción promedio del 87.58% en tiempos de ejecución y una optimización en el consumo de memoria (promediando 2.8 MB frente a los 20 MB de Python con marcadores específicos de optimización). La evaluación con una muestra de 30 participantes confirmó la viabilidad del modelo con un 93.34% de aceptación positiva, validando que el motor de inferencia de intención reduce significativamente la tasa de errores sintácticos y facilita el desarrollo ágil de software.

## RECOMENDACIONES

Tras identificar que el 46.67% de los usuarios vincula la adopción de un lenguaje a su robustez funcional, se recomienda priorizar el desarrollo de una biblioteca estándar orientada a la ingeniería profesional. Es imperativo implementar módulos nativos para el manejo de protocolos de red, interfaces gráficas y manipulación de datos masivos (Big Data), permitiendo que DinoCode trascienda del prototipo académico a entornos de producción competitivos.

Basándose en los perfiles de rendimiento observados, se recomienda investigar la implementación de mecanismos de compilación en tiempo de ejecución (JIT) y la maduración de los servicios del bootstrap. Aunque DinoCode muestra una ventaja en la latencia inicial por su arquitectura lineal y su entorno ligero, la optimización continua del transductor AEFT permitiría mitigar cualquier sobrecarga futura conforme se incremente la densidad de utilidades nativas, manteniendo una posición aguerrida frente a otros intérpretes.

Se recomienda la integración de DinoCode como herramienta fundamental en la enseñanza de la lógica de programación. Dada su alta tasa de aceptación entre principiantes (53.33% de la muestra) y su capacidad para reducir la fatiga cognitiva mediante la supresión de delimitadores redundantes, el lenguaje resulta ideal para disminuir la curva de aprendizaje inicial, permitiendo que el estudiante priorice la resolución de problemas lógicos sobre la memorización de sintaxis rígidas.

## BIBLIOGRAFÍA

Alam, A. (2025, April 11). *Automatic Semicolon Insertion in JavaScript (ASI)*. Intellipaat. https://intellipaat.com/blog/automatic-semicolon-insertion-in-javascript/

Alroobaea, R., & Mayhew, P. J. (2014, August 27). *How Many Participants are Really Enough for Usability Studies?* Proceedings of 2014 Science and Information Conference, SAI 2014. https://doi.org/10.1109/SAI.2014.6918171

Ángel, M. (n.d.). *Del Funcionamiento y comportamiento de los Compiladores*. Retrieved https://www.ingenieriasimple.com/compiladores/CompilaEnsayo.pdf

Anzules, M. C., & Moya, E. J. G. (2024). Kanban: Una metodología ágil para la gestión eficiente del flujo de trabajo en el desarrollo de software, una revisión sistemática. *Revista Ingenio global*, *3*(1), 17–28. https://doi.org/10.62943/rig.v3n1.2024.68

Astrauskas, V., Matheja, C., Poli, F., Müller, P., & Summers, A. J. (2020). How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–27. https://doi.org/10.1145/3428204

Berger, E. D., Zorn, B. G., & McKinley, K. S. (2002). *Reconsidering Custom Memory Allocation*. https://www.cs.utexas.edu/ftp/techreports/tr01-45.pdf

Cabascango-Bedoya, C. I. (2024). *ANÁLISIS DE LA PROTECCIÓN JURÍDICA DEL SOFTWARE, ESTUDIO COMPARATIVO ENTRE ECUADOR Y ESPAÑA* [Thesis, UNIB.E]. http://repositorio.unibe.edu.ec/xmlui/handle/123456789/759

Chaudhari, P., Rahate, M., Pokale, M., Shaikh, K., & Kasbe, S. (2025). *Evolution & Trends of Programming Language*. *12*(03). https://www.irjet.net/archives/V12/i3/IRJET-V12I360.pdf

de Zwart, I., Chakraborty, S., & Sprokholt, D. (2024). *Memory Layout Optimisation on Abstract Syntax Trees*. https://repository.tudelft.nl/file/File_5748302e-41df-4214-b489-32eca09e39bf?preview=1

Dijkstra, E. (1962). *EWD 28*. https://www.cs.utexas.edu/~EWD/ewd00xx/EWD28.PDF

Granlund, T. & GMP development team. (2026). *Introduction to GMP (GNU MP 6.3.0)*. https://gmplib.org/manual/Introduction-to-GMP

Gribkoff, E. (2013). *Finite State Transducers*. https://www.cs.ucdavis.edu/~rogaway/classes/120/spring13/eric-transducers.pdf

Hernández Sampieri, R., Fernandez-Collado, C. F., & Baptista Lucio, P. (2014). *Metodología de la investigación* (Sexta edición). McGraw-Hill Education. https://apiperiodico.jalisco.gob.mx/api/sites/periodicooficial.jalisco.gob.mx/files/metodologia_de_la_investigacion_-_roberto_hernandez_sampieri.pdf

Jurado Málaga, E. (2008). *Teoría de autómatas y lenguajes formales*. https://dehesa.unex.es/server/api/core/bitstreams/727aecb0-af02-4534-8b4c-c04ce6028601/content

Lakshmanan, R. (2022, August 11). Java String intern: Performance impact. *yCrash*. https://blog.ycrash.io/java-string-intern-performance-impact/

Lovelle, J. M. C. (2001). *LENGUAJES GRAMÁTICAS Y AUTÓMATAS*. https://reflection.uniovi.es/ortin/publications/automata.pdf

Matas, A. (2018). Diseño del formato de escalas tipo Likert: Un estado de la cuestión. *Revista Electrónica de Investigación Educativa*, *20*(1), 38–47. https://doi.org/10.24320/redie.2018.20.1.1347

McCandless, J., & Gregg, D. (2011). Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures: Or: how I learned to stop worrying and love hill climbing. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 161–170. https://doi.org/10.1145/2093157.2093183

Melançon, O., Serrano, M., & Feeley, M. (2025). Float Self-Tagging. *Proceedings of the ACM on Programming Languages*, *9*(OOPSLA2), 1620–1646. https://doi.org/10.1145/3763108

Myers, B. A., Stefik, A., Hanenberg, S., Kaijanaho, A.-J., Burnett, M., Turbak, F., & Wadler, P. (2016). Usability of Programming Languages: Special Interest Group (SIG) Meeting at CHI 2016. *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 1104–1107. https://doi.org/10.1145/2851581.2886434

Pardo, L. M. (2016). *Complejidad Computacional*. https://personales.unican.es/pardol/Docencia/ComputComplex.pdf

Pfenning, F. (2024). *Lecture Notes on Garbage Collection*. https://www.cs.cmu.edu/~janh/courses/411/24/lectures/22-gc.pdf

Pressman, R. S. (2010). *Ingenieria del Software. Un Enfoque Practico* (7.ª). McGraw-Hill Interamericana. https://profesorezequielruizgarcia.wordpress.com/wp-content/uploads/2015/01/ingenieria-del-software-un-enfoque-practico-roger-s-pressman.pdf

Quiroz, I. (2026). *DinoIDE* [Computer software]. https://github.com/BlassGO/DinoIDE

Rafael, R. A. R., & Recto, K. H. A. (2024). *Influence of Language Evolution and Compiler Advances on Program Creation: Implications to Electronics Engineering Education*. https://archium.ateneo.edu/cgi/viewcontent.cgi?article=1171&context=ecce-faculty-pubs

Ruehr, F. (2017). *Parsing Expressions with the Shunting-Yard Algorithm*. https://people.willamette.edu/~fruehr/353/files/ShuntingYard.pdf

Saleil, B., & Feeley, M. (2018a). Building JIT compilers for dynamic languages with low development effort. *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2018*, 36–46. https://doi.org/10.1145/3281287.3281294

Saleil, B., & Feeley, M. (2018b). Building JIT compilers for dynamic languages with low development effort. *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 36–46. https://doi.org/10.1145/3281287.3281294

Sedgewick, R., & Wayne, K. (n.d.). *Regular Expressions*. Retrieved https://algs4.cs.princeton.edu/lectures/keynote/54RegularExpressions.pdf

Sibaja, D. A. S. (2017). *Implementacio´n de la etapa de arranque (bootstrap) de un microprocesador basado en RISC-V para el Sistema de Reconocimiento de Patrones Acu´sticos (SiRPA).* https://repositoriotec.tec.ac.cr/bitstream/handle/2238/10373/implementacion_eta pa_arranque_microprocesador_basado_risc_v_sistema_reconocimiento_patrones_acusticos.pdf?sequence=1&isAllowed=y

Tomassetti, F. (2017, August 1). EBNF: How to describe the grammar of a language. *Strumenta*. https://tomassetti.me/ebnf/

UNCPBA. (2009a). *Gramáticas Libres del Contexto y Lenguajes Libres del Contexto*. https://users.exa.unicen.edu.ar/catedras/ccomp1/ClaseGLC.pdf

UNCPBA. (2009b). *Gramáticas Sensibles al Contexto y Lenguajes Sensibles al Contexto*. https://users.exa.unicen.edu.ar/catedras/ccomp1/ClaseGSC.pdf

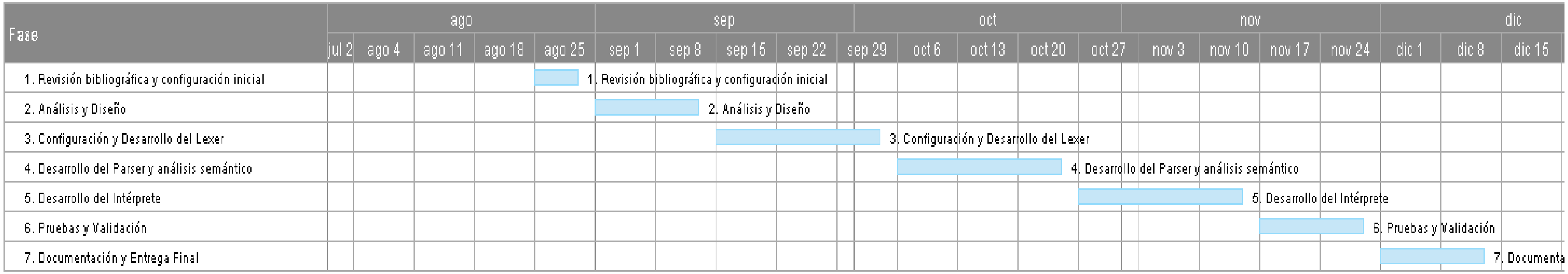
Universidad Nacional de Quilmes. (2014). *Estándar IEEE 754*. https://academy.bit2me.com/wp-content/uploads/2021/04/ieee-754-coma-flotante.pdf

Vladimir, I. (n.d.). *Lenguajes Formales y Autómatas*. Retrieved February 5, 2026, from https://drive.google.com/file/d/1EsGZ4Tr639P7HdHD8K38SloYiZLOgB7e/view?usp=drive_link&usp=embed_facebook

## ANEXOS

# Anexo 1

*Cronograma de Gantt*


Fase
ago
sep
oct
nov
dic
jul 2
ago 4
ago 11
ago 18
ago 25
sep 1
sep 8
sep 15
sep 22
sep 29
oct 6
oct 13
oct 20
oct 27
nov 3
nov 10
nov 17
nov 24
dic 1
dic 8
dic 15
1. Revisión bibliográfica y configuración inicial
2. Análisis y Diseño
3. Configuración y Desarrollo del Lexer
4. Desarrollo del Parser y análisis semántico
5. Desarrollo del Intérprete
6. Pruebas y Validación
7. Documentación y Entrega Final
1. Revisión bibliográfica y configuración inicial
2. Análisis y Diseño
3. Configuración y Desarrollo del Lexer
4. Desarrollo del Parser y análisis semántico
5. Desarrollo del Intérprete
6. Pruebas y Validación
7. Documenta

# Anexo 2

## *Presupuesto Ejecutado*

| Ítem | Descripción | Costo Unitario (USD) | Cantidad | Costo Total (USD) |
|---|---|---|---|---|
| Laptop | Laptop personal utilizada como herramienta principal para desarrollo, pruebas y documentación | $800 | 1 | $800.00 |
| Internet | Revisión bibliográfica | $20 | 1 | $20 |
| Papel | Impresión del proyecto de tesis y documentación adicional | $10 | 2 | $20 |
| Anillado | Entrega final del proyecto de tesis | $3 | 2 | $6 |
| Total | | | | **$846.00** |

# Anexo 3

## *Encuesta de usabilidad*

## 1. Caracterización del perfil técnico

¿Cuál es su nivel de experiencia previa en programación?

- ( ) Principiante (Estudiante de pregrado)
- ( ) Intermedio (Desarrollador junior o mid-level)
- ( ) Avanzado (Desarrollador senior o especialista)

## 3. Evaluación de dimensiones técnicas (Escala de Likert 1-5)

Siendo 1: Muy en desacuerdo / Muy deficiente, y 5: Muy de acuerdo / Muy eficiente.

**Carga cognitiva:** ¿Siente que eliminar los delimitadores redundantes (;, ,, {}) le permite concentrarse más en la implementación de la funcionalidad que en la sintaxis?

( ) 1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

**Insensibilidad a mayúsculas:** ¿Considera que la capacidad del lenguaje para ignorar la distinción entre mayúsculas y minúsculas reduce la posibilidad de errores accidentales?

( ) 1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

**Libertad de sintaxis:** ¿Cómo califica la libertad de elegir entre usar o no delimitadores (ej. paréntesis y comas) según su preferencia personal?

( ) 1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

**Inferencia de intención:** ¿Qué tan preciso le pareció el lenguaje al interpretar su lógica sin necesidad de símbolos físicos obligatorios?

( ) 1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

## 3. Adopción tecnológica

Si DinoCode tuviera un ecosistema completo de librerías, ¿lo usaría para sus proyectos o scripts rápidos?

- ( ) Sí, lo preferiría por su facilidad de uso.
- ( ) Tal vez, dependiendo de la complejidad del proyecto.
- ( ) No, me siento más cómodo con la sintaxis tradicional.

## 4. Pregunta de desarrollo (Cualitativa)

¿Qué aspectos de la Regla de Oro (inferencia de intención) considera que facilitan más la escritura de código?

## 5. Identificación (Opcional)

Correo electrónico o usuario de GitHub: ______________________________________

## 6. Consentimiento informado

¿Autoriza el uso anónimo de sus respuestas para el análisis estadístico de este trabajo de grado?

( ) Sí, autorizo.

# Anexo 4

## *Gramática formal de DinoCode en EBNF*

```
programa ::= {declaracion}

declaracion ::= declaracion_variable
              | declaracion_funcion
              | declaracion_clase
              | sentencia_control
              | expresion_fin

declaracion_variable ::= identificador operador_asignacion expresion_fin
                       | acceso_array operador_asignacion expresion_fin
                       | acceso_miembro operador_asignacion expresion_fin
                       | identificador expresion_redireccion
                       | acceso_array expresion_redireccion
                       | acceso_miembro expresion_redireccion

declaracion_funcion ::= ':' identificador [lista_parametros] bloque

declaracion_clase ::= '::' identificador bloque

lista_parametros ::= {identificador espacio} identificador

bloque ::= 'indent' {declaracion} 'dedent'

sentencia_control ::= 'if' expresion bloque {'elif' expresion bloque} ['else' bloque]
                    | 'while' expresion bloque
                    | 'for' identificador 'in' expresion bloque
```

```
expresion_fin ::= expresion ['fin']

expresion ::= expresion_asignacion

expresion_asignacion ::= expresion_logica [operador_asignacion
expresion_asignacion]

expresion_logica ::= expresion_or {('&&' | '||') expresion_or}

expresion_or ::= expresion_and {'or' expresion_and}

expresion_and ::= expresion_igualdad {'and' expresion_igualdad}

expresion_igualdad ::= expresion_comparacion {('==' | '!=' | '=~')
expresion_comparacion}

expresion_comparacion ::= expresion_concatenacion {('>' | '<' | '>=' | '<=' |
'in' | 'not in') expresion_concatenacion}

expresion_concatenacion ::= expresion_aditiva {'.' expresion_aditiva}

expresion_aditiva ::= expresion_multiplicativa {('+' | '-')
expresion_multiplicativa}

expresion_multiplicativa ::= expresion_potencia {('*' | '/' | '//' | '%')
expresion_potencia}

expresion_potencia ::= expresion_unaria {'**' expresion_unaria}
```

```
expresion_unaria ::= [operador_unario] expresion_primaria

expresion_primaria ::= literal
                     | identificador
                     | acceso_miembro
                     | llamada_funcion
                     | acceso_array
                     | creacion_array
                     | creacion_objeto
                     | expresion_parentizada
                     | template_string
                     | 'as' expresion
                     | 'is' expresion
                     | metodo_implícito
                     | dollar_call

metodo_implícito ::= identificador '.' identificador
[lista_argumentos_implicitos]

acceso_miembro ::= expresion_primaria '.' identificador
                  | expresion_primaria espacio '.' espacio identificador

llamada_funcion ::= expresion_primaria '(' [lista_argumentos] ')'
                  | expresion_primaria [lista_argumentos_implicitos]
                  | identificador [lista_argumentos_implicitos]
                  | metodo_implícito
                  | dollar_call
```

```
dollar_call ::= '$' '(' expresion_primaria [lista_argumentos_implicitos] ')'


lista_argumentos ::= expresion {',' expresion}

lista_argumentos_implicitos ::= expresion {espacio expresion}


acceso_array ::= expresion_primaria '[' expresion ']'

creacion_array ::= '[' [lista_elementos] ']'

lista_elementos ::= expresion {espacio expresion}


creacion_objeto ::= '{' [lista_propiedades] '}'

lista_propiedades ::= propiedad {espacio propiedad}

propiedad ::= identificador expresion

            | identificador ':' expresion


expresion_parentizada ::= '(' expresion ')'


template_string ::= '>' expresion_destino 'newline' bloque_indentado

                  | 'lstring' {interpolacion | string_literal} 'rstring'


expresion_destino ::= identificador

                    | acceso_array

                    | acceso_miembro


bloque_indentado ::= {linea_indentada}


interpolacion ::= '$' identificador
```

```
               | '${' expresion '}'


literal ::= numero_decimal | numero_hex | numero_binario | numero_flotante |
bigint | string_literal | booleano | 'none'


string_literal ::= '"' {caracter | interpolacion | escape_doble} '"'

                | "'" {caracter} "'"


escape_doble ::= '\' '"' | '\' '\' | '\' 'n' | '\' 't' | '\' 'r'


operador_asignacion ::= '=' | '+=' | '-=' | '*=' | '/=' | '//=' | '.=' | '<-'
| '++' | '--'

operador_unario ::= '+' | '-' | '!' | '++' | '--'

espacio ::= ' ' | '\t'

newline ::= '\n'
```

# Anexo 5

## *Guía de referencia del lenguaje de programación DinoCode*

# DinoCode

---

## Guía de Referencia Rápida

Versión 1.0

**Ismael Isaac Quiroz Cachimuel**

Guaranda • Ecuador • 2026

# ÍNDICE DE CONTENIDOS

# Primeros pasos

DinoCode es un lenguaje de programación que se enfoca en ser legible y fácil de aprender. No necesitas mucha sintaxis compleja para comenzar.

## Instalación y primer programa

Para crear tu primer programa, simplemente crea un archivo con extensión `.dino`:

```
:main
    print "¡Hola, mundo!"
```

## Descargar DinoCode

### Opción 1: Entorno de desarrollo integrado

Descarga DinoIDE para Windows desde: https://github.com/BlassGO/DinoIDE/releases

Entorno de desarrollo con ejecución y resaltado de sintaxis para DinoCode.

### Opción 2: Línea de comandos

Descarga DinoCode desde: https://github.com/dinocode-lang/dinocode/releases

Luego ejecuta tu programa desde la terminal:

```
dinocode tu_archivo.dino
```

**Consejo:** Todos los programas deben tener una función `:main` que se ejecutará automáticamente cuando inicie el programa.

# Referencia de sintaxis

## 2.1. Identificadores y palabras reservadas

### Identificadores válidos

Los identificadores son nombres que se dan a variables, funciones y objetos. Pueden contener:

- Letras mayúsculas y minúsculas: `a-z`, `A-Z`
- Dígitos: `0-9` (pero no al principio)
- Guiones bajos: `_`
- Caracteres especiales españoles: `ñ`, `Ñ`, acentos

Ejemplos válidos:

```
contador
mi_variable
año_actual
temperatura_celsius
función_principal
valor_2025
```

### Insensibilidad a mayúsculas

DinoCode es insensible a mayúsculas en sus identificadores:

```
NOMBRE = "Ismael"
print nombre  # Funciona - imprime "Ismael"
print NOMBRE  # Funciona - imprime "Ismael"
print Nombre  # Funciona - imprime "Ismael"
```

### Palabras reservadas

No puedes usar estas palabras como nombres de variables o funciones:

```
if elif else while for return break continye true false none and or not in as
is
```

## 2.2. Números y literales

### Tipos de números soportados

| Tipo | Ejemplos | Descripción |
|---|---|---|
| Enteros | `42`, `-100`, `0` | Números sin decimales |
| Flotantes | `3.14`, `-0.5`, `22.4` | Números con decimales |
| Notación científica | `1.5e3`, `2.5E-2` | Números en notación científica (= 1500, 0.025) |
| Hexadecimales | `0xFF`, `0x1A2B` | Base 16 (prefijo `0x`) |
| Binarios | `0b1010`, `0b1101` | Base 2 (prefijo `0b`) |
| BigInts | `1000n`, `0xFFn`, `0b1010n` | Precisión arbitraria (sufijo `n`) |

```
:main
    x = 10
    y = 22.4

    print x + y           # 32.4
    print 0xFF            # 255 (hexadecimal)
    print 0b1010          # 10 (binario)
    print 1.5e3           # 1500 (notación científica)

    # BigInts para números gigantescos
    grande = 10000000000000000444000000000000000005555000000000n
    print grande
```

## 2.3. Cadenas de caracteres

### Comillas dobles con interpolación

Las cadenas entre comillas dobles permiten incluir variables:

```
nombre = "Ana"
edad = 20
print "Hola $nombre, tienes $edad años"
```

También puedes usar expresiones complejas entre `${...}`:

```
x = 5
y = 10
print "La suma es ${x + y}"
```

### Comillas simples sin interpolación

Las cadenas entre comillas simples se toman literalmente:

```
ruta = 'C:\Windows\System32\cmd.exe'
print 'No se interpreta $variable aquí'
```

# 2.4. Operadores

## Operadores aritméticos

| Operador | Descripción | Ejemplo | Resultado |
|---|---|---|---|
| + | Suma | `5 + 3` | 8 |
| - | Resta | `10 - 2` | 8 |
| * | Multiplicación | `4 * 3` | 12 |
| / | División (decimales) | `15 / 2` | 7.5 |
| // | División entera | `15 // 2` | 7 |
| % | Módulo (residuo) | `15 % 2` | 1 |
| ** | Potencia | `2 ** 3` | 8 |

## Operadores de comparación

| Operador | Descripción | Ejemplo |
|---|---|---|
| == | Igual a | `5 == 5` → true |
| != | No igual a | `5 != 3` → true |
| < | Menor que | `3 < 5` → true |
| > | Mayor que | `10 > 5` → true |
| <= | Menor o igual | `5 <= 5` → true |
| >= | Mayor o igual | `10 >= 5` → true |

## Operadores lógicos

| Operador | Descripción | Ejemplo |
|---|---|---|
| && o and | AND lógico (ambos verdaderos) | `true and false` → false |
| \|\| o or | OR lógico (al menos uno verdadero) | `true or false` → true |
| ! o not | NOT lógico (invierte) | `not true` → false |

## Operador de concatenación

El punto `.` (con espacios) concatena valores como cadenas:

```
resultado = "El número es " . 42
print resultado  # "El número es 42"
```

# Variables y asignación

Las variables almacenan valores que puedes usar y modificar en tu programa.

## Crear y usar variables

No se requiere declarar el tipo. Ve asignando valores directamente:

```
nombre = "Ismael"
edad = 25
altura = 1.70
estatus = true

# Usar variables
print nombre
print "Edad: $edad"
```

## Operadores de asignación

| Operador | Equivalente a | Ejemplo |
|---|---|---|
| = | Asignación simple | `x = 10` |
| += | Suma y asigna | `x += 5` (x = x + 5) |
| -= | Resta y asigna | `x -= 3` (x = x - 3) |
| *= | Multiplica y asigna | `x *= 2` (x = x * 2) |
| /= | Divide y asigna | `x /= 2` (x = x / 2) |

| | | |
|---|---|---|
| `++` | Incrementa en 1 | `x++` (x = x + 1) |
| `--` | Decrementa en 1 | `x--` (x = x - 1) |

```
:main
    contador = 0
    contador += 5      # contador ahora es 5
    contador++         # contador ahora es 6
    print contador
```

## Operaciones de strings

Aunque DinoCode es flexible, la concatenación se maneja principalmente con interpolación:

```
primero = "Dino"
segundo = "Code"

# Interpolación
print "Lenguaje: $primero$segundo"

# O con print en más de un argumento
print primero segundo
```

## Conversiones de tipos

Convierte entre tipos con funciones nativas o el operador `as`:

```
# Usando funciones
numero_texto = "42"
numero = Int(numero_texto)
print numero + 8          # 50

entero = 10
texto = Str(entero)
print "El número es: " texto

# Usando el operador 'as' (más limpio)
print 10 as str           # "10"
print "10" as int         # 10
print 42 as bigint        # 42n
numero = $(input "Número:") as bigint

# Ver tipo de un valor
print Type(42)             # "int"
print Type("texto")        # "string"
print Type(true)           # "bool"
```

# Funciones

Las funciones son bloques de código reutilizable que realizan una tarea específica.

## Definir una función

Las funciones se definen con el prefijo `:`. Pueden ser procedimientos (sin retorno) o funciones (con retorno):

```
:caratula
  print "-----------"
  print "Bienvenido a DinoCode"
  print "-----------"

:suma a b
  return a + b

:main
  caratula
  resultado = suma(5 10)
  print "5 + 10 = $resultado"
```

## Formas de llamar funciones

Las funciones se pueden llamar de diferentes formas según el contexto:

| Forma | Ejemplo | Notas |
|---|---|---|
| Implícita en nivel de sentencia | `caratula` | Válido: sin paréntesis, no ambiguo |

| | | |
|---|---|---|
| Con argumentos implícitos | `suma 5 10` | Válido: argumentos separados por espacios |
| Con paréntesis y argumentos | `suma(5, 10)` | Válido: estilo tradicional con comas opcionales |
| Dollar call (desambiguación) | `$(suma 5 10)` | Válido: resuelve ambigüedad en expresiones |

```
primer_resultado = $(suma 10 20)
segundo_resultado = suma(10 20)
print "Ambos iguales: " primer_resultado " = " segundo_resultado
```

# Control de flujo

Controla qué código se ejecuta según condiciones o repetición.

## 5.1. Condicionales (if/elif/else)

Controla la ejecución según condiciones:

```
edad <- "Escribe tu edad: "

if edad >= 18
    print "Eres mayor de edad"
elif edad >= 13
    print "Eres adolescente"
else
    print "Eres menor de edad"
```

## Múltiples condiciones

Combina condiciones con `and` / `or`:

```
:main
    edad = 25
    tiene_licencia = true

    if edad >= 18 and tiene_licencia
        print "Puedes conducir"
    else
        print "No puedes conducir"
```

## 5.2. Bucles while

Repite bloque mientras la condición sea verdadera:

```
n = 0
while n < 5
    print n
    n = n + 1
```

## 5.3. Bucles for

Itera sobre elementos de un array:

```
for n in [5 6 7 8 9 10]
    print n
```

También con variables de array:

```
items = ["Estudiar" "Programar" "Hacer ejercicio"]
for item in items
    print "- " item
```

# Arrays y colecciones

## 6.1. Arrays básicos

### Crear arrays

Las comas son **opcionales** en arrays. Se pueden usar o no según preferencia:

```
lista = []

# Arrays con comas
numeros = [1, 2, 3, 4, 5]

# Arrays sin comas (válido también)
numeros = [1 2 3 4 5]

# Valores mixtos
mixto = [10 "texto" true 3.14]

# Matrices anidadas sin comas
matriz = [
    [1 2 3]
    [4 5 6]
    [7 8 9]
]
```

### Acceder a elementos

Los índices comienzan en 0:

```
frutas = ["manzana" "plátano" "cereza"]

print frutas[0]    # "manzana"
print frutas[1]    # "plátano"
print frutas[2]    # "cereza"
```

### Modificar elementos

```
lista = [1 2 3]
lista[0] = 100
print lista        # [100 2 3]
```

## 6.2. Métodos nativos de Array

| Método | Descripción | Ejemplo |
|---|---|---|
| push() | Añade al final | `arr.push "valor"` |
| pop() | Extrae y retorna último | `item = arr.pop()` |
| first() | Primero del array | `arr.first()` |
| last() | Último del array | `arr.last()` |
| get(i) | Elemento en índice i | `arr.get(1)` |
| set(i, v) | Asigna en índice i | `arr.set(1 "nuevo")` |
| len | Longitud (propiedad) | `arr.len` |
| clear() | Vacía el array | `arr.clear()` |
| is_empty | ¿Está vacío? (propiedad) | `if arr.is_empty ...` |

```
lista = ["Manzana" "Naranja" "Plátano"]

print "Longitud: " lista.len
print "Primero: " lista.first()
print "Último: " lista.last()

lista.push "Mango"
print "Nueva longitud: " lista.len

ultimo = lista.pop()
lista.set(1 "Mandarina")

if lista.is_empty
    print "Lista vacía"
else
    print "Lista tiene elementos"
```

# Objetos

Los objetos agrupan datos relacionados en pares clave-valor.

## Crear objetos

Las claves no requieren dos puntos. Ambas sintaxis son válidas: `nombre "valor"` o `nombre: "valor"`

```
persona =
{
    nombre "Ismael"
    edad 23
    ciudad "Ibarra"
}

print persona.nombre
print persona.edad
```

## Iterar sobre objetos

```
for clave in persona.keys
    print clave

for valor in persona.values
    print valor
```

## Objetos complejos

```
empresa =
{
    nombre "TechCorp"
    empleados [
        {nombre "Ana" puesto "Gerente"}
        {nombre "Luis" puesto "Desarrollador"}
    ]
    ubicacion {ciudad "Guaranda" país "Ecuador"}
}

print "Empresa: $(empresa.nombre)"
print "Primera empleada: $(empresa.empleados[0].nombre)"
```

## Definir una clase

Las clases se definen con `::NombreClase` como prefijo:

```
::ÁreaCirculo
  :new radio
    self.radio = radio

  :obtener_area
    return 3.14159 * self.radio * self.radio

  :obtener_perimetro
    return 2 * 3.14159 * self.radio

:main
  circulo = $(ÁreaCirculo 5)
  print "Radio: " circulo.radio
  print "Área: " $(circulo.obtener_area)
  print "Perímetro: " $(circulo.obtener_perimetro)
```

## Llamar métodos de clase

Usa `$(objeto.metodo args)` para llamar métodos cuando sea necesario desambiguar:

```
::Persona
  :obtener_info
    return "Soy ${self.nombre}, tengo ${self.edad} años"

:main
  p = Persona()
  p.nombre = "Ana"
  p.edad = 25

  info = $(p.obtener_info)
  print info
```

## Dollar call con métodos

Si la expresión puede ser ambigua, rodea con `$()`:

```
::Calculadora
  :suma a b
    return a + b

  :resta a b
    return a - b

:main
  calc = $(Calculadora)
  r1 = $(calc.suma 10 20)
  r2 = $(calc.resta 15 5)

  print "10 + 20 = " r1
  print "15 - 5 = " r2
```

# Biblioteca estándar

## 8.1. Funciones globales

| Función | Descripción | Ejemplo |
|---|---|---|
| print | Imprime a consola (múltiples argumentos) | `print "Hola" 123` |
| input | Entrada de usuario desde consola | `nombre = input "Nombre: "` |
| int | Convierte a número entero | `int "42"` → 42 |
| float | Convierte a número decimal | `float "3.14"` → 3.14 |
| number | Convierte a número (int o float) | `number "42"` → 42 |
| bigint | Convierte a BigInteger | `bigint "1000"` → 1000n |
| bool | Convierte a booleano | `bool "true"` → true |
| str | Convierte a cadena | `str 42` → "42" |
| Type | Retorna tipo del valor | `Type 42` → "int" |
| Abs | Valor absoluto | `Abs -5` → 5 |
| Sqrt | Raíz cuadrada | `Sqrt 16` → 4 |

### Input de usuario

Se usa el operador <− para entrada:

```
nombre <- "Escribe tu nombre: "
edad <- "Escribe tu edad: "

print "Hola " nombre ", tienes " edad " años"
```

# Ejemplos prácticos

## Calculadora simple

```
:main
  print "=== Calculadora Básica ==="

  a = 10
  b = 22.4

  print "$a + $b = ${a + b}"
  print "$a - $b = ${a - b}"
  print "$a * $b = ${a * b}"
  print "$a / $b = ${a / b}"
  print "$a // $b = ${a // b}" # División entera
```

## Tabla de multiplicar

```
:imprime_tabla numero
  print "Tabla del $numero:"
  i = 1
  while i <= 10
    resultado = numero * i
    print "$numero x $i = $resultado"
    i = i + 1

:main
  imprime_tabla 5
  imprime_tabla 7
```

## Manejo de arrays con métodos nativos

```
lista = ["Manzana" "Naranja" "Plátano"]

print "Longitud: " lista.len
print "Primero: " lista.first()
print "Último: " lista.last()

lista.push "Mango"
print "Nueva longitud: " lista.len

ultimo = lista.pop()
print "Extraído: " ultimo

lista.set(1 "Mandarina")
print "Elemento 1: " lista.get(1)
```

## Estructura de datos con objetos

```
estudiantes =
[
    {nombre "Ana" edad 20 nota 85}
    {nombre "Carlos" edad 21 nota 92}
    {nombre "Diana" edad 20 nota 78}
]

for estudiante in estudiantes
    nombre = estudiante.nombre
    nota = estudiante.nota
    print "$nombre: $nota puntos"
```

# Consejos y mejores prácticas

## La Regla de Oro en acción

Una expresión continúa en la siguiente línea si termina con operador, delimitador abierto, o coma:

```
resultado =
[
    x + y
    x - y
    x * y
    x / y
    x // y
]

print "Resultados: " resultado

mensaje = "Hola " .
          "mundo"

resultado = suma(5 +
            10 * 2)
```

## Ambigüedad y dollar call

Si la llamada a función es ambigua en una expresión, usa `$()`:

```
# Ambiguo - ¿es asignación o llamada?
r = func a b   #  No válido en asignación

# Correcto - usa dollar call
r = $(func a b)  # ✓ Válido y claro

# También válido en nivel de sentencia
func a b   # ✓ Válido sin ambigüedad
```

## Templates con >:

Para bloques multilínea: `> VARIABLE` captura el bloque indentado. El bloque termina cuando encuentra una línea con menor indentación que el inicio:

- **Requerimiento:** El contenido debe tener un margen izquierdo superior al de >
- **Continuidad:** Se detecta por saltos de línea y menor indentación
- **Asignación:** Se asigna a la variable como string con interpolación

### Tipos de templates

#### 1. Template simple

```
> saludo
    Hola, $nombre!
```

#### 2. Template con literales (arte ASCII)

```
> FIGURA
    ***
   *****
   *****
```

#### 3. Template con interpolación compleja

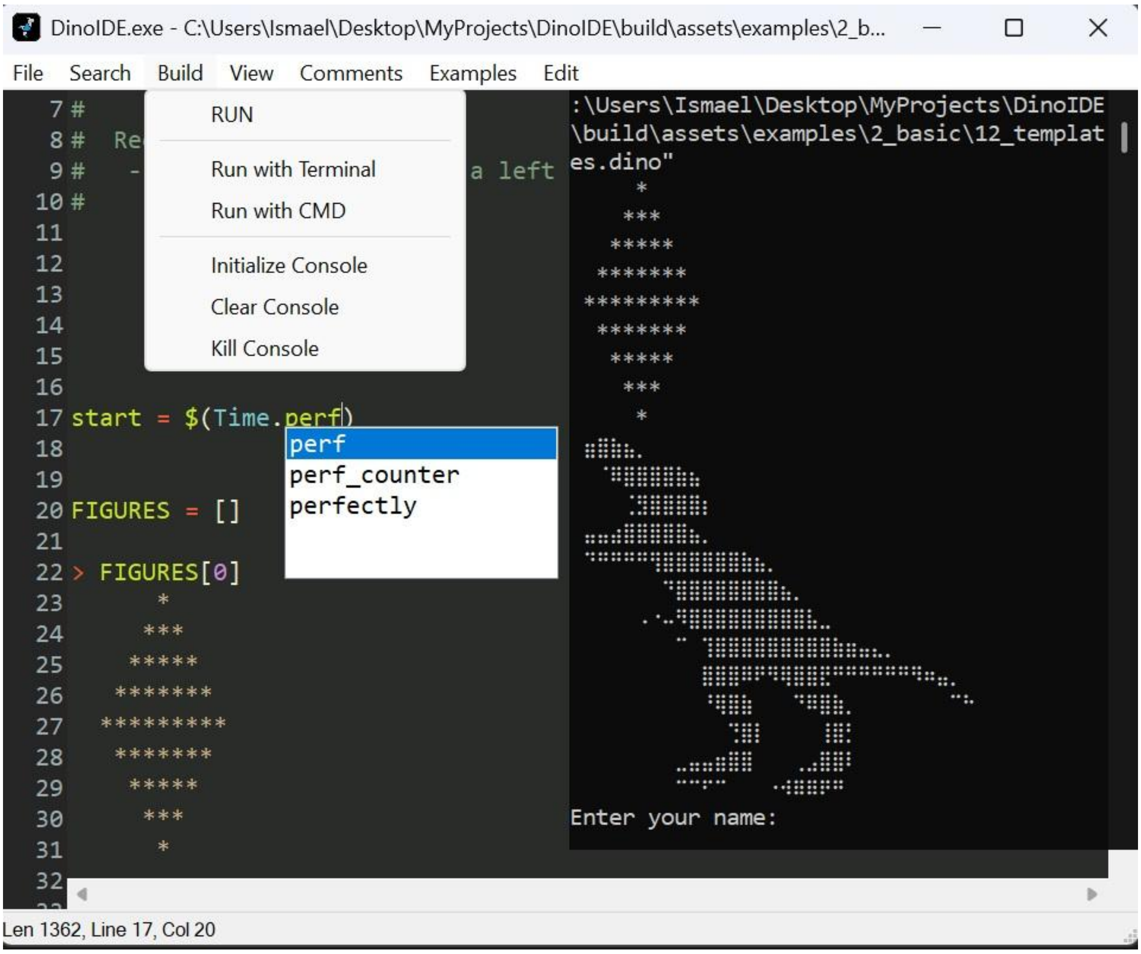
```
> datos_persona
    Nombre: $nombre
    Edad: $edad
    Estado: $activo
```

## Características avanzadas

- **Solo interpolación:** Dentro del bloque solo se reconoce interpolación `$variable` y `${expresión}`, nada más
- **Sin escapes:** No se reconocen caracteres especiales ni escapes, todo es literal
- **Limpieza automática:** Se eliminan automáticamente los espacios del margen izquierdo + 1
- **Cero overhead:** Procesado en tiempo de compilación
- **Integración:** Se puede usar dentro de cualquier bloque del código

# Anexo 6

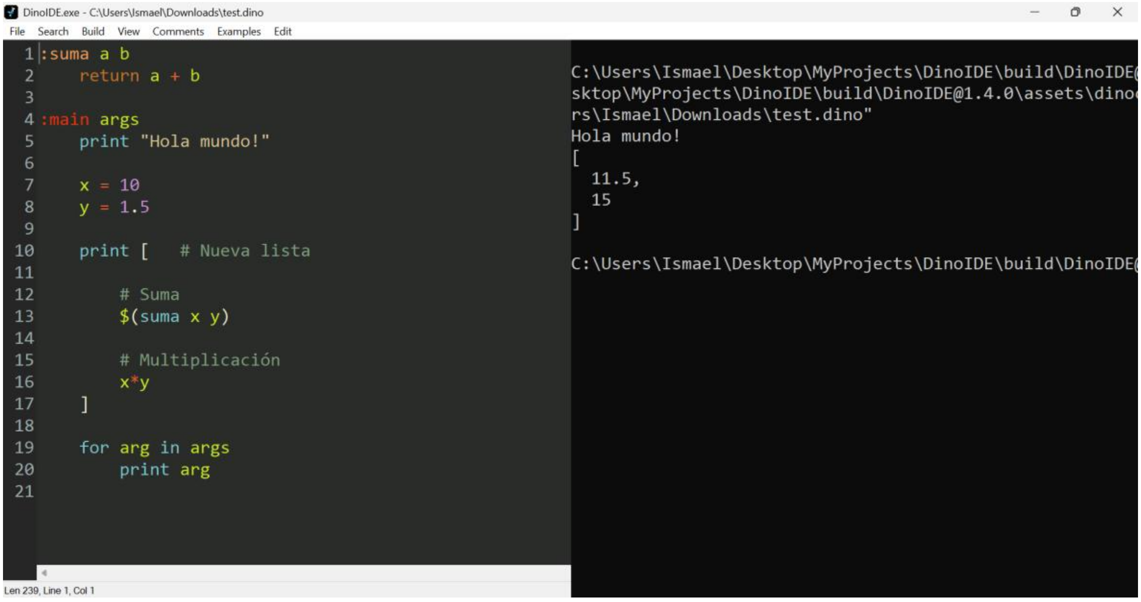
## Script de DinoCode ejecutándose junto al editor DinoIDE



```
#
#   Re
#     -                   a left
#






start = $(Time.perf)


FIGURES = []

> FIGURES[0]
        *
       ***
      *****
     *******
    *********
     *******
      *****
       ***
        *
```

```
:\Users\Ismael\Desktop\MyProjects\DinoIDE
\build\assets\examples\2_basic\12_templat
es.dino"
     *
    ***
   *****
  *******
 *********
  *******
   *****
    ***
     *
```

```
Enter your name:
```

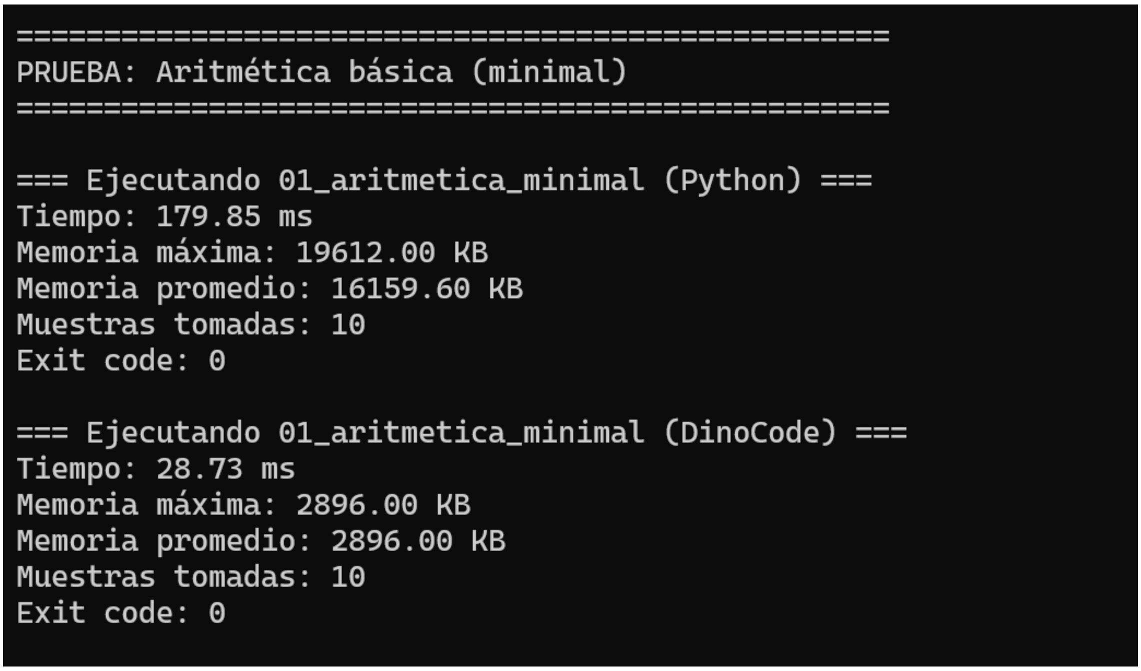


```
:suma a b
    return a + b

:main args
    print "Hola mundo!"

    x = 10
    y = 1.5

    print [    # Nueva lista

        # Suma
        $(suma x y)

        # Multiplicación
        x*y
    ]

    for arg in args
        print arg

```

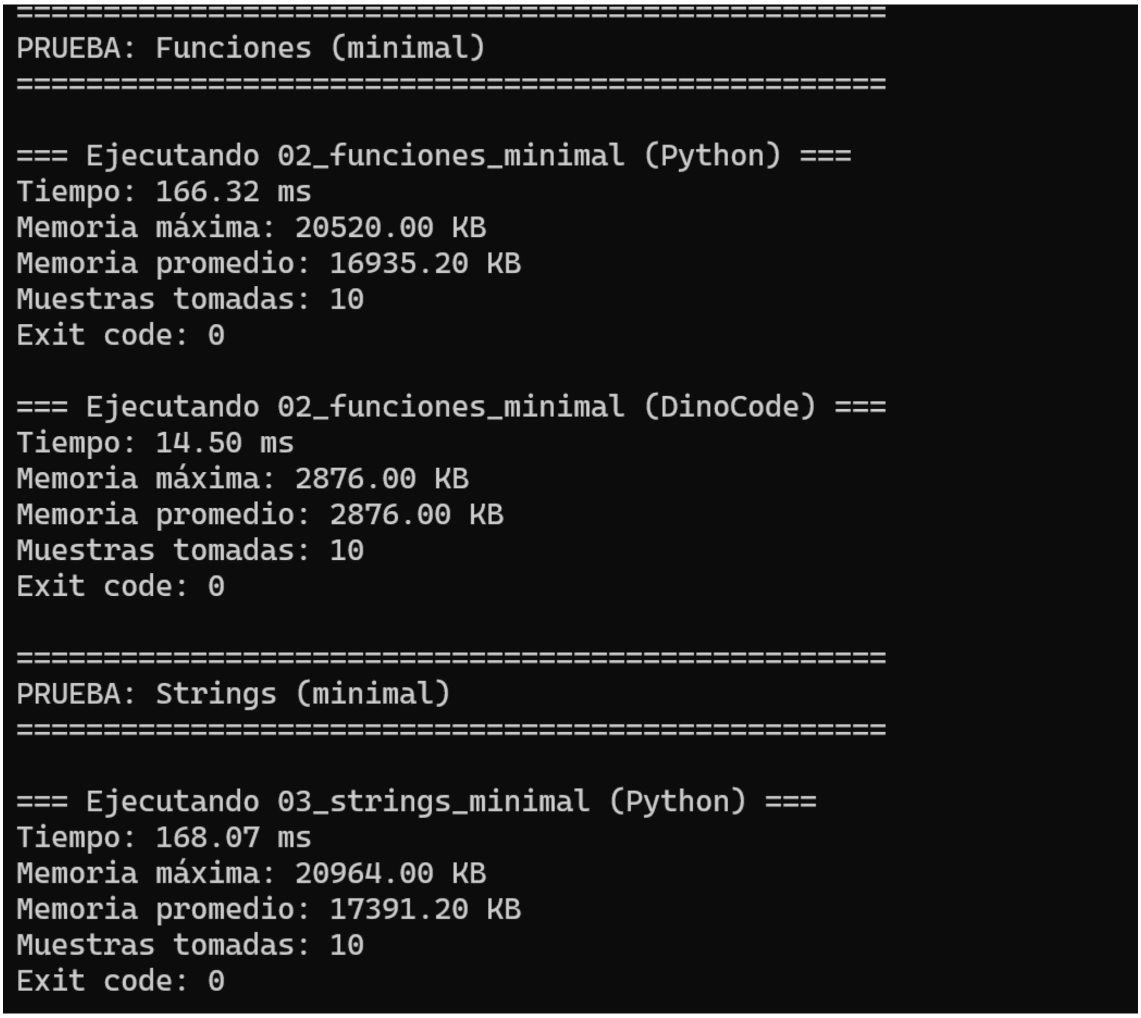
```
C:\Users\Ismael\Desktop\MyProjects\DinoIDE\build\DinoIDE
sktop\MyProjects\DinoIDE\build\DinoIDE@1.4.0\assets\dino
rs\Ismael\Downloads\test.dino"
Hola mundo!
[
  11.5,
  15
]

C:\Users\Ismael\Desktop\MyProjects\DinoIDE\build\DinoIDE
```

# Anexo 7

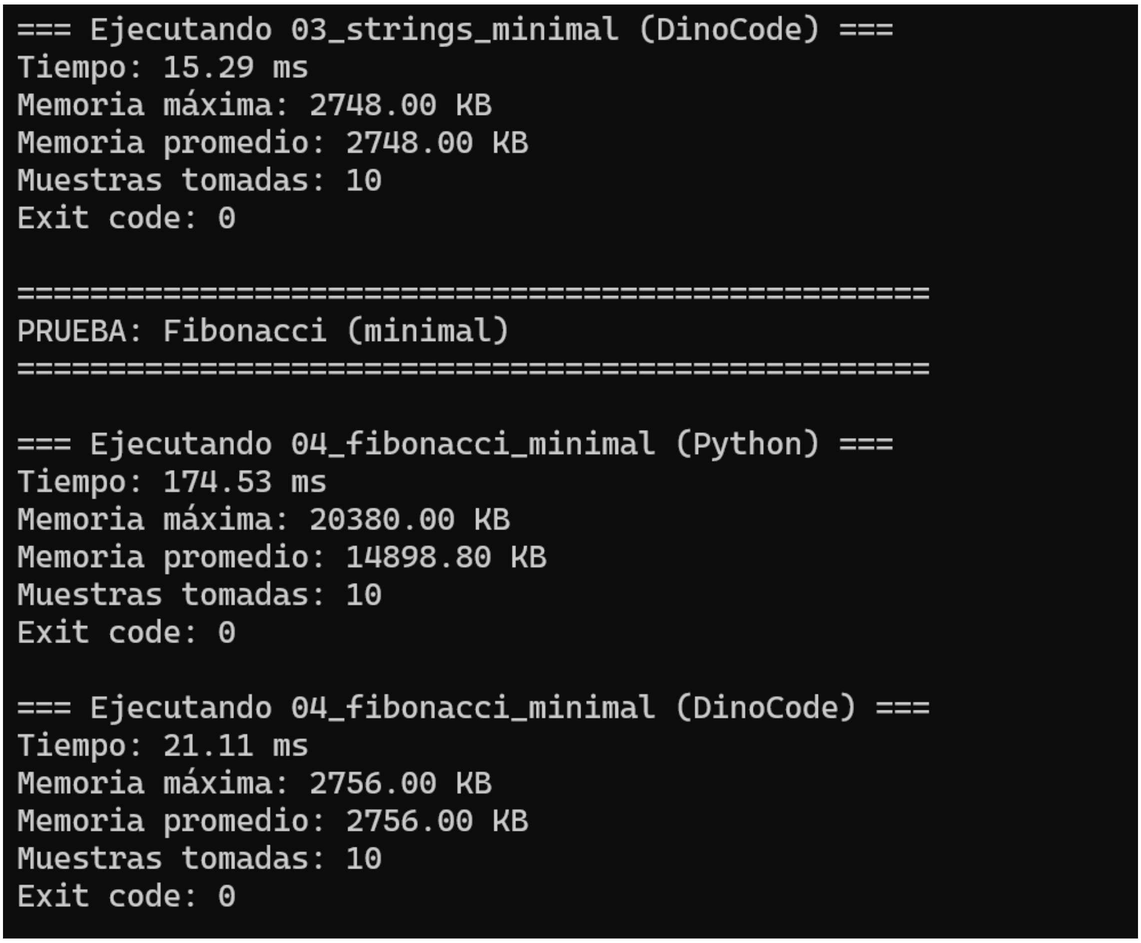
## *Pruebas de rendimiento*



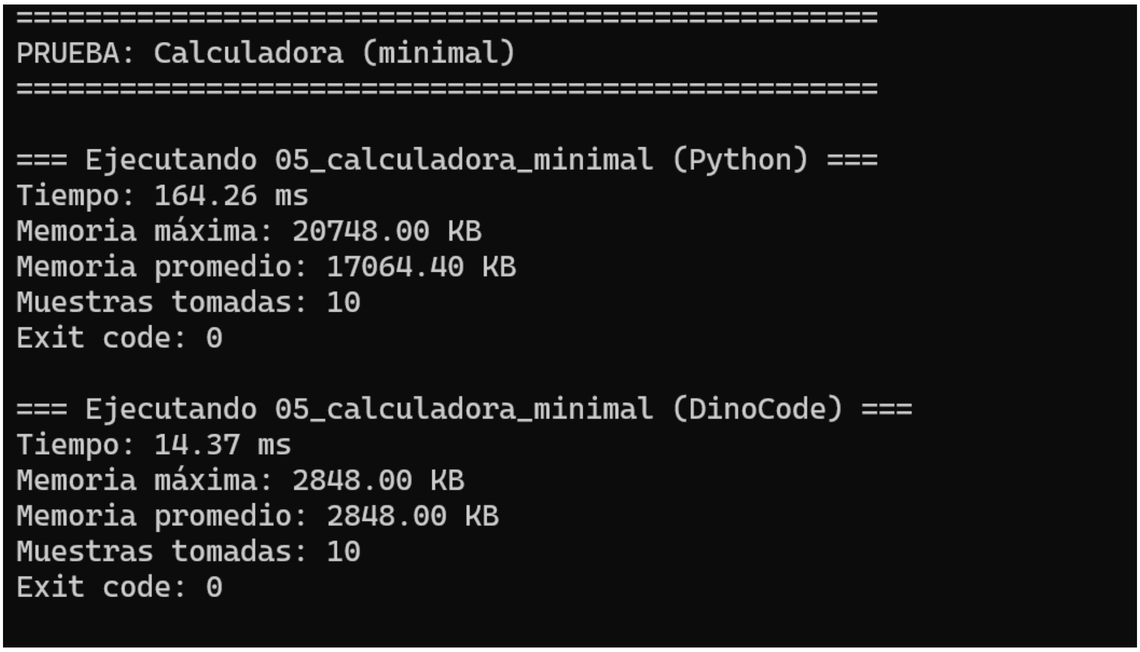
```
==================================================
PRUEBA: Aritmética básica (minimal)
==================================================

=== Ejecutando 01_aritmetica_minimal (Python) ===
Tiempo: 179.85 ms
Memoria máxima: 19612.00 KB
Memoria promedio: 16159.60 KB
Muestras tomadas: 10
Exit code: 0

=== Ejecutando 01_aritmetica_minimal (DinoCode) ===
Tiempo: 28.73 ms
Memoria máxima: 2896.00 KB
Memoria promedio: 2896.00 KB
Muestras tomadas: 10
Exit code: 0
```



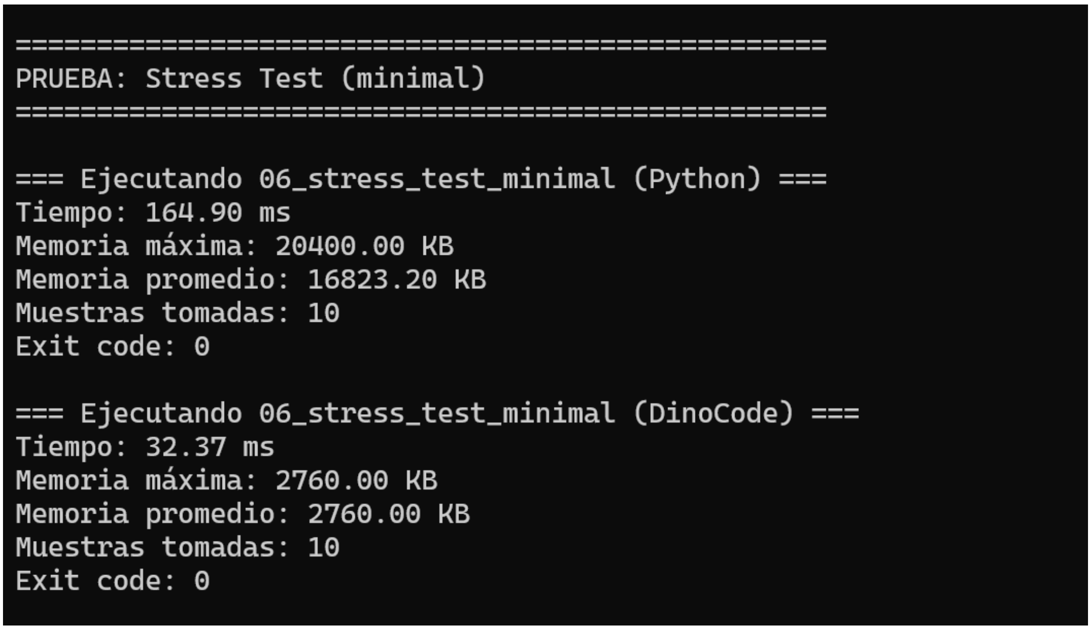
```
==================================================
PRUEBA: Funciones (minimal)
==================================================

=== Ejecutando 02_funciones_minimal (Python) ===
Tiempo: 166.32 ms
Memoria máxima: 20520.00 KB
Memoria promedio: 16935.20 KB
Muestras tomadas: 10
Exit code: 0

=== Ejecutando 02_funciones_minimal (DinoCode) ===
Tiempo: 14.50 ms
Memoria máxima: 2876.00 KB
Memoria promedio: 2876.00 KB
Muestras tomadas: 10
Exit code: 0

==================================================
PRUEBA: Strings (minimal)
==================================================

=== Ejecutando 03_strings_minimal (Python) ===
Tiempo: 168.07 ms
Memoria máxima: 20964.00 KB
Memoria promedio: 17391.20 KB
Muestras tomadas: 10
Exit code: 0
```



```
=== Ejecutando 03_strings_minimal (DinoCode) ===
Tiempo: 15.29 ms
Memoria máxima: 2748.00 KB
Memoria promedio: 2748.00 KB
Muestras tomadas: 10
Exit code: 0

==================================================
PRUEBA: Fibonacci (minimal)
==================================================

=== Ejecutando 04_fibonacci_minimal (Python) ===
Tiempo: 174.53 ms
Memoria máxima: 20380.00 KB
Memoria promedio: 14898.80 KB
Muestras tomadas: 10
Exit code: 0

=== Ejecutando 04_fibonacci_minimal (DinoCode) ===
Tiempo: 21.11 ms
Memoria máxima: 2756.00 KB
Memoria promedio: 2756.00 KB
Muestras tomadas: 10
Exit code: 0
```



```
==================================================
PRUEBA: Calculadora (minimal)
==================================================

=== Ejecutando 05_calculadora_minimal (Python) ===
Tiempo: 164.26 ms
Memoria máxima: 20748.00 KB
Memoria promedio: 17064.40 KB
Muestras tomadas: 10
Exit code: 0

=== Ejecutando 05_calculadora_minimal (DinoCode) ===
Tiempo: 14.37 ms
Memoria máxima: 2848.00 KB
Memoria promedio: 2848.00 KB
Muestras tomadas: 10
Exit code: 0
```



```
==================================================
PRUEBA: Stress Test (minimal)
==================================================

=== Ejecutando 06_stress_test_minimal (Python) ===
Tiempo: 164.90 ms
Memoria máxima: 20400.00 KB
Memoria promedio: 16823.20 KB
Muestras tomadas: 10
Exit code: 0

=== Ejecutando 06_stress_test_minimal (DinoCode) ===
Tiempo: 32.37 ms
Memoria máxima: 2760.00 KB
Memoria promedio: 2760.00 KB
Muestras tomadas: 10
Exit code: 0
```

# Anexo 8

## *Certificado Antiplagio*

UEB
UNIVERSIDAD
ESTATAL DE BOLIVAR

FACULTAD DE CIENCIAS
ADMINISTRATIVAS,
GESTIÓN EMPRESARIAL
E INFORMÁTICA

# ING. DANILO BARRENO EN CALIDAD DE DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR,

## CERTIFICA

Que el trabajo de integración curricular denominado **"Lenguaje de Programación Centrado en la Eficiencia y Facilidad de uso para Programadores"** , presentado por **ISMAEL ISAAC QUIROZ CACHIMUEL** estudiante de la **carrera de Software** pasó el análisis de coincidencia no accidental en la herramienta COMPILATIO reflejando un **porcentaje de similitud del 4%**, como se puede evidenciar en el documento adjunto.

Guaranda, 03 de marzo del 2026

Atentamente,

Ing. Danilo Barreno
**Director**

Certificado de análisis
Compilatio Magister+ | UNIVERSIDAD ESTATAL DE BOLIVAR (UEB)

# Anexo 4.d Estructura Proyecto Tecnológico

ID : 6ac11e6b9d1e4a184ca1026036d708090b402fac

**4%**
Textos sospechosos

**Nombre del fichero** : Anexo 4.d Estructura Proyecto Tecnológico.txt
**Tamaño del archivo original** : 3,15 MB
**Número de palabras** : 19.732
**Número de caracteres** : 146058

**Depositante** : DANILO GEOVANNY BARRENO NARANJO
**Fecha de depósito** : 3 de marzo de 2026
**Tipo de carga** : interface
**fecha de fin de análisis** : 3 de marzo de 2026

## Resumen (sección 1/2)

Localización de los textos sospechosos en el documento :

### Incluido en el porcentaje de textos sospechosos :

**Similitudes** <1%
Pasajes con similitudes a fuentes encontradas en diferentes colecciones.

**Detección de IA** 29%
Textos estilísticamente próximos a un texto generado por una IA. Este índice es un indicador y no una prueba. Comprueba con el autor si domina los conocimientos mencionados en el documento.

**Idiomas no reconocidos** 3%
Pasajes en los que parte del vocabulario utilizado no forma parte del diccionario de la lengua. Puede tratarse de un intento del autor de modificar el texto para evitar ser detectado.

### No incluido en el porcentaje de textos sospechosos :

**Textos entre comillas** 3%
Pasajes entre comillas, a menudo indicativos de una cita.

| N° | Descripciones |
|---|---|
| 8 | https://doi.org/10.1145/2851581.2886434Pardo |
| 9 | https://github.com/BlassGO/DinoIDE |
| 10 | https://people.willamette.edu/~fruehr/353/files/ShuntingYard.pdf |
| 11 | https://doi.org/10.1145/3281287.3281294 |
| 12 | https://users.exa.unicen.edu.ar/catedras/ccomp1/ClaseGLC.pdf |
| 13 | https://users.exa.unicen.edu.ar/catedras/ccomp1/ClaseGSC.pdf |
| 14 | https://academy.bit2me.com/wp-content/uploads/2021/04/ieee-754-coma-flota... |
| 15 | https://drive.google.com/file/d/1EsGZ4Tr639P7HdHD8K38SloYiZLOgB7e/view?usp |