Informe de rendimiento y presentación del Motor Léxico `syntx`

XXXXXXXX, abril de 2025

Resumen

Este documento presenta un informe detallado sobre el rendimiento y el consumo de memoria del motor léxico `syntx`, diseñado con un enfoque modular, predictivo y extensible para múltiples lenguajes de programación. Se incluyen comparaciones con herramientas consolidadas como `javalang`, `tree-sitter`, `rustc_lexer`, entre otras. Este motor léxico surgió originalmente como una herramienta auxiliar para el sistema de construcción `jmake`, donde se requería un análisis léxico rápido y seguro para resolver dependencias de forma incremental. Su diseño modular y expresivo motivó su separación como proyecto independiente.

Demostración de Abstracción sin Coste en Rust

Una de las propiedades clave del diseño del motor `syntx` es que permite una integración directa con cualquier conjunto léxico mediante el sistema de tipos de Rust, sin incurrir en sobrecoste de ejecución. A continuación se muestra un ejemplo funcional mínimo:

```
use syntx::engine::Lexer;
use syntx::langs::java::JavaTokenSet;

let mut lexer = Lexer::<JavaTokenSet>::from_str(source_code);
lexer.tokenize();
```

Gracias al sistema de tipos y la tecnología de genéricos de Rust, el motor es **completamente modular**: el mismo `Lexer<T>` puede adaptarse a distintos lenguajes sin modificar el motor ni reconfigurar el flujo de ejecución. El usuario puede definir un nuevo lenguaje implementando únicamente 5 o 6 funciones sobre un tipo `T` que cumpla los traits `Lexable` y `Delimeted`, junto con una definición manual de su conjunto de tokens. Esto permite que los tokens sean altamente expresivos, personalizados y ajustados a las necesidades gramaticales del lenguaje objetivo.

Estructura del Lexer

```
#[derive(Debug)]
pub struct Lexer<'a, T: Lexable + Delimeted + Eq + Hash + Clone> {
    pub tokens: Vec<T::Token>,
    pub contents: Peekable<Chars<'a>>,
    pub state: State<T>,
}
```

El analizador léxico mantiene un buffer de tokens generados (por ahora), una vista previa sobre los caracteres del código fuente, y una estructura de estado interna parametrizada por el tipo del lenguaje (Autómata Finito con conocimiento de coordenadas, niveles de indentación y contexto en tiempo real). No hay necesidad de usar `Box`, punteros dinámicos ni almacenamiento en heap adicional. Todo el trabajo es resuelto en tiempo de compilación mediante especialización monomórfica.

Representación de Gramáticas mediante Árboles Ponderados con Reserva de Memoria en Arena

El motor `syntx` utiliza una estructura de árbol genérica altamente optimizada, pensada para almacenar reglas sintácticas y caminos de predicción léxica además de un futuro AST. Cada nodo es almacenado en memoria continua usando el allocator `bumpalo`, lo cual permite una inserción masiva y constante sin realocaciones dinámicas individuales.

Definición de la Estructura en Rust

```
#[derive(Debug)]
pub struct Node<'bump, T>
where
    T: std::hash::Hash + std::cmp::Eq + Clone,
{
    pub leafs: AHashMap<T, &'bump RefCell<Node<'bump, T>>>,
    pub left: Option<&'bump RefCell<Node<'bump, T>>>,
    pub right: Option<&'bump RefCell<Node<'bump, T>>>,
    pub is_ast_node: bool,
    pub weight: usize,
    pub value: Option<T>,
    pub end: bool,
}

#[derive(Debug)]
pub struct Tree<'bump, T>
where
    T: Hash + Eq + Clone,
{
    pub arena: &'bump Bump,
    pub cursor: &'bump RefCell<Node<'bump, T>>,
    pub root: &'bump RefCell<Node<'bump, T>>,
    pub stack: BVec<'bump, &'bump RefCell<Node<'bump, T>>>,
}
```

Resumen técnico:

- Cada nodo (`Node`) representa un valor gramatical (terminal o no terminal) y apunta a un conjunto de hijos mediante un `AHashMap`.
- Todos los nodos viven en una `arena` compartida (`bumpalo::Bump`), lo que minimiza la fragmentación.
- El árbol soporta navegación por cursor y backtracking mediante una pila interna.

Ventajas del Diseño

- La inserción de nuevas ramas se realiza en tiempo constante, al igual que el desplazamiento transversal y backtracking.
- Toda la estructura vive en una sola asignación de memoria, lo cual mejora la localidad de caché, además de solventar problemas de tiempo de vida de la memoria en el **Heap**, de lo cual se ocupa el **Borrow Checker** de rust, ya que syntx está pensado desde un punto de vista **seguro** que no dé lugar a leaks de memoria.
- Facilita la serialización, visualización, y exportación de gramáticas formales.

Evaluación sobre Código Java

Archivo utilizado: `DataBaseMetadata.java (NASA JMARS)`

```
========== Resultados del Benchmark ==========
Líneas        : 8094
Tokens        : 35126
Tiempo        : 0.0048 s
Memoria       : 1.31 MB
Líneas/seg    : 1.697.167
Tokens/seg    : 7.365.293
```

Observación: El motor demuestra una velocidad equiparable a analizadores léxicos utilizados en compiladores como `rustc`, con un uso mínimo de memoria.

Evaluación sobre Código en C

Archivo utilizado: `decode.c (FFmpeg src)`

```
========== Resultados del Benchmark ==========
Líneas        : 2223
Tokens        : 11072
Tiempo        : 0.0019 s
Memoria       : 0.38 MB
Líneas/seg    : 1.146.113
Tokens/seg    : 5.708.399
```

Observación: El lexer mantiene su rendimiento en código C real con alta densidad léxica y estructuras complejas.

Comparativa de Thoughput entre Lexers

Herramienta	**Implementación**	**Tokens/seg**	**Características principales**
StreamTokenizer	Java	679.000	Mínima sensibilidad gramatical
javalang	Python	290.000	Parser completo de Java
pylyzer	Python	200.000	Sensible a espacios y estructura
tree-sitter	C/Rust	1.060.000	Análisis sintáctico por gramáticas
rustc_lexer	Rust	17.662.500	Nativo (sin inferencia)
TypeScript Lexer	TypeScript	750.000	Lexer oficial del compilador TypeScript
Acorn (JavaScript)	JavaScript	729.000	Lexer modular basado en AST minimalista
syntx	Rust	**7.165.668**	Lexer predictivo y modular

Table 1: Aproximación de rendimiento en análisis léxico (tokens por segundo)

Conclusión técnica:

- `syntx` alcanza un rendimiento comparable a `rustc_lexer` con una arquitectura extensible.
- Su uso de memoria es inferior al de la mayoría de herramientas de análisis existentes.
- El diseño permite su integración en compiladores, analizadores estáticos e IDEs.

Comparativa de Uso de Memoria

Lenguaje	Archivo	Memoria (MB)	Tokens	Tiempo (s)	Tokens/seg
Java	DataBaseMetadata.java	1.31	35126	0.0048	7.365.293
C	decode.c	0.38	11072	0.0019	5.708.399

Table 2: Comparación del uso de memoria entre lenguajes

Consideraciones Técnicas

- Las mediciones fueron realizadas con compilaciones optimizadas mediante `cargo run --release`.
- El uso de memoria fue obtenido desde `/proc/<pid>/statm` (Linux).
- Todas las pruebas incorporaron límites estrictos de tiempo y memoria como validación.
- El lexer está diseñado para un comportamiento determinista y eficiente en proyectos de gran escala.

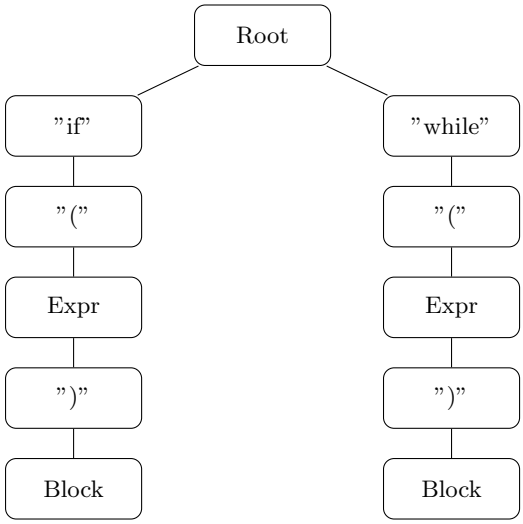

Figure 1: Árbol de producción sintáctica almacenado en arena

Planes a futuro

Se identifican múltiples direcciones de desarrollo posibles para expandir la funcionalidad y robustez del motor léxico y sintáctico `syntx`:

- Propagación de errores de sintaxis antes de llegar a fases avanzadas de creación de ASTs.
- Predicciones deterministas guiadas por pesos ajustados dinámicamente durante la interpretación léxica.
- Arquitectura multi hilo con buffers de datos compartidos para maximizar el paralelismo en la tokenización.
- Implementación de algoritmos *fuzzy* para predicción de errores y tolerancia a errores sintácticos en entradas parciales.
- Análisis incrementales mediante uso de hashes funcionales en lugar de sintácticos para evitar trabajo redundante.

- Generación de tokens especializada para lenguaje natural, con inferencia de tipos gramaticales y sintácticos orientada a tareas de aprendizaje automático.
- Métodos de inserción automática de rutas gramaticales en el árbol mediante reglas inferidas o declarativas.

Ventajas de los Enum Wrappers en Rust

Una de las características más poderosas del ecosistema de tipos en Rust es la capacidad de modelar estructuras léxicas complejas mediante enumeraciones anidadas y envueltas, también conocidas como *enum wrappers*.
Este patrón permite definir tokens jerárquicos que encapsulan variantes dentro de otras variantes, preservando información semántica sin sacrificar eficiencia ni seguridad en tiempo de compilación.

Ejemplo real de Tokenización para Java:

```
pub enum JavaToken {
    Identifier(JavaIdentifier),
    Operator(JavaOperator),
    Delimeter(JavaDelimeters),
}

pub enum JavaIdentifier {
    Keyword(Keyword),
    CharLiteral(String),
    StringLiteral(String),
    Integer(String, JavaBase),
    Unknown(String),
}
```

Gracias a esto, el lexer puede producir un flujo de tokens altamente expresivo y preciso, manteniendo toda la información relevante para las fases posteriores de análisis. Además, al estar resuelto en tiempo de compilación mediante variantes concretas.
Este enfoque también permite un tipo de coincidencia exhaustiva con `match` que mejora la legibilidad, mantenibilidad y seguridad del compilador o analizador sintáctico que lo utilice.
Este algoritmo predictivo se implementa sin costo adicional de memoria, ya que el iterador es clonado superficialmente y no se incurre en copia profunda del contenido.

Demostración de expresividad léxica con enums anidados

Para ilustrar la riqueza semántica de los tokens generados por el motor, se muestra a continuación un fragmento reducido de código Java junto a su representación tokenizada mediante enums anidados (la identificacion de tipos primitivos en lenguajes altamente tipados está en desarrollo):

Listing 1: Demo.java - Fragmento simple con riqueza léxica

```
public class Demo {
    public static void main(String[] args) {
        int x = 42;
        String msg = "Hola\nMundo";
        if (x >= 10) {
            System.out.println(msg);
        }
    }
}
```

Tokens generados: (Identificadores como Strings para claridad, normalmente normalizados)

```
Identifier(Keyword(Public))
Identifier(Keyword(Class))
Identifier(Unknown("Demo"))
Delimeter(LBrace)
Identifier(Keyword(Public))
Identifier(Keyword(Static))
Identifier(Keyword(Void))
Identifier(Unknown("main"))
Delimeter(LParen)
Identifier(Unknown("String"))
Delimeter(LBracket)
Delimeter(RBracket)
Identifier(Unknown("args"))
Delimeter(Rparen)
Delimeter(LBrace)
Identifier(Keyword(Int))
Identifier(Unknown("x"))
Operator(Assign)
Identifier(Integer("42", Decimal))
Delimeter(Semicolon)
Identifier(Unknown("String"))
Identifier(Unknown("msg"))
Operator(Assign)
Identifier(StringLiteral("Hola\nMundo"))
Delimeter(Semicolon)
Identifier(Keyword(If))
Delimeter(LParen)
Identifier(Unknown("x"))
Operator(Geq)
Identifier(Integer("10", Decimal))
Delimeter(Rparen)
Delimeter(LBrace)
Identifier(Unknown("System"))
Operator(Dot)
Identifier(Unknown("out"))
Operator(Dot)
Identifier(Unknown("println"))
Delimeter(LParen)
Identifier(Unknown("msg"))
Delimeter(Rparen)
Delimeter(Semicolon)
Delimeter(RBrace)
Delimeter(RBrace)
Delimeter(RBrace)
```

Este ejemplo demuestra cómo una única pasada del lexer puede clasificar y jerarquizar información léxica compleja mediante enums anidados, permitiendo estructuras de análisis posteriores sin ambigüedad ni pérdida de contexto.

Ejemplo de tokenización: caso sintáctico complejo

Prueba de esfuerzo: validación del lexer

Con el objetivo de evaluar la robustez del analizador léxico, usaremos el siguiente programa, sintácticamente válido pero deliberadamente complejo, diseñado para maximizar el uso de casos límite: literales binarios y hexadecimales, expresiones ternarias encadenadas, estructuras anidadas en profundidad y combinaciones lógicas de operadores. Este fichero simula un caso límite manteniéndose conforme a la gramática del lenguaje Java.

Listing 2: Torture.java - Diseñado para poner a prueba el lexer

```
public class Torture {
    public static void main(String[] args) {
        int x = 0x1F + 42, y = 0b1010, z = (x << 2) & ~(y | 0xFF);
        String s = "Preprate:␣{\n\tLgica␣intensa!\\\"";
        if (x > y && ((z != 0) ? true : false ? true : false)) {
            System.out.println("Por␣qu␣haras␣esto?");
        } else if (x == (y = z = x >>> 1)) {
            s += "Java␣>>>␣cordura";
        }
        for (int i = 0, j = 0; i < 10 && j++ < 5; i++) {
            System.out.print((i & j) == 1 ? "*" : ".");
        }
        final int result = x + y + z + (int)(Math.pow(2.5e2f, 1));
        System.out.println("␣=␣" + c + "␣|␣Resultado␣=␣" + result);
    }
}
```

Esta prueba incluye:

- Operadores bit a bit: `&`, `|`, `~`, `<<`, `>>>`
- Literales hexadecimales y binarios: `0x1F`, `0b1010`
- Literal en coma flotante con notación científica: `2.5e2f`
- Lógica ternaria profundamente anidada
- Secuencias de escape en cadenas de texto:
  `n`,
  `t`,
  `\`
- Asignaciones encadenadas y bucles `for` con múltiples variables

Resumen del análisis léxico

- Tokens generados: **185**
- Categorías léxicas: `Keyword`, `Identifier`, `Operator`, `Delimeter`, `Literal`
- Cadenas escapadas y sufijos numéricos identificados correctamente

Métricas de rendimiento

- Tiempo de ejecución: **0.0000 s** (`time target/release/syntx`)
- Memoria: **0.12 MB**
- Líneas por segundo: **490.990**
- Tokens por segundo: **4.541.660**

Anexo: Especificaciones técnicas del motor "under the hood"

El motor `syntx` implementa un conjunto de estrategias específicas para lograr un equilibrio entre rendimiento, expresividad y seguridad en tiempo de compilación. A continuación se detallan algunas de las decisiones clave en su arquitectura interna:

Reconocimiento de secuencias de comentarios mediante lectura predictiva

El reconocimiento de secuencias compuestas de comentarios se realiza siguiendo una estrategia *greedy* basada en el principio de *maximal munch*. El lexer clona superficialmente el iterador de entrada y lee la mayor secuencia posible de caracteres que coincida con una secuencia válida, deteniéndose en el primer fallo. Esta técnica permite distinguir correctamente entre símbolos como `//`, `/**`, `/*` y `///!`, sin necesidad de lookahead explícito ni retroceso.

Gestión de memoria sin heap tradicional

No se utilizan asignaciones dinámicas convencionales como `Box` o `Rc`. En su lugar, toda la memoria dinámica requerida por las estructuras de árbol es gestionada mediante el allocator `bumpalo`, garantizando inserciones constantes, sin fragmentación, y con un modelo de propiedad sencillo de verificar por el compilador.

Separación de responsabilidades

La lógica léxica (tokenización) y la lógica gramatical (predicción estructural) están desacopladas. El lexer opera únicamente sobre la entrada y genera un flujo de tokens, mientras que el árbol de predicción gestiona rutas válidas sin intervenir en la lectura de caracteres. Esto permite probar y extender ambos subsistemas por separado.

Sin introspección de tipos en tiempo de ejecución

Toda la estructura de tipos es conocida y validada en tiempo de compilación, permitiendo optimizaciones agresivas por parte del compilador y evitando ramas condicionales innecesarias para lenguajes fuertemente tipados.

Lectura inmutable del input

El lexer nunca copia ni muta el código fuente. Trabaja sobre un iterador de caracteres por referencia, lo cual minimiza el uso de memoria y garantiza acceso seguro sin duplicaciones. Esto permite tokenizar archivos de gran tamaño con un perfil de memoria mínimo.

Reducción de ruido en el flujo de tokens

El lexer almacena únicamente los tokens léxicamente relevantes. No conserva espacios, tabulaciones ni otros caracteres triviales para lenguajes en los que tales elementos no afectan la interpretación semántica. Esta decisión permite reducir el uso de memoria, disminuir el tiempo de inserción de tokens, y evitar análisis adicionales innecesarios.
En este sentido, el comportamiento del lexer es analógico al de un filtro de paso bajo: elimina el "ruido" superficial del código fuente y transmite únicamente la señal estructural que alimenta al árbol gramatical y a las etapas posteriores de análisis.