# Informe Analizador Léxico y Sintáctico — Proyecto de Compiladores

**Autor:** [Cristhian Santiago Nastar, Christian Andrés Pastrana]

**Proyecto:** Documento de la fase de implementación: Analizador Léxico y Sintáctico

**Resumen** Este documento describe el análisis del código fuente del compilador sencillo incluido en el archivo proyecto.zip, y presenta la documentación solicitada: conjunto de tokens y expresiones regulares, implementación del analizador léxico, gramática libre de contexto diseñada, implementación del analizador sintáctico (parser), pruebas con entradas válidas e inválidas.

**Universidad cooperativa de Colombia**

**Fecha:** 06/10/2025

## Índice

1. Introducción
2. Arquitectura del Sistema
3. Lenguaje de Programación a Compilar
4. Clases y Métodos Desarrollados
5. Reglas Sintácticas Mapeadas
6. Ejemplos de Entradas y Salidas
7. Código Fuente (Analizador Léxico y Sintáctico)
8. Conclusiones

## 1. Introducción

El presente informe documenta el desarrollo de un analizador léxico y sintáctico, implementado en Python, diseñado para procesar un subconjunto del lenguaje JavaScript.  
El sistema es capaz de:  
  
- Analizar el flujo de caracteres y convertirlos en tokens.  
- Validar la estructura gramatical del código fuente.  
- Generar un árbol sintáctico abstracto (AST).  
- Reportar errores léxicos y sintácticos con línea y columna.  
  
El aborda las fases iniciales del proceso de compilación.

## 2. Arquitectura del Sistema

El sistema se estructura en tres componentes principales:  
  
- Analizador Léxico (Lexer): Identifica los tokens a partir del texto fuente.  
- Analizador Sintáctico (Parser): Organiza los tokens según las reglas gramaticales.  
- Generador del Árbol de Sintaxis Abstracta (AST): Construye la representación jerárquica del programa.  
  
**Diagrama de Arquitectura:**

Fuente (.js)  
 │  
 ▼  
 [Lexer]  
 │ ───► Tokens  
 ▼  
 [Parser]  
 │ ───► Árbol Sintáctico Abstracto (AST)  
 ▼  
 [Errores]  
 │  
 ▼  
 Reporte Léxico y Sintáctico  
  
Cada módulo se comunica de forma secuencial. El Lexer entrega tokens al Parser, y este último produce el AST, registrando posibles errores para generar reportes finales.

## 3. Lenguaje de Programación a Compilar

El analizador se orienta al procesamiento de un código JavaScript, comprendiendo:  
  
- Declaraciones de variables (var, let, const).  
- Declaraciones de funciones sin parámetros.  
- Expresiones aritméticas básicas (+, -, \*, /, %).  
- Literales numéricos y de cadena.  
- Uso de paréntesis en expresiones.  
  
Lenguaje de implementación: Python   
Lenguaje fuente a analizar: Subconjunto de JavaScript

## 4. Clases y Métodos Desarrollados

El proyecto contiene tres módulos principales: lexer/lexer.py, parser/parser.py y main.py.  
  
Clases Principales:  
  
- Token: Define los elementos léxicos con atributos tipo, valor, línea, columna.  
- Lexer: Realiza el análisis léxico sobre el código fuente, generando una lista de tokens válidos o errores.  
- NodoAST: Representa los nodos del árbol sintáctico. Cada nodo tiene un tipo y puede contener hijos.  
- Parser: Implementa un analizador sintáctico de descenso recursivo que procesa los tokens y construye el AST.  
  
Métodos más relevantes:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Clase | Método | Descripción breve | | Lexer | tokenize() | Devuelve la lista de tokens a partir del texto fuente. | | Parser | parsear() | Punto de entrada del análisis sintáctico. | | Parser | parsear\_funcion() | Analiza la definición de funciones. | | Parser | parsear\_declaracion() | Analiza declaraciones de variables. | | Parser | parsear\_expresion() | Procesa expresiones aritméticas. | |

## 5. Reglas Sintácticas Mapeadas

A continuación se presentan las reglas EBNF mapeadas con las funciones del parser:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | N° | Regla EBNF | Función del Parser | | 1 | <Program> ::= { <Statement> } | parsear() | | 2 | <FunctionDeclaration> ::= 'function' <Identifier> '(' ')' <Block> | parsear\_funcion() | | 3 | <VariableDeclaration> ::= ('var' | 'let' | 'const') <Identifier> [ '=' <Expression> ] ';' | parsear\_declaracion() | | 4 | <Assignment> ::= <Addition> [ '=' <Assignment> ] | parsear\_asignacion() | | 5 | <Addition> ::= <Multiplicative> { ('+' | '-') <Multiplicative> } | parsear\_expresion() | |

## 6. Ejemplos de Entradas y Salidas

Ejemplo 1 — Código Correcto  
  
Entrada:  
var a = 3;  
var b = 4;  
var c = a + b;  
  
Salida:  
Tokens generados correctamente.  
Árbol Sintáctico Abstracto:  
  
Program  
 VariableDeclaration (var)  
 Identifier: a  
 Initializer  
 Number: 3  
 VariableDeclaration (var)  
 Identifier: b  
 Initializer  
 Number: 4  
 VariableDeclaration (var)  
 Identifier: c  
 Initializer  
 BinaryExpression (+)  
 Identifier: a  
 Identifier: b  
Errores: Ninguno.  
  
Ejemplo 2 — Error Sintáctico  
Entrada:  
var x = 2  
var y = (3 + ;  
Salida esperada:  
Error sintáctico (línea 1): se esperaba ';' al final de la declaración.  
Error sintáctico (línea 2): expresión incompleta tras '+'.  
  
Ejemplo 3 — Error Léxico  
Entrada:  
var c = a $ b;  
Salida esperada:  
Error léxico (línea 1, columna 10): carácter no reconocido '$'.

## 7. Código Fuente (Analizador Léxico y Sintáctico)

El proyecto incluye los siguientes módulos:  
  
main.py — Punto de entrada del programa. Crea instancias del Lexer y Parser, analiza el código y muestra los resultados.  
  
lexer/lexer.py — Define las clases Token y Lexer. Identifica los tipos de tokens mediante expresiones regulares y genera una lista estructurada con su posición en el texto fuente.  
  
parser/parser.py — Define las clases Parser y NodoAST. Implementa un analizador sintáctico descendente que reconoce estructuras gramaticales, construye el árbol AST y reporta errores con recuperación controlada.  
  
El parser emplea un enfoque modular donde cada regla EBNF corresponde a una función, facilitando la expansión del lenguaje.

## 8. Conclusiones

El sistema desarrollado cumple con los objetivos propuestos para el análisis léxico y sintáctico de un lenguaje estructurado.  
Entre los logros más importantes destacan:  
  
- Tokenización precisa mediante expresiones regulares.  
- Implementación de un parser de descenso recursivo.  
- Construcción de un Árbol Sintáctico Abstracto legible.  
- Reporte detallado de errores con línea y columna.