

Universidad Rafael Landívar  
Facultad de Ingeniería  
Lenguajes Formales y Automatas  
Sección: 1



# Manual de Usuario

## “Analizador Sintactico - Java”

Axel Guillermo Alvarado Taracena

[1284724]

Guatemala, viernes 24 de octubre del 2025

## Contenido

Introducción.....	3
Instalación .....	4
Ejemplos .....	7
Gramática Utilizada: .....	9
Explicacion de Modulos .....	10

# Introducción

El presente proyecto “Analizador Sintáctico – Java”, tiene como propósito el diseño e implementación de un sistema capaz de analizar la estructura sintáctica de programas escritos en un subconjunto del lenguaje Java, permitiendo validar su correcta formación de acuerdo con una gramática LL(1) previamente definida.

Este software constituye una herramienta académica que simula el funcionamiento de un parser predictivo descendente no recursivo, tomando como entrada un archivo de texto con código Java y produciendo como salida diferentes reportes que describen su proceso de análisis: los tokens detectados, los errores sintácticos encontrados, la tabla de transición generada automáticamente y el árbol de derivación del programa.

El sistema está desarrollado en Python, utilizando un entorno modular que separa claramente las fases de análisis léxico, sintáctico y de visualización. Además, incorpora una interfaz gráfica basada en Flask, que facilita la interacción del usuario con el analizador y la visualización de resultados de manera intuitiva.

Gracias a la integración de Graphviz, el usuario puede visualizar gráficamente la estructura jerárquica del árbol de derivación, lo cual favorece la comprensión del proceso de análisis sintáctico y la forma en que el parser interpreta las producciones de la gramática.

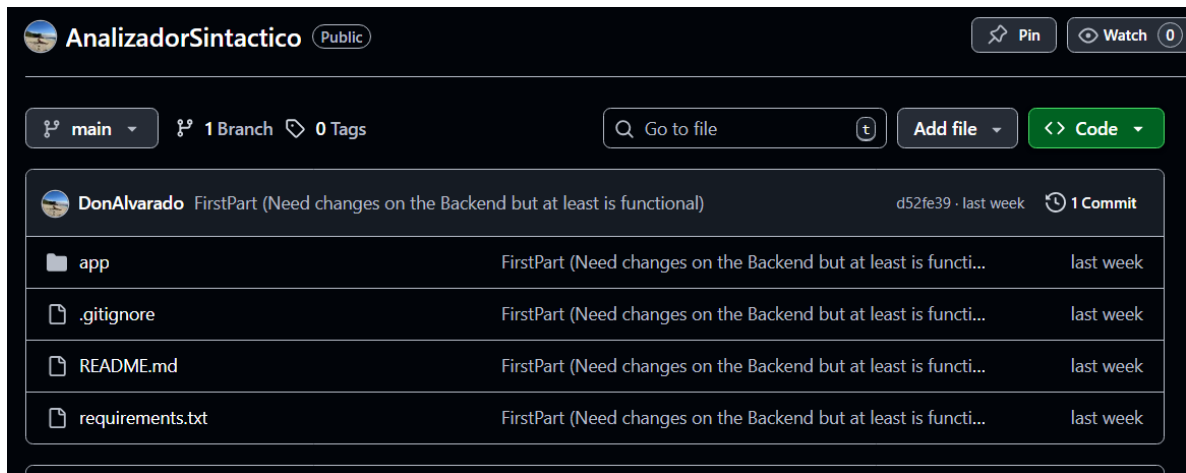
En conjunto, el proyecto no solo cumple con los requerimientos técnicos del curso de Lenguajes Formales y Autómatas, sino que también representa una aplicación práctica de los conceptos de gramáticas, autómatas, y parsers predictivos dentro de un entorno de desarrollo real, combinando teoría computacional con programación aplicada.

# Instalación

Para poder instalar el proyecto tiene que seguir una serie de pasos para poder completar la instalación y configuración del entorno virtual para poder trabajar con dicho proyecto.

## | Descarga del repositorio |

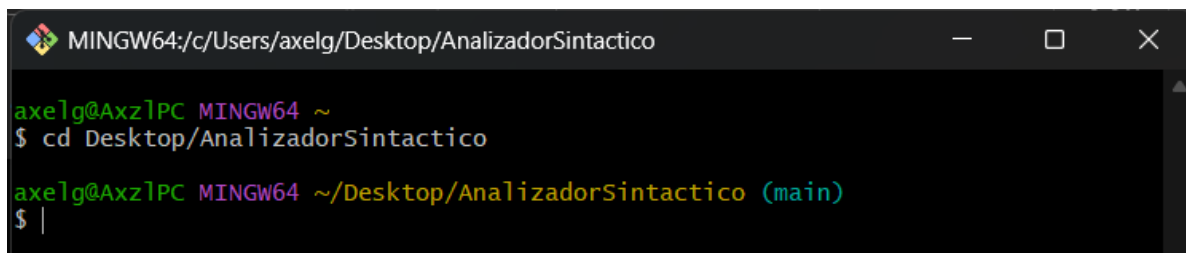
Como primer paso diríjase a el repositorio de Github en donde se encuentra el proyecto completo. <https://github.com/DonAlvarado/AnalizadorSintactico>.



Para poder realizar una instalación limpia coloque los siguientes comandos desde la terminal de Git Bash:

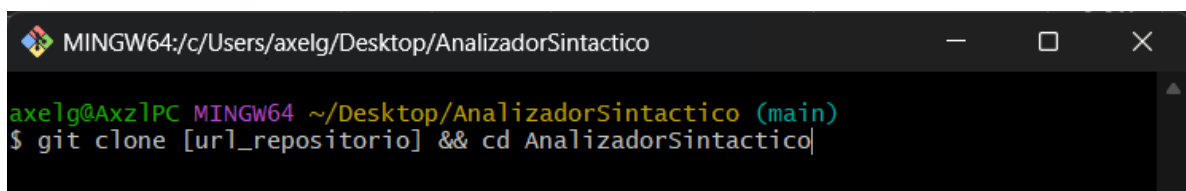
## | Elegir el directorio donde va a descargar el repositorio |

```
cd desktop/MiCarpeta
```



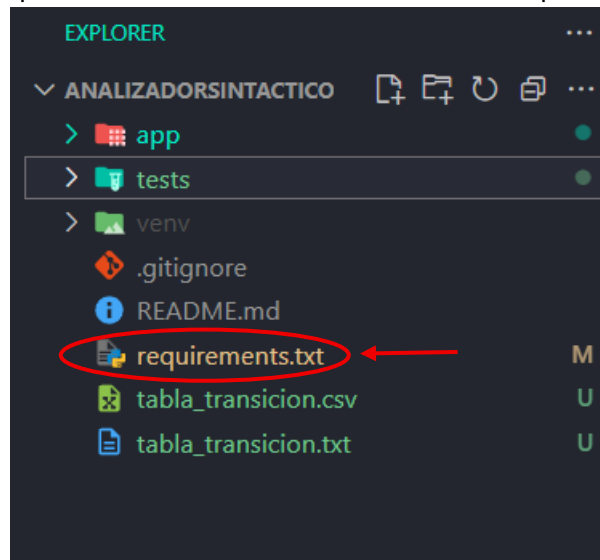
## | Hacer un clone del repositorio |

```
git clone https://github.com/DonAlvarado/AnalizadorSintactico.git  
cd AnalizadorSintactico
```



## | Abrir la Carpeta desde VS Code |

Usted al abrir la carpeta en VS Code va a encontrar varias carpetas parecidas a estas:



Como recomendación antes de tocar cualquier archivo o carpeta, es preferible que cree un entorno virtual con Python con el comando:

```
python -m venv venv
```

(Para activarlo):

```
.\venv\Scripts\Activate.ps1
```

Seguido de eso, usted tiene que realizar una instalación de los requerimientos del proyecto para que funcione, con el siguiente comando usted podrá usar mayor parte del proyecto:

```
pip install -r requirements.txt
```

Al Ejecutar el siguiente comando, usted estará instalando las librerías utilizadas en el proyecto directamente en el entorno virtual para que su maquina de trabajo no sufra con temas de compatibilidad en posibles proyectos mas adelante con las versiones de librerías.

## | Descarga e Instalacion de GraphViz |

Desde el navegador usted va a acceder a <https://graphviz.org/download/> y va a descargar la versión indicada de su Sistema Operativo y va a instalarla en su dispositivo, esta es la aplicación para poder tener un manejo completo sobre el árbol de derivación del analizador sintactico (y para que lo pueda visualizar en el software)

## | Levantar el Entorno |

Para poder visualizar el software usted tiene que escribir desde la terminal de VS Code con el entorno virtual activado el siguiente comando:

```
python -m app.run
```

```
(venv) PS C:\Users\axelg\Desktop\AnalizadorSintactico> python -m app.run
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.31:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 103-858-039
```

Con eso se le desplegará una serie de direcciones http en donde si accede desde alguna de ellas, se le desplegará el navegador con el software en pantalla.

## | Utilizar el Analizador Sintactico |

Para su primer analisis usted se tiene que dirigir a la sección de “Análisis” en el software, en donde se despliega una caja de texto con un botón de “Analizar código”, luego de que usted inserte su código de java a analizar, apase el botón y se desplegaran los 3 puntos importantes del analizador sintactico.

### Análisis de código Java

```
public int sumar(int x, int y) {
    int r = x + y;
    return r;
}

public void ejecutar() {
    a = 10;
    b = 5;
    int resultado = sumar(a, b);
    return;
}
```

Analizar código

#### Tokens detectados

Lexema	Categoría	Línea
import	import	1
java	id	1
.	.	1
util	id	1
.	.	1
.	.	1

#### Errores

[Línea 1] Error sintáctico: token inesperado 'im'

#### Árbol de derivación

Prog

# Ejemplos

## | Entrada |

```
import java.util.*;

public class Calculadora {

    int a;

    int b;

    public int sumar(int x, int y) {

        int r = x + y;

        return r;

    }

    public void ejecutar() {

        a = 10;

        b = 5;

        int resultado = sumar(a, b);

        return;

    }

}
```

## | Salida |

### Errores:

**Sin errores léxicos o sintácticos.**

## Arbol de Derivacion:



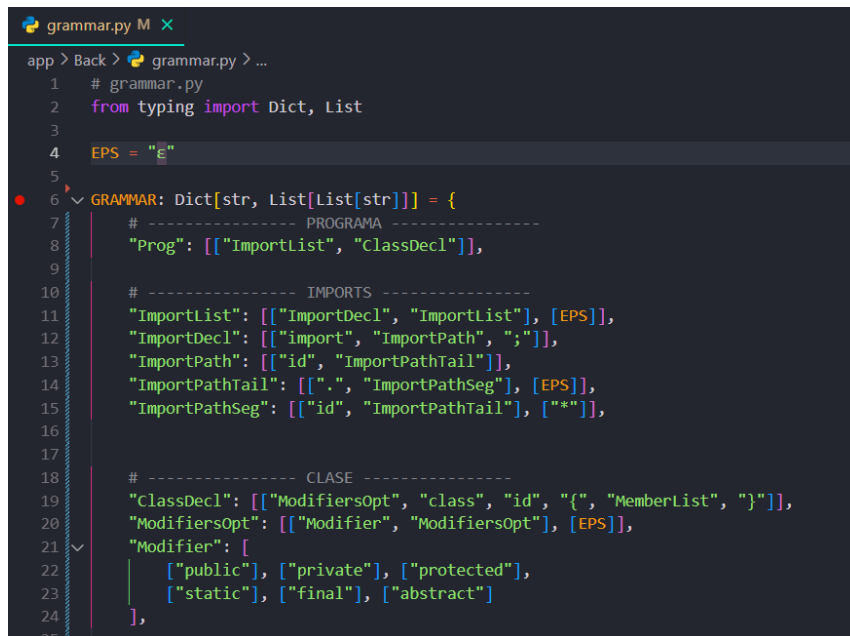


## Gramática Utilizada:

La gramática utilizada en el proyecto está basada en un subconjunto del lenguaje Java, cuidadosamente factorizada y libre de recursividad izquierda, con el fin de hacerla compatible con un analizador sintáctico predictivo LL(1).

El objetivo principal de esta gramática es definir la estructura formal que el parser utiliza para validar el código fuente ingresado. Cada producción describe cómo se pueden construir las partes válidas de un programa en Java: declaraciones de clases, variables, métodos, sentencias, expresiones y tipos de datos.

(Puede Visualizarlo en `grammar.py` en el repositorio)



```
grammar.py M X
app > Back > grammar.py > ...
1 # grammar.py
2 from typing import Dict, List
3
4 EPS = "ε"
5
6 GRAMMAR: Dict[str, List[List[str]]] = {
7     # ----- PROGRAMA -----
8     "Prog": ["ImportList", "ClassDecl"],
9
10    # ----- IMPORTS -----
11    "ImportList": ["ImportDecl", "ImportList", [EPS]],
12    "ImportDecl": ["import", "ImportPath", ";"],
13    "ImportPath": ["id", "ImportPathTail"],
14    "ImportPathTail": [".", "ImportPathSeg", [EPS]],
15    "ImportPathSeg": ["id", "ImportPathTail", ["*"]],
16
17    # ----- CLASE -----
18    "ClassDecl": ["ModifiersOpt", "class", "id", "(", "MemberList", ")"],
19    "ModifiersOpt": ["Modifier", "ModifiersOpt", [EPS]],
20    "Modifier": [
21        ["public", "private", "protected"],
22        ["static", "final", "abstract"]
23    ],
24 }
25
```

## Explicacion de Modulos

El sistema fue diseñado bajo una arquitectura modular que separa claramente cada fase del análisis. Esto facilita la depuración, el mantenimiento y la comprensión del flujo completo del analizador.

<i><b>Módulo / Archivo</b></i>	<i><b>Descripción Funcional</b></i>
<i><b>lexer.py</b></i>	Implementa el <b>análisis léxico</b> , encargado de leer el código fuente y dividirlo en <b>tokens</b> (palabras clave, identificadores, números, operadores, símbolos, etc.). También detecta y reporta errores léxicos, como caracteres ilegales.
<i><b>grammar.py</b></i>	Contiene la <b>definición formal de la gramática LL(1)</b> usada por el parser. Aquí se especifican las producciones y los símbolos terminales y no terminales del lenguaje.
<i><b>parser_generator.py</b></i>	Genera de manera automática la <b>tabla de transición LL(1)</b> a partir de la gramática definida. Si existen conflictos (como ambigüedad o falta de factorización), los muestra en consola.
<i><b>parser.py</b></i>	Implementa el <b>análisis sintáctico predictivo no recursivo</b> . Utiliza la tabla de transición y la pila de análisis para verificar si la secuencia de tokens cumple con la gramática. También genera los reportes de errores sintácticos.
<i><b>semantic.py</b></i>	Realiza una <b>clasificación básica de identificadores</b> (variables, métodos, operadores y símbolos), apoyando la fase de análisis semántico inicial.
<i><b>tree_viz.py</b></i>	Genera la estructura del <b>árbol de derivación</b> del programa y lo exporta en formato .dot para su visualización con Graphviz.
<i><b>table_gen.py</b></i>	Administra la construcción y exportación del archivo tabla_transicion.txt que contiene la tabla LL(1) completa generada.
<i><b>tokenizer.py</b></i>	Intermediario entre el lexer y el parser; organiza los tokens en estructuras manejables para el análisis sintáctico.
<i><b>routes.py</b></i>	Define las <b>rutas Flask</b> del sistema web, conectando el backend (análisis) con la interfaz gráfica del usuario.
<i><b>__init__.py y run.py</b></i>	Permiten inicializar la aplicación Flask y ejecutar el entorno web desde la terminal con python -m app.run.
<i><b>Carpeta Front/</b></i>	Contiene los <b>recursos visuales y plantillas HTML/CSS</b> que conforman la interfaz de usuario. Incluye los archivos index.html y ProjectDisplay.html junto con los estilos (cover.css, dashboard.css).

