Generación de árbol de sintaxis abstracta gráficos para el análisis de programas de complejidad intermedia, mediante el lenguaje de programación en Python

José Pedro Zamudio Baeza[1], Antonio Bautizta Juan[2], Joel Tapia Flores
[1] Departamento de Estudios Multidisciplinarios Sede Yuriria, Universidad de Guanajuato, Guanajuato, México. jp.zamudiobaeza@ugto.mx
[2] Instituto Tecnológico Superior del Sur de Guanajuato (ITSUR), Uriangato, Guanajuato, México.

Resumen

La enseñanza de la programación en etapas tempranas enfrenta importantes retos pedagógicos debido a la dificultad de los estudiantes para asimilar estructuras de código de forma abstracta, lo que suele derivar en frustración y deserción. Los Árboles de Sintaxis Abstracta (AST) ofrecen un mapeo jerárquico robusto de la lógica de un programa; no obstante, tradicionalmente se generan en formatos de texto crudo orientados a compiladores, resultando inaccesible para programadores principiantes. Este trabajo propone el desarrollo de una herramienta de diagnóstico que analiza código fuente escrito en Python para transformar su estructura en una representación gráfica e interactiva. Al contrastar el árbol visual generado por el estudiante contra el AST de una solución de referencia ideal, el sistema permite aislar errores estructurales y lógicos de manera automatizada. Este enfoque visual disminuye la carga cognitiva, transformando la depuración manual en una experiencia didáctica intuitiva que facilita la corrección de fallos y fortalece el pensamiento computacional.

Palabras clave: Árboles de Sintaxis Abstracta; pensamiento computacional; visualización de software; análisis estático.

Introducción

El desarrollo de habilidades de programación en etapas tempranas presenta deficiencias significativas debido a la falta de enfoques pedagógicos que faciliten la asimilación de conceptos abstractos (Tapia Flores *et al*., 2026). La depuración de código ha dependido tradicionalmente de una revisión manual exhaustiva del texto plano, lo cual implica un desgaste considerable de tiempo y genera frustración en los estudiantes, promoviendo la deserción en asignaturas tecnológicas. Una alternativa tecnológica para optimizar este proceso son los Árboles de Sintaxis Abstracta (AST), los cuales proporcionan una radiografía jerárquica y organizada de la estructura lógica de un programa. Sin embargo, los AST suelen generarse como complejas estructuras de datos crudos diseñadas para ser leídas por compiladores, lo que limita su aprovechamiento didáctico al carecer de un formato visual accesible para usuarios en formación.

En el estado del arte, el uso de AST se ha consolidado en la ingeniería de software para tareas complejas de reingeniería, como la detección de clones de código mediante la eliminación de subexpresiones comunes (Baxter *et al*., 1998). Asimismo, se han empleado en el rastreo de la evolución del software emparejando árboles de sintaxis de diferentes versiones (Neamtiu *et al*., 2005), y en la inspección directa del código fuente a través de lenguajes de consulta declarativos como ASTLOG (Crew, 1997). En el ámbito de la educación, tendencias internacionales respaldan que la visualización gráfica de estructuras de software y la simulación animada mejoran drásticamente el razonamiento abstracto y la motivación de los estudiantes (Nicolalde Rodríguez y Urquiza Fuentes, 2017). A pesar del alto desarrollo de estas herramientas analíticas, la aplicación de los AST como un recurso visual directo e interactivo centrado en la cognición de programadores principiantes permanece subdesarrollada.

Para abordar esta problemática, el presente trabajo tiene como objetivo general evaluar el nivel de comprensión lógica de los estudiantes al resolver problemas de programación, mediante la generación y comparación de AST gráficos contra una solución de referencia. La hipótesis principal plantea que la representación visual del código fuente en Python mediante AST proporciona una estructura jerárquica que optimiza el análisis lógico de soluciones de programación. Al generar estos modelos y contrastarlos contra una solución ideal, es posible aislar errores estructurales y cuantificar la integridad de los algoritmos, ofreciendo una métrica que disminuye la complejidad del análisis visual y fortalece el aprendizaje significativo.

.

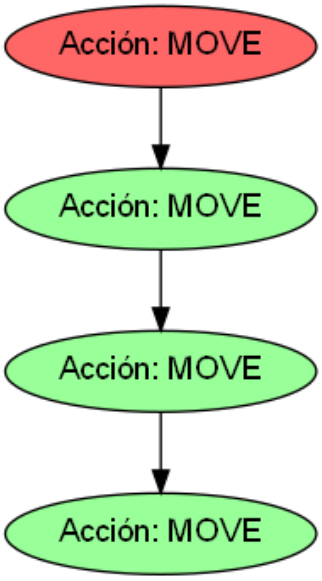


Figura 1. Árbol de sintaxis abstracta gráfico.
Fuente: Autoría propia.

Metodología Propuesta

El desarrollo de esta herramienta computacional se fundamenta en la extracción, procesamiento y visualización de Árboles de Sintaxis Abstracta (AST, por sus siglas en inglés) a partir de estructuras de datos serializadas. Para garantizar la replicabilidad y precisión del experimento, la metodología se estructuró en tres fases secuenciales: Ingesta de datos, Modelado lógico del árbol, y Generación del grafo dirigido.

Sección 1: Arquitectura de Ingesta y Preprocesamiento de Datos

El sistema recibe como entrada un archivo de texto plano estructurado en formato JSON, el cual contiene la representación jerárquica del código o los bloques de programación. El motor lógico fue diseñado en Python para deserializar estas secuencias, transformándolas en diccionarios en memoria. Esta estructura permite iterar sobre los metadatos de los nodos (tales como etiquetas XML, bloques lógicos y punteros de continuidad funcional como next o children).

Sección 2: Algoritmo Recursivo de Extracción y Recorrido

Para la reconstrucción lógica del árbol, se implementó un algoritmo de recorrido en profundidad (Depth-First Search o DFS) adaptado para el análisis de sintaxis.

Sea T la estructura de datos del árbol deserializado, el sistema implementa una función recursiva $f(n, p, \epsilon)$ donde:

- n representa el nodo actual en evaluación.
- p representa el identificador único del nodo padre.
- $\epsilon \in \{0,1\}$ es una variable booleana que indica si n es el nodo raíz (inicio del árbol).

El algoritmo evalúa condicionalmente la etiqueta topológica del nodo. Si el nodo corresponde a estructuras de agrupamiento estructural (etiquetas que contienen secuencias como xml o next), el algoritmo transfiere el flujo a los nodos descendientes (almacenados en la clave children).

Cuando el algoritmo identifica un nodo funcional (etiqueta block), procede a la extracción de atributos sintácticos. Para evitar colisiones en la memoria al generar el modelo gráfico, a cada nodo se le asigna un identificador único absoluto derivado de su dirección de memoria en tiempo de ejecución, denotado computacionalmente como $ID(n) = \text{str}(\text{id}(n))$.

Sección 3: Modelado Topológico y Generación de Grafos

La visualización del AST se formalizó utilizando los principios de la teoría de grafos, empleando la biblioteca Graphviz. El sistema construye un grafo dirigido $G = (V, E)$, donde:

- V es el conjunto de vértices (nodos lógicos del código).
- E es el conjunto de aristas dirigidas que representan la relación jerárquica padre-hijo.

Para cada vértice $v \in V$, el sistema genera una etiqueta visual compuesta por el atributo type (tipo de acción). Se implementó una lógica de semántica de color para facilitar la legibilidad espacial del árbol visual:

- Si $\epsilon = 1$ (Nodo raíz), el vértice asume una tonalidad de alerta (código hexadecimal #ff6666).
- Si $\epsilon = 0$ (Nodos secundarios), el vértice asume una tonalidad de flujo normal (código hexadecimal #99ff99).

La relación jerárquica E se establece generando una arista direccional desde el vértice padre $ID(p)$ hacia el vértice actual $ID(n)$.

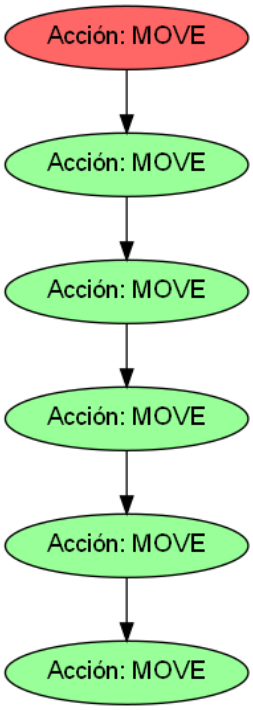
Sección 4: Interfaz de Usuario y Renderización de Salida

Para dotar a la metodología de un entorno de experimentación ágil, la arquitectura lógica fue encapsulada dentro de una Interfaz Gráfica de Usuario (GUI) construida sobre Tkinter. El flujo de operación experimental diseñado consta de los siguientes estados lógicos:

1. **Carga (Input):** Apertura segura y en validación de codificación (UTF-8) del archivo origen.
2. **Previsualización (Render-in-memory):** Ejecución del motor Graphviz para compilar el grafo G en memoria temporal, permitiendo al investigador auditar la estructura topológica del árbol.
3. **Exportación (Output):** Rasterización final del grafo lógico hacia una imagen gráfica en formato PNG, asegurando la preservación estática de las jerarquías extraídas para su análisis posterior.

Resultados

Los resultados obtenidos mediante la metodología empleada demuestran la transformación exitosa de los árboles de sintaxis abstracta (AST) desde su formato de texto estructurado (JSON) hacia una representación gráfica. Esta traducción visual resulta fundamental para reducir la carga cognitiva, facilitando a los usuarios una mejor comprensión de la lógica computacional oculta en el código y agilizando la resolución de problemas o detección de anomalías lógicas.



En la figura anterior se presenta un AST renderizado directamente a partir de su archivo JSON original, correspondiente a un bloque de instrucciones básicas de desplazamiento. En este caso específico, mostramos el AST gráfico de un problema de ejecución lineal, cuyo nodo principal o raíz está destacado en la parte superior con un relleno de color rojo. Este nodo denota el punto de entrada del flujo de ejecución y extrae del JSON la etiqueta "Acción: MOVE".

A partir de este vértice inicial, el algoritmo de visualización despliega una estructura estrictamente secuencial y descendente. Mediante aristas dirigidas (flechas), el árbol conecta en cascada a cinco nodos secundarios identificados con el color verde, todos etiquetados de igual forma con la instrucción "Acción: MOVE".

La disposición visual de este grafo describe y confirma de manera explícita que el código de origen no contiene ramificaciones condicionales ni ciclos anidados; por el contrario, representa un flujo de control lineal ininterrumpido. Cada paso lógico transfiere el estado de ejecución directamente a la siguiente instrucción de movimiento, permitiendo al observador trazar la trayectoria completa del bloque de código con un solo vistazo.

Referencias

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., y Bier, L. (1998). Clone detection using abstract syntax trees. *Proceedings International Conference on Software Maintenance,* 368–377, IEEE. 10.1109/ICSM.1998.738528

Crew, R. F. (1997). ASTLOG: A language for examining abstract syntax trees. En *Proceedings of the Conference on Domain-Specific Languages (DSL'97)* (pp. 229–242). USENIX Association.http://usenix.org/publications/library/proceedings/dsl97/full_papers/crew/crew.pdf

Neamtiu, I., Foster, J. S., y Hicks, M. (2005). *Understanding Source Code Evolution Using Abstract Syntax Tree Matching. SIGSOFT Softw. Eng. Notes,* 30, 4, 1–5. https://doi.org/10.1145/1082983.1083143

Nicolalde Rodríguez, D. A., y Urquiza Fuentes, J. (2017). Revisión de herramientas de visualización de la Traducción Dirigida por Sintaxis. *IE Comunicaciones*, *Revista*

Iberoamericana de Informática Educativa, (26), 24-37. https://dialnet.unirioja.es/servlet/articulo?codigo=6231881

Tapia Flores, J., Avila Garcia, M. S., y Lopez Ramirez, M. (2026). Recommending computational thinking learning paths using fuzzy logic. *International Journal of Combinatorial Optimization Problems and Informatics, 17*(1), 307-317. https://doi.org/10.61467/2007.1558.2026.v17i1.1170