Proyecto Final: Fase 2

Nombres:

- **Frank Esteban Soto Paz**
- **Santiago Velandia Gallo**

Presentado a: Ana María Tamayo Ocampo

Universidad del Quindío

Facultad Ingeniería

Programa Ingeniería de Sistemas y Computación

Curso Teoría de Lenguajes Formales

Grupo Noche

Resumen

El proyecto consiste en el desarrollo de un analizador léxico y sintáctico para un lenguaje ficticio en el contexto de la asignatura Teoría de Lenguajes Formales de Ingeniería de Sistemas y Computación. Utilizando la biblioteca PLY (Python Lex-Yacc) en Python, se implementa un analizador que reconoce tokens y estructuras sintácticas del lenguaje definido, así como la visualización de diagramas AFN y AFD, y la construcción y visualización de árboles de derivación. La interfaz gráfica proporcionada mediante PyQt5 permite al usuario cargar archivos de código fuente y realizar análisis léxicos y sintácticos, generando un reporte visual del proceso.

Tabla de Contenidos

Descripción del Proyecto

El proyecto aborda la implementación de un analizador léxico y sintáctico como parte de la asignatura de Teoría de Lenguajes Formales. Utilizando Python y la biblioteca PLY, se construye un analizador capaz de reconocer tokens y estructuras sintácticas de un lenguaje ficticio definido para el proyecto. El análisis léxico se encarga de identificar elementos como operadores aritméticos, palabras reservadas, identificadores y valores, mientras que el análisis sintáctico se encarga de verificar la estructura gramatical del código fuente.

Una vez analizado el código, el proyecto ofrece la visualización de diagramas AFN y AFD que representan los estados y transiciones del autómata finito no determinista y determinista respectivamente. Además, se construyen y visualizan árboles de derivación que muestran la estructura jerárquica del

código fuente. Todo esto se realiza mediante el uso de herramientas como Graphviz y Pydot para la generación de gráficos y la interfaz gráfica proporcionada por PyQt5 para la interacción con el usuario.

El proyecto se presenta como una herramienta educativa y práctica para comprender los conceptos de la teoría de lenguajes formales y su aplicación en el diseño y desarrollo de analizadores léxicos y sintácticos. Permite a los estudiantes experimentar con la construcción de autómatas y árboles de derivación, así como con el uso de herramientas de visualización para comprender mejor el proceso de análisis de código fuente.

Análisis Léxico

Ply (Python Lex-Yacc):

> PLY es una implementación en Python de las herramientas Lex y Yacc. Permite construir el analizador léxico y sintáctico de un lenguaje.

Funcionalidad del Analizador Léxico

El analizador léxico define las expresiones regulares para los tokens del lenguaje. Este módulo reconoce una serie de tokens que representan operadores aritméticos, relacionales, lógicos, símbolos de apertura y cierre, palabras reservadas, identificadores y valores. Utiliza PLY para convertir una cadena de entrada en una secuencia de tokens.

Análisis Sintáctico

El analizador sintáctico define las reglas gramaticales para reconocer y procesar asignaciones y expresiones aritméticas, generando un árbol de derivación basado en la estructura del código de entrada.

Diagramas AFN y AFD

1. **Construcción del AFN (Autómata Finito No Determinista):**

    - Se define un AFN en función de los tokens reconocidos por el analizador léxico.
    - Cada estado del AFN representa una etapa en el análisis del código fuente.

- Las transiciones entre estados están determinadas por los tokens reconocidos en el código fuente.

2. **Construcción del AFD (Autómata Finito Determinista):**

    - El AFD se construye a partir de la entrada del usuario (el código fuente).
    - Se utiliza el analizador léxico para reconocer los tokens en el código.
    - Cada estado del AFD representa una etapa específica en el análisis del código.
    - Las transiciones entre estados están determinadas por los tokens reconocidos en el código.

Visualización de los Diagramas AFN y AFD:

1. **Visualización del AFN:**

    - Utilizamos la biblioteca Graphviz para generar una representación gráfica del AFN.
    - Cada estado se representa como un nodo en el gráfico.
    - Las transiciones entre estados se muestran como bordes etiquetados con los tokens correspondientes.

2. **Visualización del AFD:**

    - Al igual que con el AFN, utilizamos Graphviz para generar el diagrama del AFD.
    - Cada estado se representa como un nodo en el gráfico, con estados finales marcados de manera distintiva.
    - Las transiciones entre estados se muestran como bordes etiquetados con los tokens correspondientes.

Cómo se Diferenciarían los AFD y AFN:

- **AFD (Autómata Finito Determinista)**:
    - Cada estado tiene una única transición para cada símbolo del alfabeto.
    - Ejemplo: Si el estado actual es **q0** y el token es **NEWVAL**, la transición siempre será hacia un estado específico, digamos **q1**.

- **AFN (Autómata Finito No Determinista)**:

- Un estado puede tener múltiples transiciones para el mismo símbolo del alfabeto.
- Ejemplo: Desde el estado **q0** con el token **NEWVAL**, podría haber transiciones a múltiples estados, digamos **q1** y **q2**.

Funcionalidad de Scroll:

- Para manejar imágenes grandes, implementamos funcionalidad de scroll utilizando el widget **QScrollArea** de PyQt5.
- Esto permite ver las imágenes de los diagramas, incluso si son más grandes que la ventana.
- Cuando se muestra una imagen, se coloca dentro de un **QScrollArea**, lo que permite desplazarse si la imagen es demasiado grande para caber completamente en la ventana.

Implementación de prueba

```
NEWVAL := 10
IFTHIS NEWVAL == 10 && NEWVAL != 5 | OTHERWISE NEWVAL >= 5 || NEWVAL < 20
NEWVAL := NEWVAL * 2
NEWVAL := NEWVAL - 5
NEWVAL := NEWVAL / 2
NEWVAL := NEWVAL % 3
LOOPFOR NEWVAL := NEWVAL + 1
```

Explicación del Ejemplo de prueba:

1. **Asignación Inicial**:
   - **NEWVAL := 10**
   - Esto inicia el valor de **NEWVAL** a 10.

2. **Condicional Complejo**:
   - **IFTHIS NEWVAL == 10 && NEWVAL != 5 | OTHERWISE NEWVAL >= 5 || NEWVAL < 20**
   - Este condicional verifica varias condiciones lógicas:
     - ✓ Si **NEWVAL** es 10 y no es 5, o si **NEWVAL** es mayor o igual a 5 o menor que 20.

3. **Operaciones Aritméticas**:
   - **NEWVAL := NEWVAL * 2**
   - **NEWVAL := NEWVAL - 5**
   - **NEWVAL := NEWVAL / 2**
   - **NEWVAL := NEWVAL % 3**

- Estas líneas realizan varias operaciones aritméticas sobre **NEWVAL**.

4. **Bucle**:
    - **LOOPFOR NEWVAL := NEWVAL + 1**
    - Un bucle que incrementa **NEWVAL** en 1 (la estructura exacta y repetición del bucle dependerá de las reglas completas del lenguaje).

Este ejemplo es adecuado porque contiene diversas construcciones del lenguaje, incluyendo asignaciones, operaciones aritméticas, condicionales y un bucle, permitiendo así la generación de gráficos que muestren cómo cada estructura es manejada por un AFD y un AFN.

Construcción y Visualización del Árbol de Derivación

Graphviz y Pydot

Para visualizar los árboles de derivación, se utilizan Graphviz junto con Pydot en Python.

Tkinter

Se utiliza Tkinter para crear una interfaz gráfica simple que permita visualizar directamente el árbol de derivación en una ventana de Python.

- **TreeNode**: Usa una clase **TreeNode** para construir el árbol de derivación.
- **Generación de PNG**: Genera un archivo PNG del árbol de derivación utilizando Pydot y Graphviz.
- **Visualización**: Muestra el árbol de derivación en una ventana gráfica utilizando Tkinter y Pillow.

Requisitos e Instalación

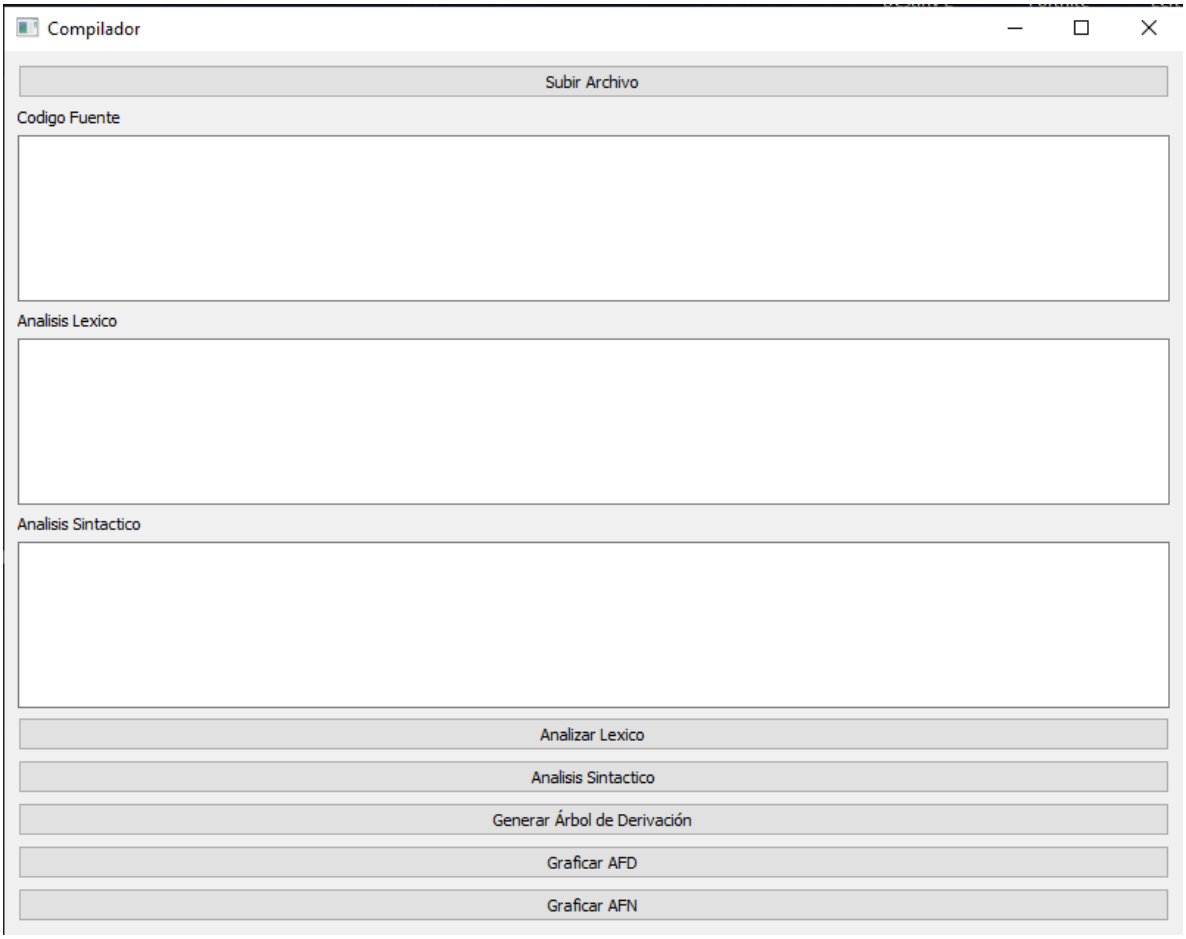
Para ejecutar este proyecto, asegúrate de importarlo, tener Python instalado y luego instala las bibliotecas.

1. **PLY (Python Lex-Yacc)**: PLY es una herramienta poderosa que te permite construir analizadores léxicos y sintácticos en Python. Es una implementación de las herramientas Lex y Yacc, que son estándares en la construcción de compiladores.

2. **Graphviz**: Graphviz es una herramienta de visualización de gráficos que te permite representar gráficamente los diagramas AFN y AFD generados durante el análisis léxico y sintáctico.
3. **PyQt5**: PyQt5 es un conjunto de herramientas que te permite crear aplicaciones de escritorio con interfaces gráficas de usuario (GUI) en Python. En tu proyecto, lo has utilizado para crear la interfaz gráfica que permite visualizar los árboles de derivación y otros resultados.
4. **Pillow**: Pillow es una biblioteca de procesamiento de imágenes en Python que utilizas para generar archivos PNG de los árboles de derivación y otras imágenes.
5. **tokens** (Archivo propio): Este archivo contiene los tokens generados por el analizador léxico y es esencial para el funcionamiento del analizador sintáctico. Define los elementos básicos que constituyen el lenguaje que estás analizando.
6. **my_parser** (Archivo propio): Este archivo contiene el analizador sintáctico generado por PLY a partir de las reglas gramaticales que has definido para tu lenguaje. Es responsable de analizar la estructura del código fuente y generar un árbol de derivación.
7. **tree** (Archivo propio): Este archivo contiene funciones para construir y dibujar árboles de derivación, lo que te permite visualizar la estructura del código fuente de manera gráfica.
8. **graph_helpers** (Archivo propio): Este archivo contiene funciones para graficar los diagramas AFN y AFD, lo que te ayuda a visualizar los estados y las transiciones del autómata.

Uso

Para ejecutar el proyecto, simplemente ejecuta el script principal: python main.py



En la vista, selecciona el archivo que leerá el código fuente (por ejemplo, **prueba.txt**). Luego, utiliza los botones para realizar el análisis léxico y sintáctico, y genera un reporte en formato HTML con el árbol de derivación visualizado.

Categorías y Tokens

Categoría	Subcategoría	Tokens
Operadores Aritméticos		PS (Suma), MS (Resta), TS (Multiplicación), DB (División), PW (Potencia), SQ (Raíz), MOD (Módulo)
Operadores Relacionales		EQ (Igualdad), GTOE (Mayor o igual), LTOE (Menor o igual), GT (Mayor), LT (Menor), EQNOT (Desigual)

Categoría	Subcategoría	Tokens
Operadores Lógicos		IAI (And), IOI (Or), INI (Not)
Operadores de Asignación		:= (Igual)
Símbolos de Apertura		(, {, ¿
Símbolos de Cierre		), }, ?
Terminal y/o Inicial		BREVE
Separadores de Sentencias		
Palabras Reservadas	Bucle o ciclo	LOOPFOR, WHILEFOR
	Decisión	IFTHIS, OTHERWISE
	Clase	ICLASSI, INTI, IENUMI
Identificadores	Variable	NEWVAL
	Método	NEWFUNC
Valor de Asignación	Enteros	+(0,..., 9)
	Reales	+(0,..., 9) . (0,..., 9)
	Cadenas de caracteres	"(a,...,z,A,...,Z,0,...,9,!, ", #, $, %, &, ', (,), *, +, , , ., /, : , ; , <, =, >, ?, @, [, ,], ^, _, `, {,

Categoría	Subcategoría	Tokens
	Caracteres	`(a,...,z,A,...,Z,0,...,9,!, ", #, $, %, &, ', ‟, (, ), *, +, , , ., /, : , ; , <, =, >, ?, @, [, , ], ^, _, `, {,
Tipo de Dato	Enteros	NEWINT
	Reales	NEWNUM
	Cadenas de caracteres	NEWEXT
	Caracteres	NEWCHAR
Comentarios	De línea	B {texto aquí}
	De bloque	B {texto aquí}
Hexadecimal		H(A,...,F,0,...,9 +)

Conclusiones y Reflexiones

En este trabajo final para la asignatura Teoría de Lenguajes Formales de Ingeniería de Sistemas y Computación en la Universidad del Quindío, hemos explorado en profundidad la implementación de un analizador léxico y sintáctico para un lenguaje ficticio utilizando herramientas como PLY (Python Lex-Yacc), Graphviz, Pydot y Tkinter.

1. **Importancia de la Teoría de Lenguajes Formales en la Ingeniería de Sistemas y Computación**

   La Teoría de Lenguajes Formales proporciona las bases fundamentales para el diseño y desarrollo de herramientas de procesamiento de lenguajes de programación, como compiladores e intérpretes. Este conocimiento es esencial en el campo de la ingeniería de sistemas y computación, ya que permite comprender y manipular el comportamiento de los lenguajes de programación de manera efectiva.

2. **Aplicabilidad Práctica de los Conceptos Teóricos**

La implementación de un analizador léxico y sintáctico para un lenguaje ficticio demuestra la aplicabilidad práctica de los conceptos teóricos de la teoría de lenguajes formales. Utilizando herramientas como PLY, pudimos definir reglas gramaticales precisas y generar un análisis del código fuente que facilita su comprensión y procesamiento.

3. **Flexibilidad y Potencia de las Herramientas de Análisis Léxico y Sintáctico**

   Hemos podido experimentar con la flexibilidad y potencia de herramientas como PLY, que permiten a los desarrolladores definir y personalizar reglas gramaticales y patrones de tokens según las necesidades específicas de un lenguaje de programación. Esta capacidad es crucial para el desarrollo de sistemas complejos que requieren un análisis preciso del código fuente.

4. **Visualización y Comunicación de Resultados**

   La visualización de los diagramas AFN, AFD y árboles de derivación mediante herramientas como Graphviz y Pydot facilita la comunicación y comprensión de los resultados del análisis léxico y sintáctico. Estas representaciones gráficas son útiles tanto para los desarrolladores que implementan el analizador como para aquellos que utilizan el lenguaje en cuestión.

5. **Desafíos y Consideraciones de Implementación**

   Reconocemos los desafíos inherentes a la implementación de un analizador léxico y sintáctico, como la gestión de grandes conjuntos de reglas gramaticales y la optimización del rendimiento del análisis. También destacamos consideraciones prácticas, como la instalación y configuración de herramientas externas como Graphviz, que son necesarias para la generación de diagramas.

Bibliografía

1. Pillow Documentation. (n.d.). Retrieved from https://python-pillow.org/
2. Graphviz Documentation. (n.d.). Retrieved from https://graphviz.org/documentation/
3. Python Software Foundation. (n.d.). PLY (Python Lex-Yacc). Retrieved from https://github.com/dabeaz/ply