Universidad Autónoma De Tamaulipas

Facultad De Ingeniería Tampico

PROGRAMACION DE SISTEMAS DE BASE II

2025 – 3 Otoño

Profesor: ***Dante Adolfo Muñoz Quintero***

MANUAL DE USUARIO Y TÉCNICO

Nombre del alumno	Matricula
Falcon Alejandre Marcos Israel	**2223330152**
Aquino Reyes Jorge Alfredo	**2213332133**
Espinosa Arguelles Cesar Eduardo	**2173270075**

SEMESTRE	GRUPO	HORA
9	I	L-J 19:00 - 20:00

Introducción

Propósito del proyecto

El propósito de este proyecto es desarrollar una aplicación de escritorio que funcione como traductor y visor de documentos en formato Markdown, convirtiéndolos a HTML mediante una gramática definida (ANTLR) y lógica en Python. El sistema busca Aplicar los conceptos de análisis léxico y sintácticos vistos en la materia (parseo del Markdown). Generar y visualizar el árbol de sintaxis (AST) del documento para entender su estructura. Permitir al usuario escribir, cargar y guardar archivos Markdown, y ver el HTML resultante de forma clara.

Alcance

El alcance de este proyecto se limita a la traducción de un **subconjunto** de las características del lenguaje Markdown hacia HTML, así como a la visualización básica del resultado y del árbol de sintaxis. En particular, el traductor soporta:

- Encabezados de nivel H1 a H6.
- Párrafos y saltos de línea.
- Formatos de texto: **negritas**, *cursiva* y ~~tachado~~.
- Listas ordenadas y no ordenadas con un solo nivel.
- Enlaces (links) en línea.
- Bloques de cita (blockquotes).
- Tablas estándar (sin combinaciones complejas de celdas).
- Reglas horizontales.

Adicionalmente, el sistema contempla:

- Procesar archivos de texto plano con extensión .md almacenados de forma local.
- Mostrar el **AST** generado a partir del análisis sintáctico.
- Generar el **HTML** correspondiente y presentarlo al usuario en un área de salida o archivo de resultado.
- Reportar **errores léxicos y sintácticos** cuando el documento no cumpla con la gramática definida.

Limitaciones del alcance

- En el caso de las listas, **no se soportan listas anidadas o sublistas**. solo se maneja un nivel de viñetas o numeración. Esto se debe a que Markdown depende fuertemente del

contexto y el manejo correcto de estructuras anidadas incrementa considerablemente la complejidad del análisis, lo cual queda fuera del alcance de este proyecto.
- El traductor solo soporta las características de Markdown mencionadas en la lista anterior. Cualquier otro elemento de Markdown se considera fuera del alcance del proyecto.
- La conversión es **unidireccional**: de Markdown a HTML. No se contempla la edición directa sobre el HTML ni la conversión inversa.
- No se implementa un sistema avanzado de estilos (CSS); la apariencia final del documento HTML dependerá del navegador o visor utilizado.

Instalación y configuración

Requisitos del sistema

Para ejecutar el traductor y su interfaz gráfica es necesario contar con:

- Sistema operativo Windows 10 o superior.
- Python 3.8 o superior instalado y agregado a la variable de entorno PATH.
- Java Runtime Environment (JRE) (solo necesario para generar los archivos de ANTLR a partir de la gramática).

Dependencias y librerías

El proyecto requiere las siguientes librerías de Python las cuales son las siguientes:

- antlr4-python3-runtime
- PyQt5

Se pueden instalar ejecutando el siguiente comando en la terminal, dentro de la carpeta del proyecto:

```
pip install antlr4-python3-runtime PyQt5_
```

Compilación de la gramática

Antes de ejecutar el programa, es necesario generar los analizadores léxico y sintáctico a partir de la gramática de Markdown.

Ubíquese en la carpeta src del proyecto y ejecute:

```
antlr4 -Dlanguage=Python3 -visitor MarkdownGrammar.g4_
```

Este comando generará los archivos **MarkdownGrammarLexer.py**, **MarkdownGrammarParser.py** y **MarkdownGrammarVisitor.py**, que serán utilizados por la aplicación.

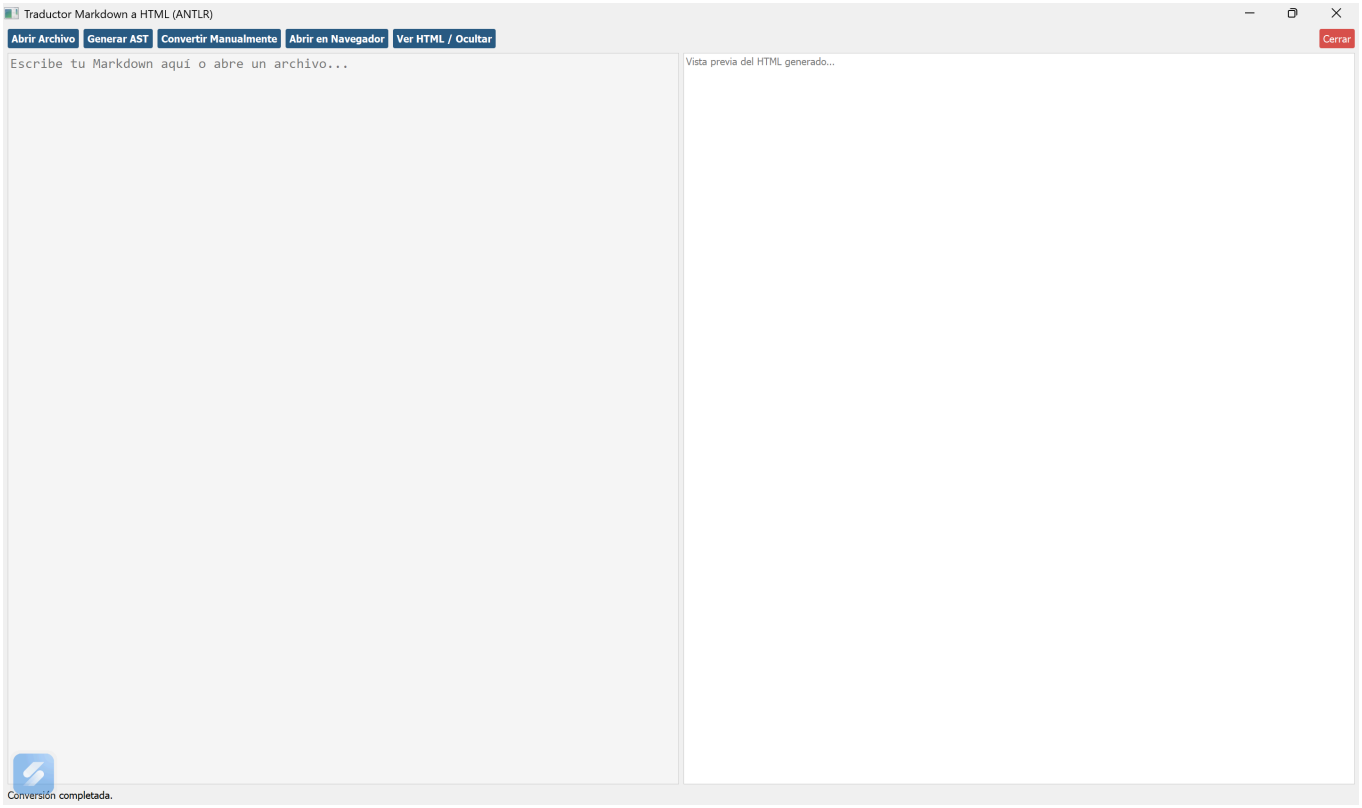
Guía de uso

Interfaz gráfica (Main.py)

La aplicación ofrece un entorno visual intuitivo para la traducción en tiempo real de documentos en formato Markdown a HTML. Desde esta interfaz el usuario puede escribir, abrir, convertir y visualizar sus archivos sin necesidad de utilizar comandos en consola.

Componentes de la interfaz

- Editor de Markdown (izquierda): Área de entrada de texto donde el usuario puede escribir directamente o pegar contenido en formato Markdown. Aquí se redactan los encabezados, listas, tablas, citas y demás elementos que serán procesados por el traductor.
- Vista previa HTML (derecha): Panel de visualización que muestra el código HTML generado automáticamente a partir del contenido Markdown. Permite verificar en tiempo real cómo se verá el documento una vez convertido.
- Barra de herramientas:
  - Abrir archivo: Permite cargar un archivo con extensión .md desde el disco.
  - Generar AST: Muestra la estructura interna del análisis sintáctico (árbol de sintaxis) del documento, lo cual es útil para fines de depuración y para entender cómo la gramática interpreta el texto.
  - Abrir en navegador: Abre el resultado en el navegador web predeterminado (Chrome, Edge, etc.), mostrando el HTML final tal como lo vería un usuario en una página web.



Modo consola (Driver.py)

Además de la interfaz gráfica, la aplicación cuenta con un modo de uso por consola para realizar conversiones rápidas sin necesidad de abrir la ventana gráfica.

Para utilizarlo, se ejecuta el script Driver.py indicando el archivo de entrada y el archivo de salida:

```
python Driver.py archivo_entrada.md archivo_salida.html_
```

Donde:

- archivo_entrada.md es el archivo que contiene el texto en formato Markdown.
- archivo_salida.html es el nombre del archivo HTML que se generará con el resultado de la traducción.

Este modo es útil cuando se desea automatizar la conversión de varios archivos o integrarla en scripts y procesos de desarrollo.

Ejemplos

En esta sección se muestran ejemplos concretos de cómo la aplicación traduce el contenido de Markdown a HTML. Cada ejemplo incluye:

- Entrada (Markdown): lo que el usuario escribe.

- Salida (HTML): lo que genera la herramienta.

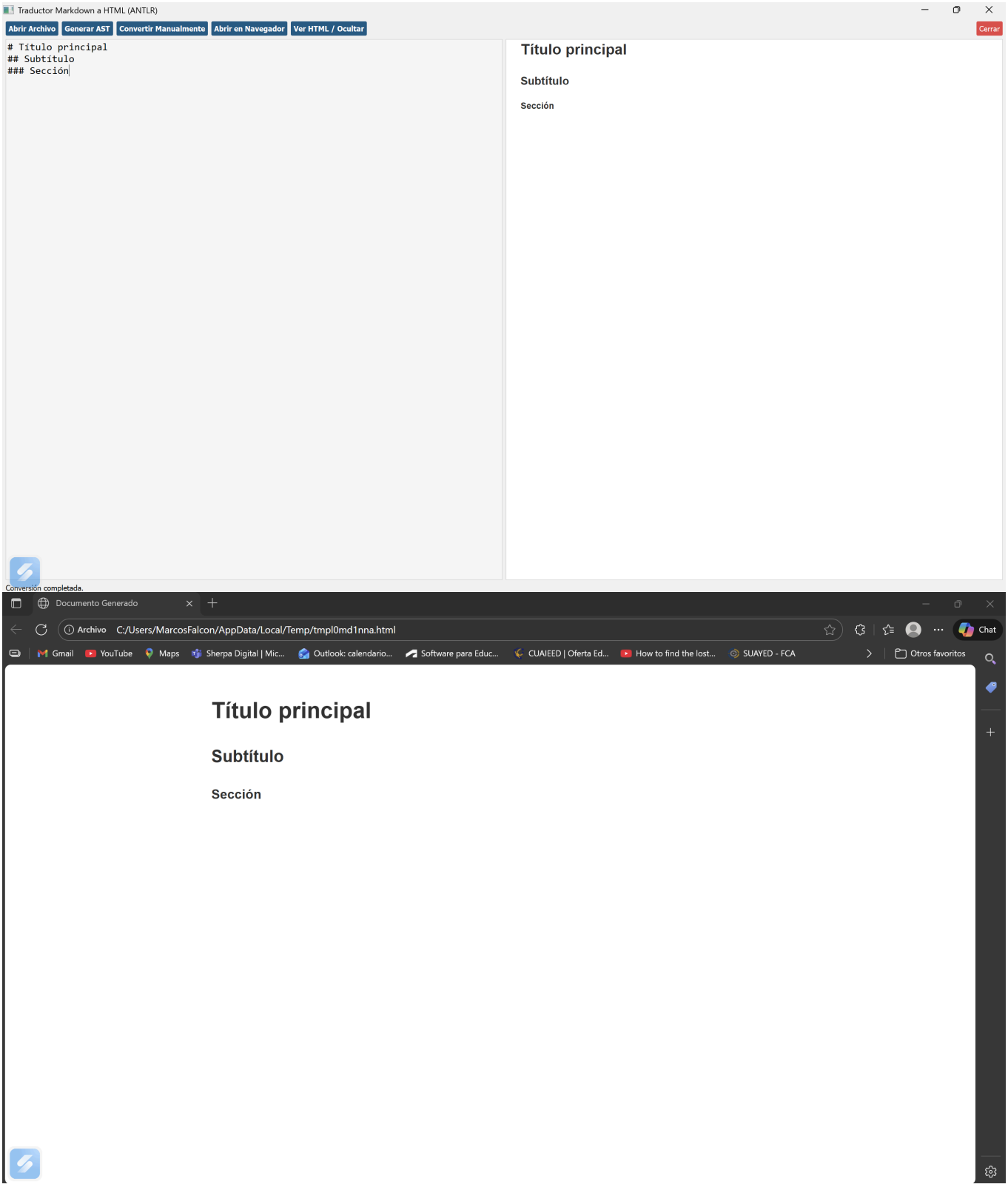
Ejemplo 1: Encabezados

Entrada

```
# Título principal
## Subtítulo
### Sección
```

Salida

```
<h1>Título principal</h1>
<h2>Subtítulo</h2>
<h3>Sección</h3>
```



Ejemplo 2: Listas

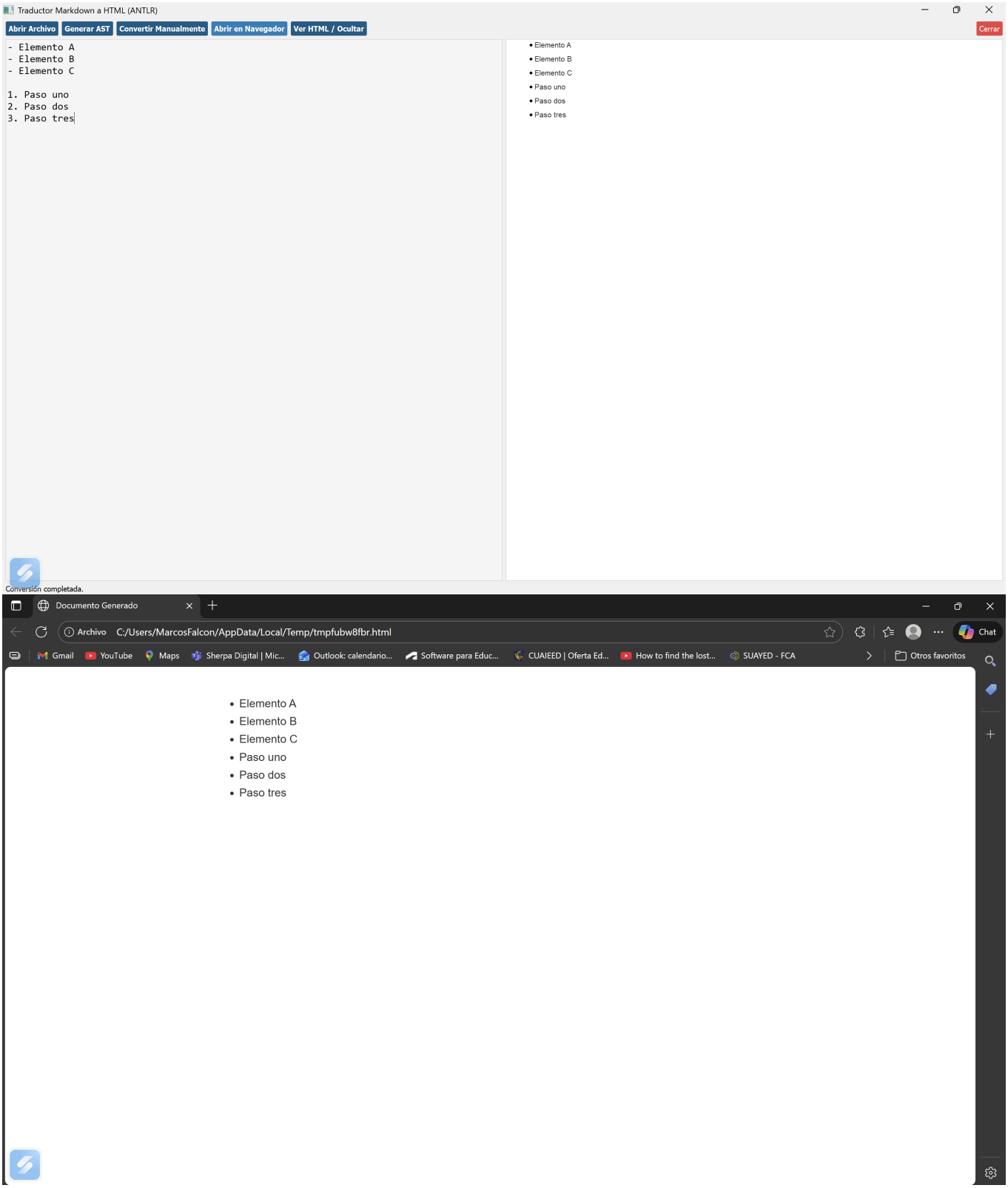
Entrada

```
- Elemento A
- Elemento B
- Elemento C
```

```
1. Paso uno
2. Paso dos
3. Paso tres
```

Salida

```
<ul>
  <li>Elemento A</li>
  <li>Elemento B</li>
  <li>Elemento C</li>
</ul>

<ol>
  <li>Paso uno</li>
  <li>Paso dos</li>
  <li>Paso tres</li>
</ol>
```



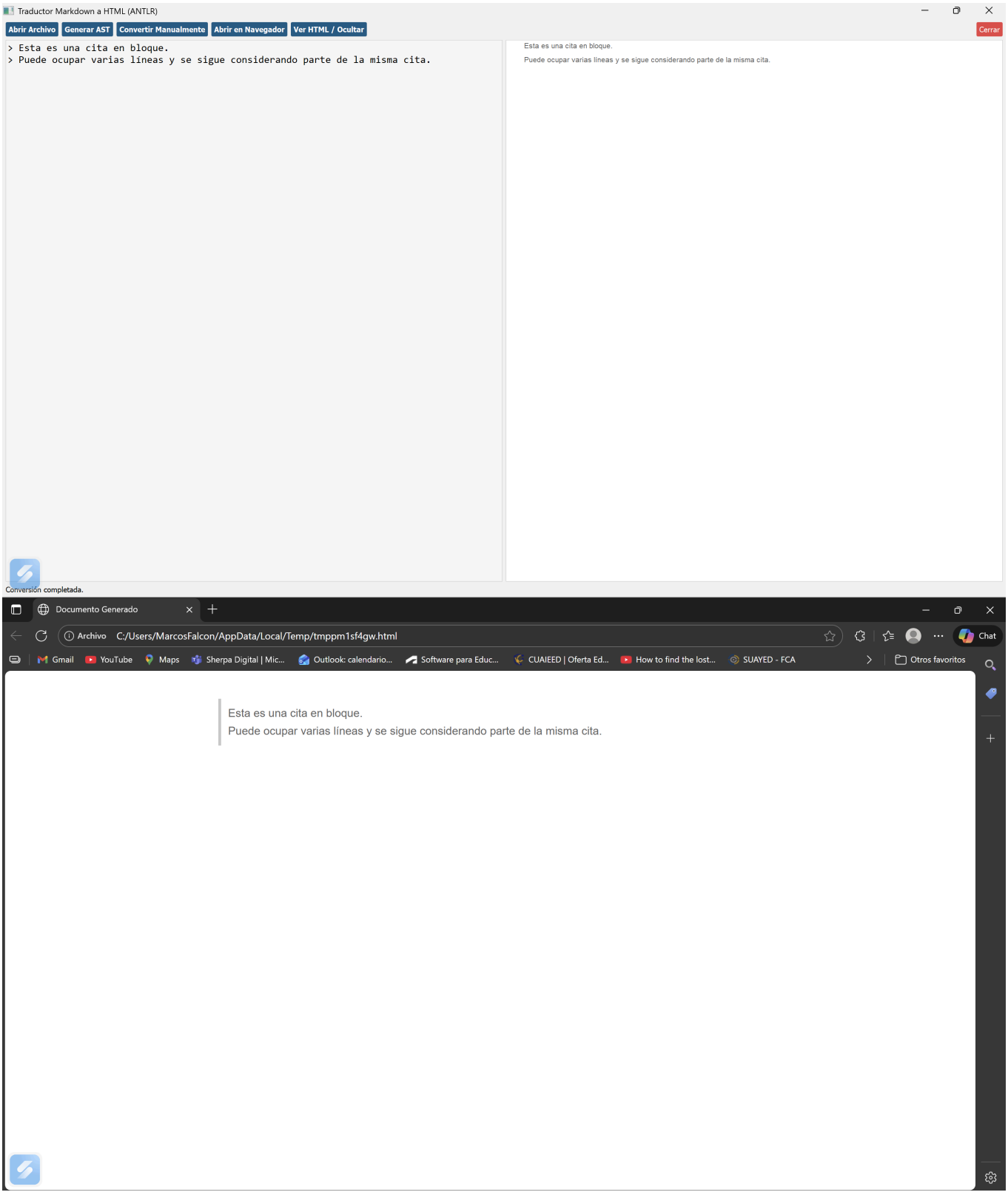
Ejemplo 3: Citas (blockquotes)

Entrada

```
> Esta es una cita en bloque.
> Puede ocupar varias líneas y se sigue considerando parte de la misma cita.
```

Salida

```
<blockquote>Esta es una cita en bloque.
Puede ocupar varias líneas y se sigue considerando parte de la misma cita.
</blockquote>
```



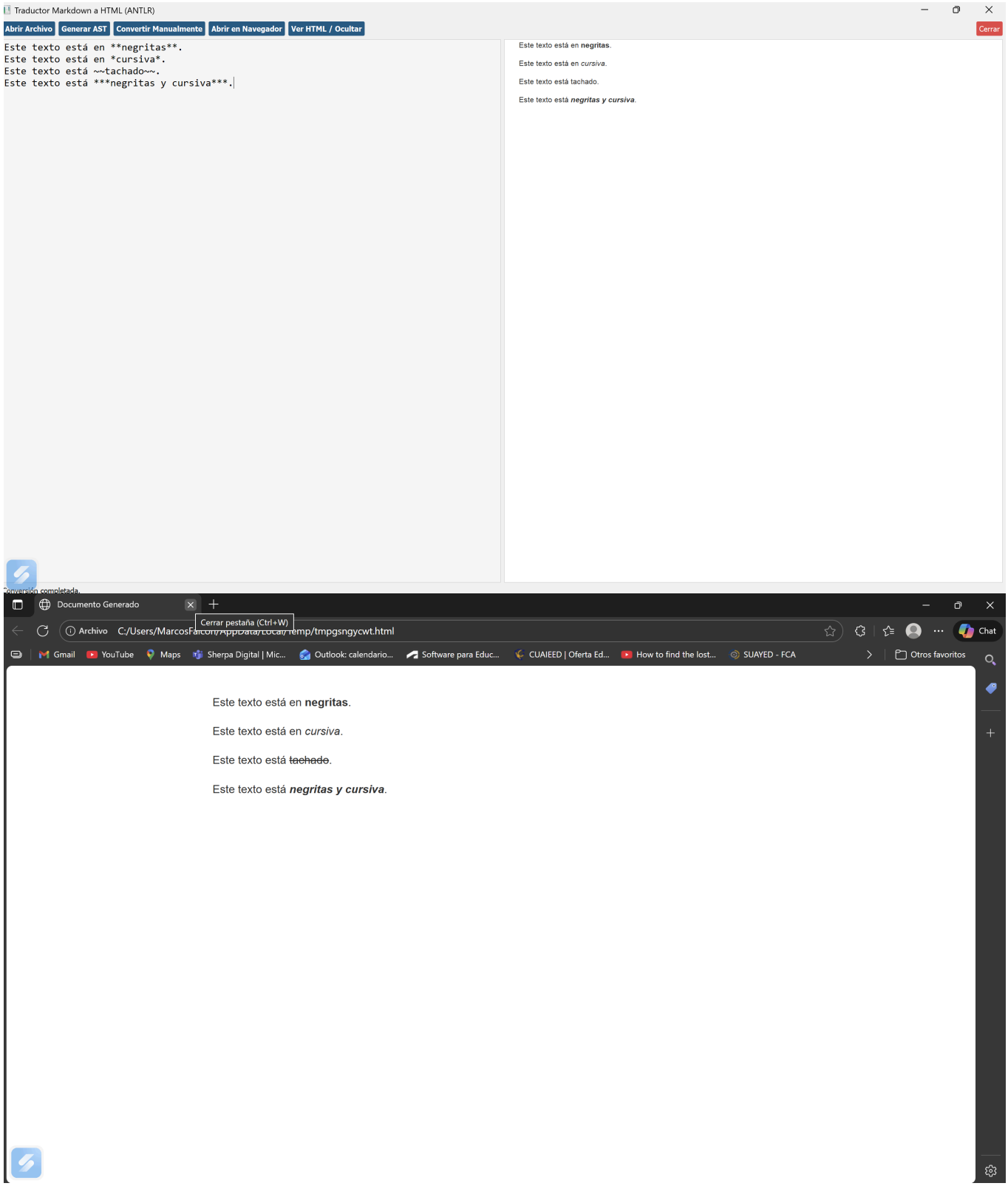
Ejemplo 4: Texto con formato

Entrada

```
Este texto está en **negritas**.
Este texto está en *cursiva*.
Este texto está ~~tachado~~.
Este texto está ***negritas y cursiva***.
```

Salida

```
<p>Este texto está en <strong>negritas</strong>.</p>
<p>Este texto está en <em>cursiva</em>.</p>
<p>Este texto está <del>tachado</del>.</p>
<p>Este texto está <strong><em>negritas y cursiva</em></strong>.</p>
```



Ejemplo 5: Tablas

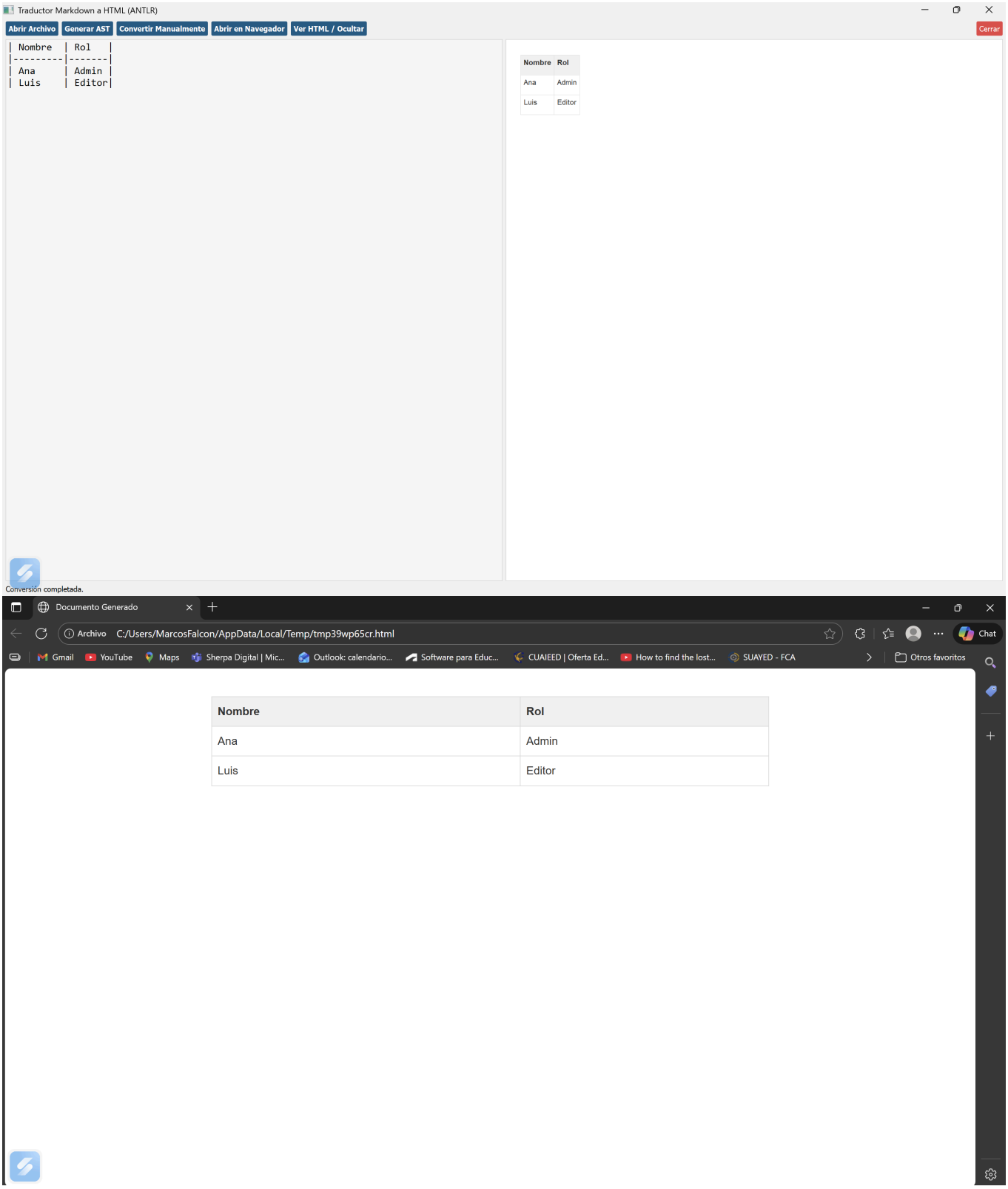
Entrada

```
| Nombre   | Rol    |
|----------|--------|
```

```
| Ana     | Admin |
| Luis    | Editor|
```

Salida

```
<table>
  <thead>
    <tr>
      <th>Nombre</th>
      <th>Rol</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Ana</td>
      <td>Admin</td>
    </tr>
    <tr>
      <td>Luis</td>
      <td>Editor</td>
    </tr>
  </tbody>
</table>
```



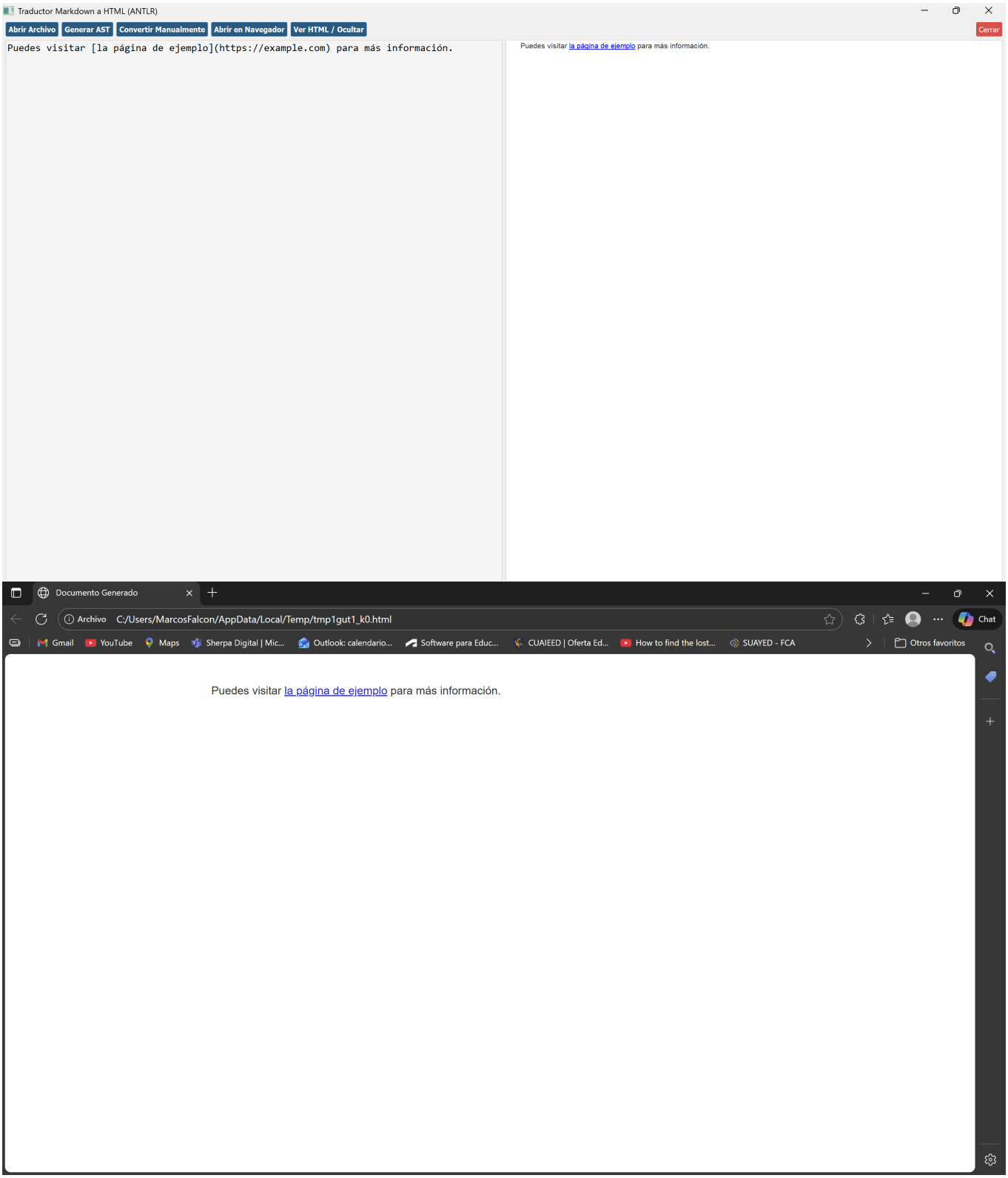
Ejemplo 6: Enlaces en línea

Entrada

```
Puedes visitar [la página de ejemplo](https://example.com) para más
información.
```

Salida

```
<p>Puedes visitar <a href="https://example.com">la página de ejemplo</a> para
más información.</p>
```



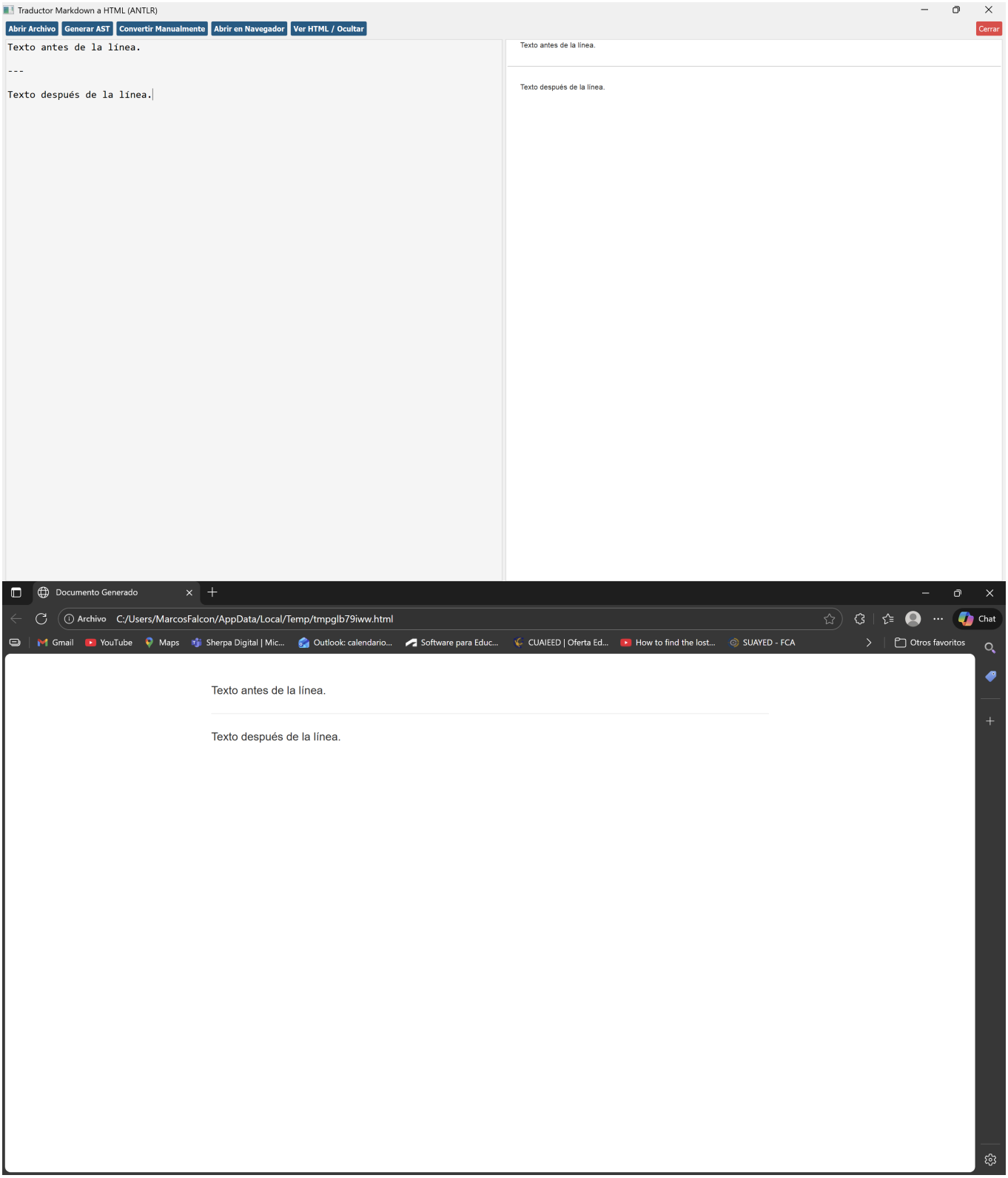
Ejemplo 7: Regla horizontal

Entrada

```
Texto antes de la línea.

---

Texto después de la línea.
```

Salida

```
<p>Texto antes de la línea.</p>
<hr/>
<p>Texto después de la línea.</p>
```



Ejemplo 8: Documento completo

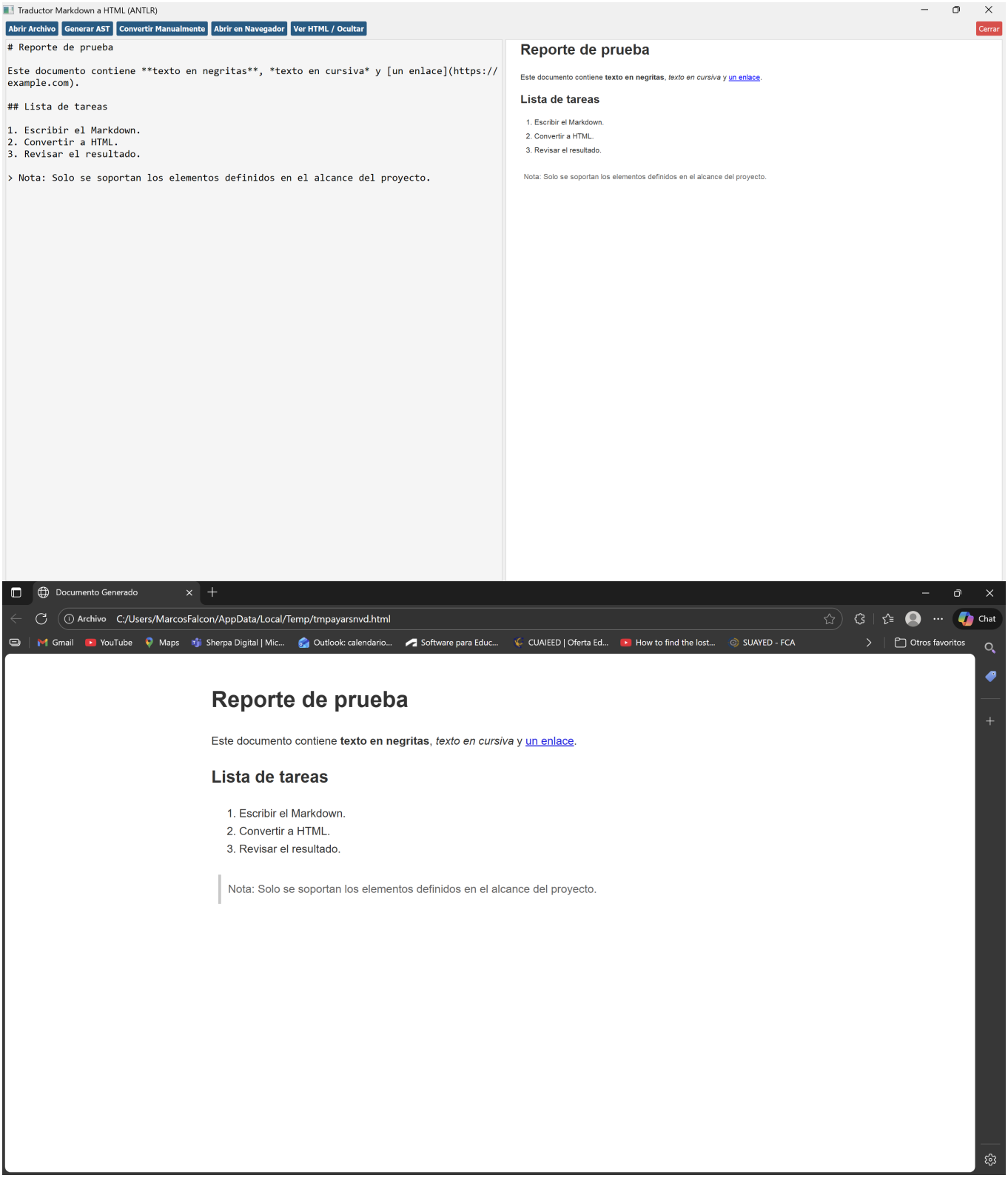
Entrada

```
# Reporte de prueba

Este documento contiene **texto en negritas**, *texto en cursiva* y [un
```

```
enlace](https://example.com).

## Lista de tareas

1. Escribir el Markdown.
2. Convertir a HTML.
3. Revisar el resultado.

> Nota: Solo se soportan los elementos definidos en el alcance del proyecto.
```

Salida

```
<h1>Reporte de prueba</h1>
<p>Este documento contiene <strong>texto en negritas</strong>, <em>texto en
cursiva</em> y <a href="https://example.com">un enlace</a>.</p>

<h2>Lista de tareas</h2>
<ol>
  <li>Escribir el Markdown.</li>
  <li>Convertir a HTML.</li>
  <li>Revisar el resultado.</li>
</ol>

<blockquote>Nota: Solo se soportan los elementos definidos en el alcance del
proyecto.</blockquote>
```



Referencia técnica

Arquitectura general del sistema

El traductor de Markdown a HTML está organizado como una aplicación de escritorio que utiliza una arquitectura por módulos. Cada módulo se encarga de una parte específica del proceso, desde la captura del texto hasta la generación del código HTML final.

De manera general, el sistema se puede dividir en tres capas principales:

1. Capa de presentación: Encargada de la interacción con el usuario. Incluye la interfaz gráfica desarrollada en Python (por ejemplo, con PyQt5) y, en su caso, el modo de ejecución por consola.
   - Ventana principal (Main.py).
   - Componentes visuales de edición y vista previa.
2. Capa de compilación (núcleo del traductor): Implementa la lógica de análisis léxico y sintáctico, así como la generación de HTML.
   - Gramática en ANTLR para Markdown.
   - Analizador léxico y sintáctico generados por ANTLR.
   - Visitor en Python que recorre el árbol y construye el HTML.
3. Capa de entrada/salida de archivos: Se encarga de leer archivos Markdown desde el sistema de archivos y de escribir los archivos HTML generados.
   - Apertura de archivos .md.
   - Guardado de resultados .html.

El flujo general es el siguiente:
El usuario escribe o abre un archivo Markdown → el texto se envía al compilador → el lexer y el parser construyen el árbol de análisis → el visitor recorre el árbol y genera el HTML → el resultado se muestra en pantalla y/o se guarda en un archivo.

Estructura de archivos del proyecto

La organización de los archivos puede variar según la implementación, pero normalmente se sigue una estructura similar a la siguiente:

- Main.py: Archivo principal de la interfaz gráfica. Contiene la ventana principal, la configuración de los botones, menús y áreas de texto.
- Driver.py: Script para la ejecución por consola. Recibe nombres de archivo por parámetros y realiza la conversión sin interfaz gráfica.
- MarkdownGrammar.g4: Archivo de gramática ANTLR que define la sintaxis de Markdown soportada por el proyecto (encabezados, listas, tablas, citas, etc.).
- Archivos generados por ANTLR (ejemplo):
  - MarkdownGrammarLexer.py
  - MarkdownGrammarParser.py
  - MarkdownGrammarVisitor.py
- MarkdownVisitor: la lógica que recorre el árbol generado por el parser y convierte cada nodo en etiquetas HTML.

Descripción de módulos principales

Módulo de interfaz gráfica (Main.py)

Este módulo implementa la ventana principal de la aplicación. Sus funciones principales son:

- Inicializar la aplicación de escritorio.
- Mostrar un área de texto donde el usuario puede escribir o pegar contenido en Markdown.
- Proporcionar botones o menús para:
  - Abrir un archivo .md desde el sistema de archivos.
  - lanzar el proceso de traducción a HTML.
  - Mostrar la salida HTML en un panel de vista previa.
  - Abrir el resultado en el navegador web.

Además, el módulo se encarga de la comunicación con la capa de compilación: toma el texto del editor y lo envía al analizador para obtener el resultado HTML, que posteriormente se muestra al usuario.

Módulo de consola (Driver.py)

El módulo de consola proporciona una forma alternativa de utilizar el traductor sin interfaz gráfica. Sus responsabilidades son:

- Leer los argumentos de la línea de comandos (archivo Markdown de entrada y archivo HTML de salida).
- Abrir y leer el contenido del archivo .md.
- Enviar el texto al lexer y parser generados por ANTLR.
- Invocar al visitor que genera el HTML.}
- Guardar el resultado HTML en el archivo especificado.

Este módulo es útil para automatizar conversiones, integrarlo en scripts o realizar pruebas rápidas sin necesidad de abrir la interfaz.

Módulo de gramática y análisis (MarkdownGrammar.g4 y archivos generados)

La gramática MarkdownGrammar.g4 define las reglas del lenguaje Markdown que el proyecto reconoce. Entre estas reglas se incluyen las correspondientes a:

- · Encabezados de nivel 1 a 6.
- · Listas ordenadas y no ordenadas.
- · Tablas (cabecera, separadores y filas de datos).
- · Citas en bloque.
- · Texto con formato (negritas, cursivas, combinaciones, tachado, subrayado).

- · Enlaces en línea.
- · Párrafos y saltos de línea.

A partir de este archivo, ANTLR genera automáticamente el analizador léxico (Lexer), el analizador sintáctico (Parser) y una clase base para el visitor. Estos componentes se encargan de transformar el texto de entrada en un árbol de análisis estructurado.

Módulo de generación de HTML (Visitor en Python)

El visitor es la pieza central de la conversión. Para cada regla importante de la gramática se define un método en el que se indica cómo traducir la estructura correspondiente a HTML.

El visitor recorre el árbol de análisis en orden lógico y va concatenando el HTML resultado. Al final del recorrido se obtiene el documento completo en HTML.

Manejo de errores y validación

El sistema contempla el manejo de errores que pueden surgir durante el análisis del archivo Markdown. Algunos ejemplos de situaciones a considerar son:

- Uso de estructuras de Markdown no contempladas en la gramática.
- Anidamiento incorrecto de ciertos elementos.
- Líneas que no se ajustan a ninguna regla reconocida.

Cuando ocurre un error, el analizador puede:

- Mostrar mensajes en la consola o en la interfaz gráfica indicando el tipo de error.
- Traducirlo como parrafo html

De esta forma, el usuario y el desarrollador pueden identificar más fácilmente la causa de un fallo en la traducción.

Codigo fuente

MarkdownGrammar.g4

```
grammar MarkdownGrammar;

// --- PARSER ---

doc : (block | NEWLINE)* EOF;

block
    : horizontalRule
```

```
    | header
    | table
    | blockquote
    | list
    | paragraph
    ;

header : HASHES inlineContent? NEWLINE;
blockquote : blockquoteLine+;
blockquoteLine : GT inlineContent? NEWLINE;
list : listItem+ ;

listItem
    : ul_marker inlineContent NEWLINE        # unorderedItem
    | OL_NUM inlineContent NEWLINE           # orderedItem
    ;

ul_marker : STAR | PLUS | MINUS ;

horizontalRule : (HR_DASH | HR_STAR | HR_UNDER | BOLD_ITALIC_STAR) NEWLINE;

// --- TABLAS -
table : tableHeaderLine tableSepLine tableBodyLine*;

// Cabecera: Una fila normal
tableHeaderLine : tableRow;


tableSepLine : WS? PIPE (WS? (DASHES | HR_DASH | MINUS | DASHES_COLON |
COLON_DASHES | COLON_DASHES_COLON) WS? PIPE)+ WS? NEWLINE;

// Cuerpo: Filas normales
tableBodyLine : tableRow;

// Fila: | celda | celda |
tableRow : WS? PIPE (inlineItem_no_pipe* PIPE)+ WS? NEWLINE;

paragraph : inlineContent NEWLINE;

inlineContent : inlineItem+ ;

inlineItem
    : link
    | bold_italic
    | bold
    | italic
```

```
    | strikethrough
    | WS
    | INDENT
    | TEXT
    | DOT | COLON
    | STAR | PLUS | MINUS | HASHES | GT | OL_NUM | HR_DASH | HR_STAR |
HR_UNDER
    | PIPE | COLON_DASHES_COLON | DASHES_COLON | COLON_DASHES | DASHES
    | BOLD_ITALIC_STAR | BOLD_ITALIC_STAR_UNDER_START |
BOLD_ITALIC_STAR_UNDER_END
    | BOLD_ITALIC_UNDER_STAR_START | BOLD_ITALIC_UNDER_STAR_END
    | BOLD_STAR | BOLD_UNDER | STRIKE | UNDER
    | LBRACKET | RBRACKET | LPAREN | RPAREN
    | ANY_SYMBOL
    ;

inlineItem_no_pipe
    : link
    | bold
    | italic
    | strikethrough
    | WS
    | INDENT
    | TEXT
    | ANY_SYMBOL_NO_PIPE
    | DOT | COLON
    | STAR | PLUS | MINUS | HASHES | GT | OL_NUM | HR_DASH | HR_STAR |
HR_UNDER
    | BOLD_ITALIC_STAR | BOLD_ITALIC_STAR_UNDER_START |
BOLD_ITALIC_STAR_UNDER_END
    | BOLD_ITALIC_UNDER_STAR_START | BOLD_ITALIC_UNDER_STAR_END
    | BOLD_STAR | BOLD_UNDER | STRIKE | UNDER
    | LBRACKET | RBRACKET | LPAREN | RPAREN
    ;

bold_italic
    : (BOLD_ITALIC_STAR inlineContent BOLD_ITALIC_STAR)
    | (BOLD_ITALIC_STAR_UNDER_START inlineContent BOLD_ITALIC_STAR_UNDER_END)
    | (BOLD_ITALIC_UNDER_STAR_START inlineContent BOLD_ITALIC_UNDER_STAR_END)
    ;
bold
    : (BOLD_STAR inlineContent BOLD_STAR)
    | (BOLD_UNDER inlineContent BOLD_UNDER)
    ;
italic
    : (STAR inlineContent STAR)
```

```
    | (UNDER inlineContent UNDER)
    ;
strikethrough : STRIKE inlineContent STRIKE;
link : LBRACKET inlineContent RBRACKET LPAREN URL RPAREN;

// --- LEXER ---

NEWLINE  : [\r\n]+ ;
HASHES   : '#' + ;
GT       : '>' ;
INDENT   : [ \t]{2,} ;

BOLD_ITALIC_STAR: '***';
BOLD_ITALIC_STAR_UNDER_START: '**_';
BOLD_ITALIC_STAR_UNDER_END: '_**';
BOLD_ITALIC_UNDER_STAR_START: '__*';
BOLD_ITALIC_UNDER_STAR_END: '*__';
BOLD_STAR  : '**' ;
BOLD_UNDER : '__' ;
STRIKE     : '~~' ;

HR_DASH  : '---' '-'* ;
HR_STAR  : '****' '*'* ;
HR_UNDER : '___' '_'* ;

STAR      : '*' ;
PLUS      : '+' ;
MINUS     : '-' ;
UNDER     : '_' ;
DOT       : '.' ;
COLON     : ':' ;

OL_NUM    : [0-9]+ DOT ;

PIPE     : '|' ;
COLON_DASHES_COLON : ':' '-' '-'* ':' ;
DASHES_COLON       : '-' '-'* ':' ;
COLON_DASHES       : ':' '-' '-'* ;
DASHES             : '-' '-'* ;

LBRACKET : '[' ;
RBRACKET : ']' ;
LPAREN   : '(' ;
RPAREN   : ')' ;

URL      : 'http' [a-zA-Z0-9:/?#[\]@!$&'*+,;=.\-_%]+ ;
```

```
WS : [ \t]+ ;

TEXT : ~[#*+\-_>~[\]():|.\r\n]+ ;

ANY_SYMBOL_NO_PIPE : ~[|\r\n] ;
ANY_SYMBOL : . ;
```

MarkdownVisitor.py

```
from __future__ import annotations

from typing import List
from antlr4 import ParserRuleContext
from antlr4.tree.Tree import TerminalNode

from MarkdownGrammarParser import MarkdownGrammarParser
from MarkdownGrammarVisitor import MarkdownGrammarVisitor


class MarkdownToHtmlVisitor(MarkdownGrammarVisitor):
    """Convierte el parse tree de Markdown a HTML mediante el patrón
visitor."""

    def visitDoc(self, ctx: MarkdownGrammarParser.DocContext) -> str:
        html_parts: List[str] = []
        for block in ctx.block():
            piece = self.visit(block)
            if piece:
                html_parts.append(piece)

        body_content = ''.join(html_parts)

        return f"""<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">    <meta name="viewport" content="width=device-
width, initial-scale=1.0">    <title>Documento Generado</title>    <style>
body {{
            font-family: Arial, sans-serif;            line-height: 1.6;
margin: 40px auto;            max-width: 800px;            padding: 0 10px;
color: #333;        }}
        blockquote {{
            border-left: 4px solid #ccc;            margin: 1.5em 10px;
padding: 0.5em 10px;            color: #666;        }}
```

```
        table {{
            border-collapse: collapse;            width: 100%;
margin-bottom: 1em;        }}
        th, td {{
            border: 1px solid #ddd;            padding: 8px;            text-
align: left;        }}
        th {{
            background-color: #f2f2f2;        }}
        code {{
            background-color: #f4f4f4;            padding: 2px 4px;
border-radius: 4px;        }}
        hr {{
            border: 0;            border-top: 1px solid #eee;
margin: 20px 0;        }}
    </style></head>
<body>
{body_content}
</body>
</html>"""

    # --- MÉTODO DE SEGURIDAD ---
    def visitTerminal(self, node: TerminalNode) -> str:
        # Si el visitor encuentra un token terminal (hoja) directamente,
devuelve su texto.
        # Esto previene que devuelva None y rompa concatenaciones.
return node.getText()

    def visitBlock(self, ctx: MarkdownGrammarParser.BlockContext) -> str:
        for child in ctx.getChildren():
            if isinstance(child, ParserRuleContext):
                return self.visit(child)
        return ''

    def visitHeader(self, ctx: MarkdownGrammarParser.HeaderContext) -> str:
        level = len(ctx.HASHES().getText())
        inline_ctx = ctx.inlineContent()
        content = self.visit(inline_ctx) if inline_ctx is not None else ''
        return f"<h{level}>{content}</h{level}>\n"

    def visitParagraph(self, ctx: MarkdownGrammarParser.ParagraphContext) ->
str:
        content = self.visit(ctx.inlineContent())
        return f"<p>{content}</p>\n"

    def visitHorizontalRule(self, ctx:
MarkdownGrammarParser.HorizontalRuleContext) -> str:
```

```
        return "<hr/>\n"

    def visitBlockquote(self, ctx: MarkdownGrammarParser.BlockquoteContext) ->
str:
        lines: List[str] = []
        for line_ctx in ctx.blockquoteLine():
            if line_ctx.inlineContent():
                lines.append(self.visit(line_ctx.inlineContent()))
            else:
                lines.append('')
        inner = '<br/>'.join(lines)
        return f"<blockquote>{inner}</blockquote>\n"

    # --- INICIO LÓGICA DE TABLAS ---
    def visitTable(self, ctx: MarkdownGrammarParser.TableContext) -> str:
        header_html = self.visit(ctx.tableHeaderLine())

        body_parts = []
        for body_line in ctx.tableBodyLine():
            body_parts.append(self.visit(body_line))
        body_html = ''.join(body_parts)

        return f"""<table>
<thead>
{header_html}</thead>
<tbody>
{body_html}</tbody>
</table>
"""

    def visitTableHeaderLine(self, ctx:
MarkdownGrammarParser.TableHeaderLineContext) -> str:
        row_html = self.visit(ctx.tableRow())
        return row_html.replace("<td>", "<th>").replace("</td>", "</th>")

    def visitTableSepLine(self, ctx:
MarkdownGrammarParser.TableSepLineContext) -> str:
        return ""

    def visitTableBodyLine(self, ctx:
MarkdownGrammarParser.TableBodyLineContext) -> str:
        return self.visit(ctx.tableRow())

    def visitTableRow(self, ctx: MarkdownGrammarParser.TableRowContext) ->
str:
        cells = []
```

```
        current_cell_nodes = []

        # Obtenemos todos los hijos
        children = list(ctx.getChildren())

        # Bandera para saber si ya entramos a la zona de celdas (después del
primer pipe)
        first_pipe_passed = False

        for child in children:
            text = child.getText()

            # Ignoramos espacios (WS) que estén ANTES del primer pipe
            if not first_pipe_passed:
                if text.strip() == '|':
                    first_pipe_passed = True
                continue
            # Procesamiento normal de celdas
            if text.strip() == '|':
                # Al encontrar un pipe de cierre, guardamos lo acumulado como
celda
                cell_content = ""
                for node in current_cell_nodes:
                    res = self.visit(node)
                    if res is not None:
                        cell_content += res
                    else:
                        cell_content += node.getText()

                cells.append(f"<td>{cell_content.strip()}</td>")
                current_cell_nodes = []
            elif text.strip() == '' and ('\n' in text or '\r' in text):
                # Ignorar saltos de línea al final
                continue
            else:
                # Acumular contenido de la celda actual
                current_cell_nodes.append(child)

        return f"<tr>{''.join(cells)}</tr>\n"
    # --- FIN LÓGICA DE TABLAS ---

    def visitList(self, ctx: MarkdownGrammarParser.ListContext) -> str:
        items_ctx = ctx.listItem()
        if not items_ctx:
            return ''
        first = items_ctx[0]
```

```
        if isinstance(first, MarkdownGrammarParser.OrderedItemContext):
            open_tag, close_tag = '<ol>', '</ol>'
        else:
            open_tag, close_tag = '<ul>', '</ul>'

        items_html = []
        for item_ctx in items_ctx:
            items_html.append(self.visit(item_ctx))
        return f"{open_tag}\n{''.join(items_html)}{close_tag}\n"

    def visitUnorderedItem(self, ctx:
MarkdownGrammarParser.UnorderedItemContext) -> str:
        content = self.visit(ctx.inlineContent())
        return f"<li>{content}</li>\n"

    def visitOrderedItem(self, ctx: MarkdownGrammarParser.OrderedItemContext)
-> str:
        content = self.visit(ctx.inlineContent())
        return f"<li>{content}</li>\n"

    def visitInlineContent(self, ctx:
MarkdownGrammarParser.InlineContentContext) -> str:
        parts: List[str] = []
        for item in ctx.inlineItem():
            parts.append(self.visit(item))
        return ''.join(parts)

    def visitInlineItem(self, ctx: MarkdownGrammarParser.InlineItemContext) ->
str:
        # 1. Estructuras complejas: Delegamos a sus métodos específicos
        if ctx.link(): return self.visit(ctx.link())
        if ctx.bold_italic(): return self.visit(ctx.bold_italic())
        if ctx.bold(): return self.visit(ctx.bold())
        if ctx.italic(): return self.visit(ctx.italic())
        if ctx.strikethrough(): return self.visit(ctx.strikethrough())

        # 2. Todo lo demás (Texto plano, Puntuación, Espacios, Símbolos
nuevos):
        # En lugar de listar cada token (DOT, STAR, etc.), simplemente
devolvemos
        # el texto que capturó el parser. Esto cubre CUALQUIER token simple.
return ctx.getText()

    def visitInlineItem_no_pipe(self, ctx:
MarkdownGrammarParser.InlineItem_no_pipeContext) -> str:
        # Misma lógica simplificada para tablas
```

```
        if ctx.link(): return self.visit(ctx.link())
        if ctx.bold(): return self.visit(ctx.bold())
        if ctx.italic(): return self.visit(ctx.italic())
        if ctx.strikethrough(): return self.visit(ctx.strikethrough())

        # Devuelve el texto de cualquier otro token (TEXT, WS, DOT, números,
etc.)
        return ctx.getText()

    def visitLink(self, ctx: MarkdownGrammarParser.LinkContext) -> str:
        text = self.visit(ctx.inlineContent())
        url = ctx.URL().getText()
        return f'<a href="{url}">{text}</a>'

    def visitBold(self, ctx: MarkdownGrammarParser.BoldContext) -> str:
        content = self.visit(ctx.inlineContent())
        return f'<strong>{content}</strong>'

    def visitItalic(self, ctx: MarkdownGrammarParser.ItalicContext) -> str:
        content = self.visit(ctx.inlineContent())
        return f'<em>{content}</em>'

    def visitBold_italic(self, ctx: MarkdownGrammarParser.Bold_italicContext)
-> str:
        content = self.visit(ctx.inlineContent())
        return f'<strong><em>{content}</em></strong>'

    def visitStrikethrough(self, ctx:
MarkdownGrammarParser.StrikethroughContext) -> str:
        content = self.visit(ctx.inlineContent())
        return f'<del>{content}</del>'

    # Ya no necesitamos visitText, visitDot, visitStar, etc. por separado
    # porque visitInlineItem los cubre todos con ctx.getText()
    def visitChildren(self, node):
        return ''.join([self.visit(child) for child in node.getChildren()])


def translate_markdown_to_html(md_text: str) -> str:
    from antlr4 import InputStream, CommonTokenStream
    from MarkdownGrammarLexer import MarkdownGrammarLexer
    lexer = MarkdownGrammarLexer(InputStream(md_text))
    token_stream = CommonTokenStream(lexer)
    parser = MarkdownGrammarParser(token_stream)
    tree = parser.doc()
```

```
    visitor = MarkdownToHtmlVisitor()
    return visitor.visit(tree)
```

Main.py

```
import sys
import os
import webbrowser
import tempfile
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout,
                             QHBoxLayout, QPushButton, QTextEdit, QFileDialog,
                             QSplitter, QMessageBox, QLabel, QDialog)
from PyQt5.QtCore import Qt, QTimer
from PyQt5.QtGui import QFont

# --- IMPORTACIÓN DE TU LÓGICA ANTLR ---
try:
    from antlr4 import InputStream, CommonTokenStream
    from src.MarkdownGrammarLexer import MarkdownGrammarLexer
    from src.MarkdownGrammarParser import MarkdownGrammarParser
    from src.MarkdownVisitor import MarkdownToHtmlVisitor

    ANTLR_AVAILABLE = True
except ImportError:
    ANTLR_AVAILABLE = False
    print("ADVERTENCIA: No se encontraron los archivos de ANTLR
(Lexer/Parser/Visitor).")
    print("Se usará un modo de simulación para la interfaz gráfica.")


class MarkdownConverter:
    """Clase encargada de la lógica de negocio y formateo."""

    @staticmethod
    def convert(text: str) -> str:
        """Convierte Markdown a HTML."""
        if not text.endswith("\n"):
            text += "\n"

        if ANTLR_AVAILABLE:
            try:
                lexer = MarkdownGrammarLexer(InputStream(text))
                token_stream = CommonTokenStream(lexer)
                parser = MarkdownGrammarParser(token_stream)
                tree = parser.doc()
```

```
                visitor = MarkdownToHtmlVisitor()
                return visitor.visit(tree)
            except Exception as e:
                return f"<div style='color:red;'>Error de Parsing: {str(e)}
</div>"
        else:
            return f"<h1>Simulación</h1><p>{text}</p>"

    @staticmethod
    def generate_ast(text: str) -> str:
        """Genera el AST formateado con indentación para mejor lectura."""
        if not text.endswith("\n"):
            text += "\n"

        raw_ast = ""
        if ANTLR_AVAILABLE:
            try:
                lexer = MarkdownGrammarLexer(InputStream(text))
                token_stream = CommonTokenStream(lexer)
                parser = MarkdownGrammarParser(token_stream)
                tree = parser.doc()
                # Obtenemos la cadena LISP plana de ANTLR: (rule (child1)
(child2))
                raw_ast = tree.toStringTree(recog=parser)
            except Exception as e:
                return f"Error generando AST: {str(e)}"
        else:
            # AST Simulado para pruebas sin librerías
            raw_ast = "(doc (header (level ##) (text Tabla de ejemplo))
(paragraph (text ...)))"

        # Formateamos la cadena plana a algo visualmente indentado
        return MarkdownConverter.format_ast_string(raw_ast)

    @staticmethod
    def format_ast_string(lisp_str: str) -> str:
        """
        Transforma una cadena estilo LISP '(a (b c))' en un árbol indentado:
a          b            c        """        formatted_output = []
        indent_level = 0
        current_token = ""

        # Recorremos caracter por caracter para formatear
        # Nota: Esta es una lógica simple de visualización basada en
paréntesis        for char in lisp_str:
            if char == '(':
```

```
                if current_token.strip():
                    formatted_output.append(current_token.strip())
                    current_token = ""

                if formatted_output:  # Nueva línea para cada nodo nuevo
                    formatted_output.append("\n")

                formatted_output.append("    " * indent_level)  # Indentación
                formatted_output.append("├─ ")  # Adorno opcional
                indent_level += 1
            elif char == ')':
                if current_token.strip():
                    formatted_output.append(current_token.strip())
                    current_token = ""
                indent_level -= 1
            else:
                current_token += char

        if current_token.strip():
            formatted_output.append(current_token.strip())

        return "".join(formatted_output)


class ASTViewerDialog(QDialog):
    """Ventana secundaria para visualizar el AST."""

    def __init__(self, ast_content, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Visualizador de AST")
        self.resize(600, 800)

        layout = QVBoxLayout(self)

        lbl = QLabel("Árbol de Sintaxis Abstracta generado:")
        layout.addWidget(lbl)

        self.text_view = QTextEdit()
        self.text_view.setReadOnly(True)
        # Usamos fuente monoespaciada para que la indentación se vea perfecta
        self.text_view.setFont(QFont("Consolas", 10))
        self.text_view.setPlainText(ast_content)

        layout.addWidget(self.text_view)

        btn_close = QPushButton("Cerrar")
```

```
        btn_close.clicked.connect(self.accept)
        layout.addWidget(btn_close)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Traductor Markdown a HTML (ANTLR)")
        self.resize(1000, 700)

        # --- Temporizador ---
        self.timer = QTimer()
        self.timer.setSingleShot(True)
        self.timer.setInterval(800)
        self.timer.timeout.connect(self.run_conversion)

        self.init_ui()

    def init_ui(self):
        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        main_layout = QVBoxLayout(central_widget)
        main_layout.setContentsMargins(10, 10, 10, 10)
        main_layout.setSpacing(10)

        # --- BARRA SUPERIOR ---
        btn_layout = QHBoxLayout()
        btn_style = "QPushButton { background-color: #2b5b84; color: white;
padding: 8px; border-radius: 4px; font-weight: bold; } QPushButton:hover {
background-color: #3d7fb5; }"

        self.btn_open = QPushButton("Abrir Archivo")
        self.btn_open.setStyleSheet(btn_style)
        self.btn_open.clicked.connect(self.open_file)

        self.btn_ast = QPushButton("Generar AST")
        self.btn_ast.setStyleSheet(btn_style)
        self.btn_ast.clicked.connect(self.generate_and_save_ast)

        self.btn_convert = QPushButton("Convertir Manualmente")
        self.btn_convert.setStyleSheet(btn_style)
        self.btn_convert.clicked.connect(self.run_conversion)

        # Nuevo botón para abrir en navegador
        self.btn_browser = QPushButton("Abrir en Navegador")
```

```
        self.btn_browser.setStyleSheet(btn_style)
        self.btn_browser.clicked.connect(self.open_in_browser)

        self.btn_view_html = QPushButton("Ver HTML / Ocultar")
        self.btn_view_html.setStyleSheet(btn_style)
        self.btn_view_html.clicked.connect(self.toggle_preview)

        self.btn_close = QPushButton("Cerrar")
        self.btn_close.setStyleSheet("background-color: #d9534f; color: white;
padding: 8px; border-radius: 4px;")
        self.btn_close.clicked.connect(self.close)

        btn_layout.addWidget(self.btn_open)
        btn_layout.addWidget(self.btn_ast)
        btn_layout.addWidget(self.btn_convert)
        btn_layout.addWidget(self.btn_browser)  # Agregado al layout
        btn_layout.addWidget(self.btn_view_html)
        btn_layout.addStretch()
        btn_layout.addWidget(self.btn_close)

        main_layout.addLayout(btn_layout)

        # --- ÁREA DE TRABAJO ---
        splitter = QSplitter(Qt.Horizontal)

        self.text_editor = QTextEdit()
        self.text_editor.setPlaceholderText("Escribe tu Markdown aquí o abre
un archivo...")
        self.text_editor.setFont(QFont("Consolas", 11))
        self.text_editor.setStyleSheet("border: 1px solid #ccc; background-
color: #f9f9f9;")
        self.text_editor.textChanged.connect(self.reset_timer)

        self.html_viewer = QTextEdit()
        self.html_viewer.setReadOnly(True)
        self.html_viewer.setPlaceholderText("Vista previa del HTML
generado...")
        self.html_viewer.setStyleSheet("border: 1px solid #ccc; background-
color: #ffffff;")

        splitter.addWidget(self.text_editor)
        splitter.addWidget(self.html_viewer)
        splitter.setSizes([500, 500])

        main_layout.addWidget(splitter, 1)  # Factor de estiramiento 1 para
llenar pantalla
```

```
        self.status_label = QLabel("Listo.")
        main_layout.addWidget(self.status_label)

    def reset_timer(self):
        self.status_label.setText("Escribiendo... (Esperando para convertir)")
        self.timer.start()

    def run_conversion(self):
        markdown_text = self.text_editor.toPlainText()
        html_output = MarkdownConverter.convert(markdown_text)
        self.html_viewer.setHtml(html_output)
        self.status_label.setText("Conversión completada.")

    def open_file(self):
        options = QFileDialog.Options()
        file_path, _ = QFileDialog.getOpenFileName(self, "Abrir archivo
Markdown", "",
                                                   "Markdown Files (*.md);;All
Files (*)", options=options)
        if file_path:
            try:
                with open(file_path, "r", encoding="utf-8") as f:
                    content = f.read()
                    self.text_editor.setPlainText(content)
                    self.status_label.setText(f"Archivo cargado:
{os.path.basename(file_path)}")
            except Exception as e:
                QMessageBox.critical(self, "Error", f"No se pudo abrir el
archivo:\n{e}")

    def generate_and_save_ast(self):
        """Genera el AST, lo guarda en archivo y lo muestra."""
        markdown_text = self.text_editor.toPlainText()
        ast_formatted = MarkdownConverter.generate_ast(markdown_text)

        # 1. Guardar Archivo
        options = QFileDialog.Options()
        file_path, _ = QFileDialog.getSaveFileName(self, "Guardar AST en
archivo", "ast_output.txt",
                                                   "Text Files (*.txt);;All
Files (*)", options=options)

        if file_path:
            try:
                with open(file_path, "w", encoding="utf-8") as f:
```

```
                    f.write(ast_formatted)
                self.status_label.setText(f"AST guardado en: {file_path}")
            except Exception as e:
                QMessageBox.warning(self, "Error al guardar", f"No se pudo
guardar el archivo:\n{e}")

        # 2. Mostrar Gráficamente (Texto Indentado)
        viewer = ASTViewerDialog(ast_formatted, self)
        viewer.exec_()

    def open_in_browser(self):
        """Genera el HTML y lo abre en el navegador web por defecto."""
        markdown_text = self.text_editor.toPlainText()
        html_output = MarkdownConverter.convert(markdown_text)

        try:
            # Creamos un archivo temporal con extensión .html
            # delete=False para que el archivo persista lo suficiente para ser
abierto             with tempfile.NamedTemporaryFile(delete=False,
suffix='.html', mode='w', encoding='utf-8') as f:
                f.write(html_output)
                temp_path = f.name

            # Abrimos el archivo en el navegador
            webbrowser.open('file://' + temp_path)
            self.status_label.setText(f"Abierto en navegador: {temp_path}")

        except Exception as e:
            QMessageBox.critical(self, "Error", f"No se pudo abrir el
navegador:\n{e}")

    def toggle_preview(self):
        if self.html_viewer.isVisible():
            self.html_viewer.hide()
        else:
            self.html_viewer.show()


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Driver.py

```python
from antlr4 import InputStream, CommonTokenStream

from src.MarkdownGrammarLexer import MarkdownGrammarLexer
from src.MarkdownGrammarParser import MarkdownGrammarParser
from src.MarkdownVisitor import MarkdownToHtmlVisitor


def convert_markdown_to_html(text: str) -> str:
    """Convierte un texto en Markdown a una cadena HTML usando ANTLR."""
    # Asegurar un salto de línea final para que el parser no falle al final de
archivo
    if not text.endswith("\n"):
        text += "\n"
    lexer = MarkdownGrammarLexer(InputStream(text))
    token_stream = CommonTokenStream(lexer)
    parser = MarkdownGrammarParser(token_stream)
    tree = parser.doc()
    visitor = MarkdownToHtmlVisitor()
    return visitor.visit(tree)


def convert_file(input_path: str, output_path: str) -> None:
    """Lee un archivo Markdown y escribe el HTML resultante en otro
archivo."""
    with open(input_path, "r", encoding="utf-8") as f:
        md_text = f.read()
    html = convert_markdown_to_html(md_text)
    with open(output_path, "w", encoding="utf-8") as f:
        f.write(html)


if __name__ == "__main__":
    convert_file("entrada.md", "salida.html")
```

Troubleshooting y Preguntas Frecuentes (Q&A)

Esta sección reúne los problemas más comunes que pueden presentarse al usar el traductor de Markdown a HTML, junto con sus posibles causas y soluciones. También incluye una breve lista de preguntas frecuentes sobre el funcionamiento del sistema.

Troubleshooting (Solución de problemas)

La aplicación no abre o se cierra al iniciar

Posibles causas:

- No está instalada la versión correcta de Python.
- Faltan dependencias (por ejemplo, `antlr4-python3-runtime` o `PyQt5`).
- Se está ejecutando el archivo desde una carpeta distinta a la del proyecto.

Solución:

1. Verificar la versión de Python:
   `python --version`
   Debe ser 3.8 o superior.
2. Instalar las dependencias:
   `pip install antlr4-python3-runtime PyQt5`
3. Ejecutar la aplicación desde la carpeta raíz del proyecto:
   `python Main.py`

Aparecen errores de ANTLR o de parser al convertir

Síntomas:

- Mensajes como `line X:Y mismatched input` o `missing NEWLINE at '<EOF>'`.
- El programa indica que no puede analizar el archivo.

Posibles causas:

- El texto Markdown contiene estructuras que la gramática no soporta.
- El documento no tiene saltos de línea donde la gramática los espera.
- Se modificó la gramática `.g4` pero no se regeneraron los archivos de ANTLR.

Solución:

1. Verificar que solo se usen los elementos de Markdown definidos en el alcance del proyecto (encabezados, listas, tablas, citas, formato básico, enlaces y reglas horizontales).
2. Asegurarse de que los encabezados tengan un espacio después de los `#`, por ejemplo:

   `# Título correcto`
   y no:
   `#Título incorrecto`
3. Si se modificó la gramática, regenerar los archivos de ANTLR:
   `antlr4 -Dlanguage=Python3 -visitor MarkdownGrammar.g4`

Todo el documento se convierte en un solo párrafo `<p>`

Síntomas:

- La salida HTML muestra todo el contenido dentro de una sola etiqueta `<p>`.
- No se ven encabezados, listas ni citas aunque se escriban.

Posibles causas:

- Falta de líneas en blanco entre bloques de Markdown.
- Los encabezados o listas no respetan la sintaxis esperada por la gramática.

Solución:

1. Insertar líneas en blanco entre párrafos, listas y encabezados.
2. Revisar que las listas y encabezados tengan el formato correcto, por ejemplo:
   `# Encabezado - Elemento 1 - Elemento 2`
3. Probar primero con ejemplos pequeños (como los de la sección 3.3) para verificar que el traductor funciona y luego aplicar el mismo estilo al documento grande.

No se genera el archivo HTML en modo consola

Síntomas:

- Se ejecuta `Driver.py` pero no aparece el archivo de salida.
- No se muestra mensaje de error claro.

Posibles causas:

- Se pasó mal el nombre del archivo o la ruta.
- El archivo de entrada no existe o no es `.md`.
- No se proporcionaron los dos parámetros esperados (entrada y salida).

Solución:

1. Verificar el comando:
   `python Driver.py entrada.md salida.html`
2. Confirmar que `entrada.md` existe en la carpeta desde donde se ejecuta el comando.
3. Si se usan rutas, probar primero con rutas simples y sin espacios para descartar errores de escritura.

El HTML se ve “raro” en el navegador

Síntomas:

- El navegador muestra texto sin formato o etiquetas visibles.
- Algunos elementos aparecen desalineados o sin estilos.

Posibles causas:

- El archivo HTML se abrió con un editor de texto y luego se guardó con modificaciones incorrectas.
- Solo se visualiza el fragmento HTML sin una estructura mínima de documento (`<html>` , `<head>` , `<body>`).
- El navegador está mostrando un caché antiguo.

Solución:

1. Verificar que el archivo tenga una estructura básica de página (si aplica).
2. Volver a generar el HTML con la aplicación y abrirlo directamente desde el explorador de archivos.
3. Actualizar la página en el navegador (Ctrl+F5) para limpiar caché.

6.2. Preguntas Frecuentes (Q&A)

P1. ¿Qué tipos de elementos de Markdown soporta el traductor?
R: Soporta encabezados de nivel 1 a 6, listas ordenadas y no ordenadas, citas, tablas, texto en negritas, cursiva, negritas+cursiva, tachado, subrayado (si la gramática lo define), enlaces en línea y reglas horizontales. Otros elementos de Markdown no están garantizados.

P2. ¿Puedo usar el traductor con archivos `.txt` o `.docx` ?
R: El proyecto está pensado para trabajar con archivos en formato Markdown (`.md`). Si tienes un `.txt` , puedes renombrarlo a `.md` siempre que el contenido siga la sintaxis de Markdown. Los archivos `.docx` no son soportados.

P3. ¿La aplicación soporta imágenes y bloques de código?
R: En la versión actual del proyecto no se garantiza el soporte para imágenes ni bloques de código en Markdown. Solo se procesan los elementos incluidos en el alcance definido para la gramática.

P4. ¿Por qué un encabezado con más de seis `#` genera error o no se interpreta?
R: La gramática solo permite encabezados de nivel 1 a 6 (`#` a `######`). Cualquier otro caso se considera inválido o se trata como texto normal.

P5. ¿Se puede personalizar el estilo visual del HTML generado (colores, fuentes, etc.)?
R: El traductor genera HTML estructural (etiquetas como `<h1>`, `<p>`, `<ul>`, `<table>`, etc.). El diseño visual (CSS) puede agregarse después, ya sea editando el HTML generado o enlazando una hoja de estilos. La aplicación en sí no se encarga de los estilos.

P6. ¿Puedo integrar el traductor en otro sistema o automatizar conversiones masivas?
R: Sí. Para ello se recomienda usar el modo consola (`Driver.py`), llamándolo desde scripts o herramientas externas y pasándole como parámetros los archivos de entrada y salida.

P7. ¿Qué debo hacer si modifico la gramática y el proyecto deja de funcionar?
R: Primero, regenerar los archivos de ANTLR. Si el problema persiste, revisar la regla que se modificó y probar con ejemplos pequeños. Es buena práctica hacer cambios incrementales y usar control de versiones para poder volver a una versión estable.