Universidad de San Carlos de Guatemala
Escuela de Ciencias y Sistemas
Facultad de Ingeniería Lenguajes formales y de programación
Vacaciones Primer Semestre 2025
Catedrático: Inga. Asunción Mariana Sic Sor
Tutor académico: Elian Saúl Estrada Urbina

PROYECTO 2

"Transpilador"

Axel David González Molina
202402074
Fecha de entrega:
29/6/2025

Introducción

Este proyecto se centra en el desarrollo de un transpilador completo en TypeScript, diseñado para traducir código fuente escrito en C# a su equivalente en TypeScript, abarcando las estructuras de programación más utilizadas en el desarrollo de aplicaciones.

La necesidad de un sistema que permita la traducción automática entre estos lenguajes surge de la creciente demanda de interoperabilidad en el ecosistema de desarrollo web y de aplicaciones, donde TypeScript ha ganado popularidad significativa por su capacidad de proporcionar tipado estático sobre JavaScript. El transpilador no solo se encargará de realizar la traducción sintáctica del código, sino que también implementará un intérprete capaz de ejecutar las instrucciones de salida por consola, proporcionando una validación inmediata de la funcionalidad del código traducido.

A través de la implementación de un analizador léxico basado en autómatas finitos deterministas (AFD) y un analizador sintáctico fundamentado en gramáticas libres de contexto mediante autómatas de pila, el sistema será capaz de reconocer y procesar patrones complejos del lenguaje C#.

El objetivo principal de este proyecto es implementar un análisis léxico y sintáctico robusto que no solo cumpla con los requisitos de traducción, sino que también proporcione una comprensión profunda de los conceptos fundamentales en la teoría de compiladores y lenguajes formales. La interfaz gráfica desarrollada será intuitiva y funcional, permitiendo a los usuarios cargar archivos C# (.cs), visualizar la traducción resultante en TypeScript (.ts), monitorear la ejecución a través de una consola integrada y acceder a reportes detallados de tokens, errores y símbolos. Este documento establece los fundamentos teóricos y metodológicos necesarios para la implementación exitosa de esta herramienta de transpilación.

Manual técnico

A continuación se describirán los métodos creados y requerimientos de la aplicación.

Requerimientos:

1. Componentes Clave de la Plataforma

a) Entorno de Ejecución de TypeScript

- **Node.js:**
  - **Versión mínima:** Node.js 14 o superior (requerido para compatibilidad con las últimas características de TypeScript).
- **Bibliotecas Específicas:**
  - **TypeScript:** Para la escritura y compilación del código en TypeScript.
  - **Express:** Para la creación del servidor web que manejará las solicitudes y respuestas.
  - **React:** Para la construcción de la interfaz de usuario interactiva del pensum.
  - **Recharts:** Para la generación de gráficos interactivos que muestren la estructura del pensum.

b) Sistema Operativo Compatible

- **Windows:**
  - **Versiones soportadas:** 10, 11 (64-bit).
  - **Requisitos adicionales:** Instalación de Microsoft Visual C++ Redistributable para dependencias nativas.
- **Linux:**
  - **Distros validadas:** Ubuntu 22.04 LTS, Fedora 36+.
  - **Nota:** Configuración de permisos de escritura en el directorio de trabajo del proyecto.
- **macOS:**
  - **Versiones:** Monterey (12.x) o superior.

2. Configuraciones Esenciales

a) Variables de Entorno

- **NODE_ENV:** Debe estar configurada como 'development' o 'production' según el entorno de ejecución.
- **PORT:** Puerto en el que se ejecutará la aplicación (por defecto, 3000).
- **API_URL:** Ruta base para posibles integraciones con sistemas universitarios (opcional).

b) Espacio en Disco

- **Mínimo:** 100 MB (para archivos de datos y dependencias).
- **Recomendado:** 500 MB SSD (mejor rendimiento en operaciones de lectura/escritura, especialmente para generar visualizaciones complejas).

3. Dependencias de Visualización

- **Bibliotecas Gráficas:**
  - **D3.js o Recharts:** Para generar visualizaciones interactivas del pensum académico.

- **React Flow:** Para crear diagramas de flujo que muestren las relaciones entre cursos y prerrequisitos.

4. Herramientas de Soporte

a) IDE para Desarrollo

- **Visual Studio Code:** Recomendado por su soporte completo para TypeScript, React y extensiones para visualización de datos.

b) Monitor de Recursos

- **Uso de Memoria:** Supervisión continua para garantizar un rendimiento óptimo durante la generación de visualizaciones complejas.
- **Optimización de Gráficos:** Herramientas para analizar el rendimiento de las visualizaciones interactivas.

Métodos creados:

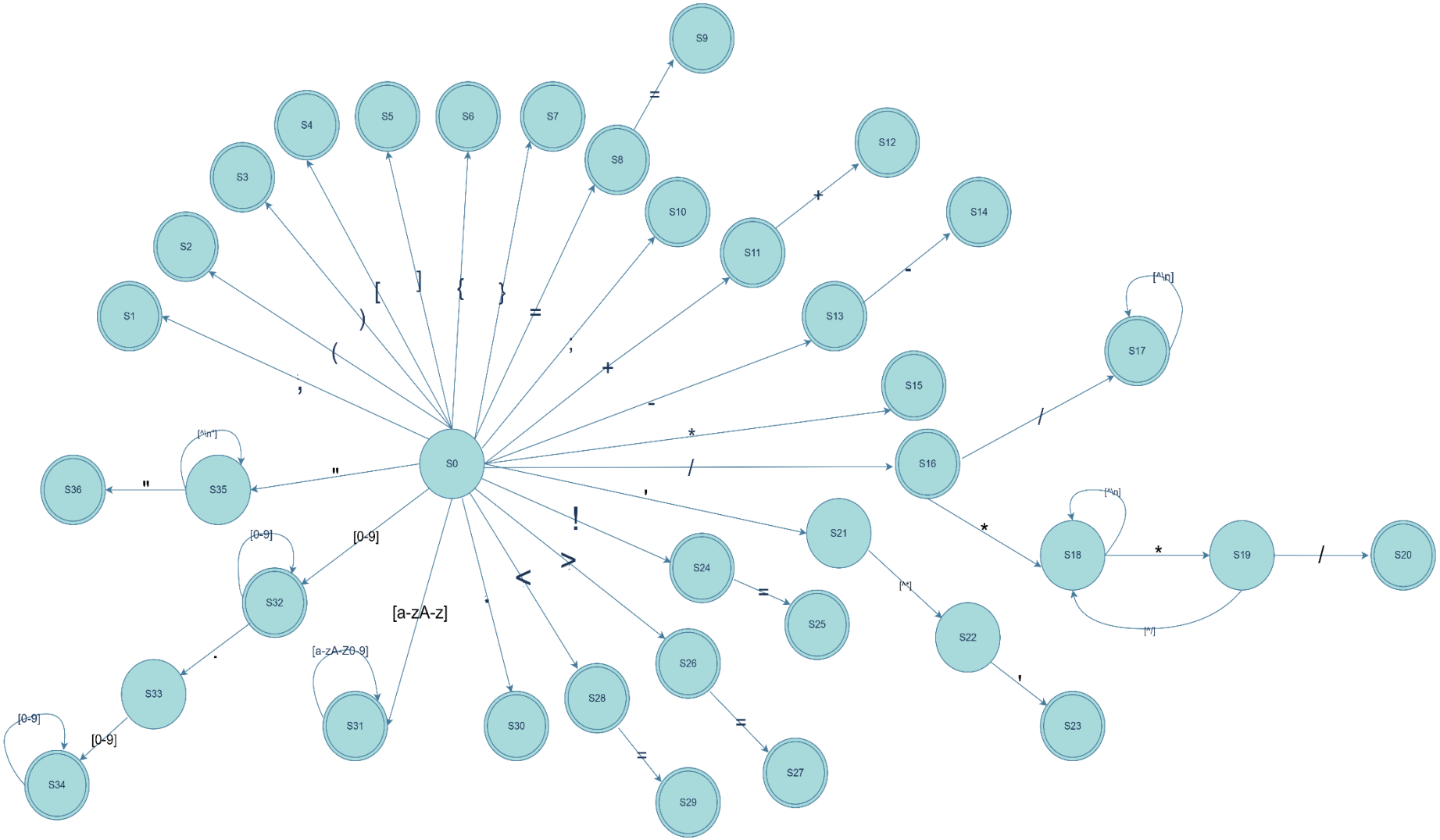
Para el desarrollo de la aplicación en el lenguaje TypeScript se trabajó con un AFD que nos permitió trabajar con el analizador léxico para identificar tokens, se crearon varios métodos auxiliares para poder simplificar y trabajar de manera más ordenada y eficiente el código, a continuación se mencionarán algunos de los métodos principales para el desarrollo del programa.

Diccionario de métodos usados:

Método	Propósito
analizarTexto()	Función principal del cliente para enviar el texto al backend y obtener tokens/errores.
mostrarTokens(tokens)	Muestra los tokens recibidos en la tabla HTML.
colorearTexto(tokens)	Resalta visualmente los tokens en el editor aplicando estilos según su tipo.
cargarArchivo()	Permite al usuario subir un archivo .plfp y cargar su contenido al editor.
guardarArchivo()	Guarda el contenido del editor como un archivo de texto descargable.
limpiarEditor()	Limpia tanto el contenido del editor como la tabla de tokens.
DOMContentLoaded	Inicializa eventos y referencias al cargar la página.
clear.addEventListener	Limpia editor, salida, tablas y localStorage.
open.addEventListener	Permite cargar un archivo .cs al editor.
save.addEventListener	Descarga el contenido del editor como archivo .cs.
button.addEventListener	Envía el texto al backend para analizar y muestra resultados.
fetch('/analyze')	Solicita el análisis al backend y recibe tokens y errores.
alert("¡Se encontraron errores léxicos!")	Muestra un mensaje si hay errores léxicos.
table.innerHTML = textTable	Muestra los tokens en la tabla correspondiente.
tableError.innerHTML = textErrors	Muestra los errores sintácticos en la tabla de errores.
editor.innerHTML = result.colors	Colorea el editor con el resultado del análisis.
salida.innerText = result.traduction	Muestra la traducción si no hay errores sintácticos.
localStorage.setItem	Guarda errores léxicos o sintácticos para otras vistas.
localStorage.clear()	Limpia todos los datos almacenados localmente.
FileReader.readAsText	Lee el contenido de un archivo cargado por el usuario.
window.location.href	Redirige a otra vista (por ejemplo, a la de errores léxicos).
JSON.stringify / JSON.parse	Convierte datos entre string y objeto para almacenamiento.
forEach sobre result.tokens	Recorre y muestra los tokens obtenidos del análisis.
forEach sobre result.syntacticErrors	Recorre y muestra los errores sintácticos.

forEach sobre result.errors	Recorre y muestra los errores léxicos (en la vista de errores).
addEventListener('change')	Detecta cuando el usuario selecciona un archivo para abrir.
download.click()	Dispara la descarga del archivo generado.
alert("Error en el servidor...")	Muestra un mensaje si el backend no responde correctamente.
console.error	Muestra errores en la consola del navegador para depuración.
editor.innerText	Obtiene o establece el texto del editor.
salida.innerText	Muestra mensajes o traducciones en el área de salida.
table.innerHTML = ''	Limpia la tabla de tokens.
tableError.innerHTML = ''	Limpia la tabla de errores.
fileInput.click()	Abre el diálogo para seleccionar archivos.
reader.onload	Evento que se ejecuta cuando se termina de leer un archivo.
download.href	Define el contenido a descargar como archivo.
analyze(req, res) (controlador)	Recibe el texto, ejecuta el análisis léxico y devuelve tokens, errores y pensum.
showIndex(req, res)	Renderiza la vista principal con tokens, errores y carreras si existen.
showErrors(req, res)	Renderiza la vista errors.ejs con la tabla de errores léxicos.
addToken	Agrega un token a la lista de tokens durante el análisis léxico.
addError	Agrega un error a la lista de errores durante el análisis.
getTokenList	Devuelve la lista de tokens generados por el analizador léxico.
getErrorList	Devuelve la lista de errores léxicos encontrados.
clean	Limpia el estado interno del analizador (tokens, errores, etc).
parser	Ejecuta el análisis sintáctico sobre la lista de tokens.
getErrors	Devuelve la lista de errores sintácticos encontrados.
transpile	Realiza la traducción del código fuente a otro lenguaje.
render	Renderiza una vista EJS desde el backend.
set	Configura una propiedad o parámetro en la aplicación Express.

Autómata Finito Determinista (AFD)



Conclusiones

El desarrollo de un transpilador de C# a TypeScript aplica principios fundamentales de teoría de compiladores y lenguajes formales. La implementación de analizadores léxico y sintáctico mediante autómatas finitos y gramáticas libres de contexto permite una traducción efectiva y la interpretación de instrucciones, validando la funcionalidad del sistema.

Además, la interfaz gráfica de usuario, que incluye capacidades de edición, análisis y generación de reportes, es una herramienta educativa valiosa que ayuda a comprender conceptos complejos como tablas de símbolos y análisis semántico.

El proyecto equilibra robustez técnica y usabilidad, proporcionando reportes detallados que facilitan el debugging y la comprensión del transpilador. Esta aproximación integral sienta las bases para futuras extensiones y el estudio avanzado de la teoría de compilador