Manual técnico

TRADUCTOR DE CODIGO JAVA A PYTHON UTILIZANDO ANALIZADOR LÉXICO Y SINTÁCTICO

Mishell Monroy

PROYECTO 2, LENGUAJES FORMALES DE PROGRAMACIÓN 202404409

Índice

Introducción

Este manual, explica la creación del código, para la elaboración de un traductor de código java a Python, utilizando un analizador léxico y sintáctico, este proyecto está libre de librerías como regex, pues busca simular mejor como es el comportamiento de las fases de un copilador, que hace para poder traducir el código, en este caso vemos explícitamente estos dos importantísimos analizadores, puesto que es exactamente lo que hace un compilador, traduce el texto en lenguaje de programación, un lenguaje que todos los humanos podemos entender para desarrollar aplicaciones, paginas, etc. A un lenguaje máquina, que sabemos es en binario, cosa que para el ser humano es casi que imposible traducir. Bien, pues en este caso es similar, se traduce un lenguaje en formato .java a un lenguaje en formato .py utilizando tokens, para representar las palabras que ingresa, definir las entradas validas y la estructura valida.

Objetivos:

Brindar una idea clara de cómo fue creado el programa, para que cualquier otra persona pueda interpretar de una buena manera el código creado, incluso es viable ver este manual para cuando se le desea hacer una actualización el programa, se sabe dónde se debe de hacer, debido a que en este documento se explica de forma clara y ordenada las funciones del código.

Dirigido

Está dirigido a todas aquellas personas que deseen modificar o actualizar el programa, también para quienes deseen un programa como este, para que sepan cómo deben leer el código y como es su funcionamiento, incluso está dirigido a el programador en sí, ya que de esta forma puede saber de una manera más clara y ordenada que es y donde está el código que desea implementar o actualizar.

Especificación técnica

Requisitos de hardware

- Disco duro capaz de correr el programa, a partir de 250 GB
- Procesador Core i5 o superior
- 4GB de memoria RAM como mínimo.

Requisitos de Software

- Debe tener instalado algún editor de código fuente, para usar el programa directamente, Computadora de escritorio o portátil, se recomienda utilizar Visual Studio Code.
- Node, version v22.17.1

Lógica del programa

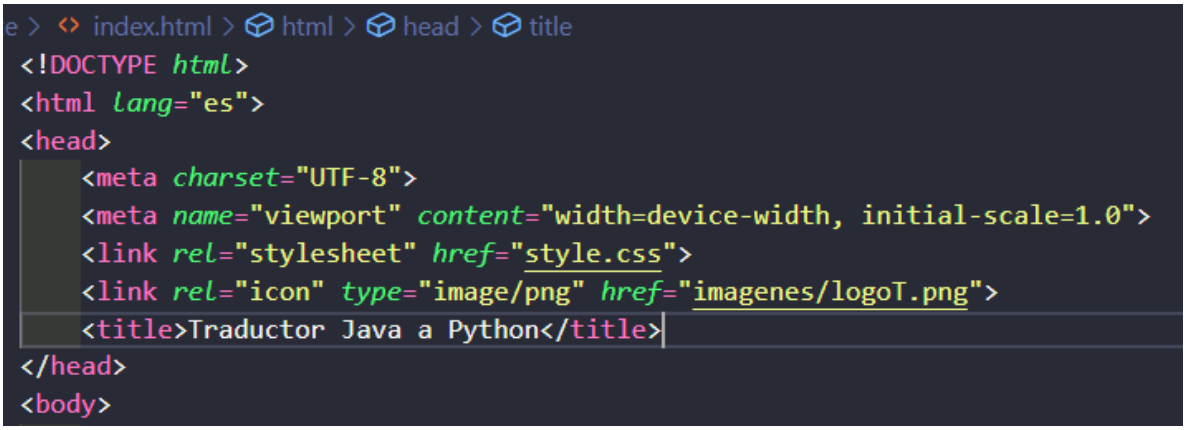
Método de análisis:

- **Token**: los tokens, son las palabras o símbolos que van a ser validos para el sistema, aquí van todas aquellas palabras reservadas que formarán parte de la lógica en el programa.
- **Analizador Léxico**: su función es leer los caracteres de entrada y generar una secuencia de componentes léxicos que serán utilizados por el analizador sintáctico. También se le conoce como lexer y se encarga de identificar tokens válidos; si encuentra un token no válido, genera un error. Analiza el léxico de la estructura, si hay símbolos o algo que el programa no identifica, los enviará como un error léxico, porque el lenguaje no lo reconoce.
- **Analizador Sintáctico:** también conocido como parser, es un programa que se utiliza para analizar la estructura gramatical de una oración o texto. Su función principal es determinar si la oración está bien construida según las reglas del lenguaje. Además, ayuda a descomponer las oraciones en sintagmas para entender mejor la función de cada elemento dentro de ellas. El parser verifica que el código esté escrito correctamente.

Interfaz gráfica

HTML

La interfaz gráfica es por medio de un archivo .html, en este caso se llama “Index.html” y desde acá es donde se realizan todas las interacciones con el usuario. Se utilizó CSS, para modificar el estilo del HTML, desde aquí se mandan las peticiones al backend (en JS) para realizar las operaciones necesarias



Menú de opciones:

Se tiene un <nav class="navbar"> para ver las opciones del sistema, acá utiliza esta estructura:

<a *href*="#" *class*="submenu-item" *id*="newFile">Nuevo

Para colocar cada una de las opciones y por medio del ID son utilizadas en el JS, mismo que despliega dinámicamente las opciones que pueden elegir:

- Nuevo: en el main, se llama por medio del ID al presionar el botón, para que por medio de un javaCodeTextarea.value = '' se limpie el textarea en donde se ingresa el código.java
- Abrir: se utiliza un openFileLink.addEventListener('click', (e) => para que pueda subirse el archivo en el textarea para poder hacer la traducción de código y javaCodeTextarea.value = event.target.result; para que se pueda ver el código en el textarea
- Traducir: una vez el código en el textarea, se llama al analizador léxico y sintáctico para que lean y analicen el código, una vez echo esto se puede traducir a Python, toda la estructura del traductor

se desarrolla y valida en el parser, si esta bien va devolviendo el código en Python.

- Analizar Tokens: llama al método main, donde se crea una tabla para los errores léxicos, sintácticos y devuelve una tabla de todos los tokens extraidos.
- Guardar como: guarda el archivo de salida en Python como archivo .py, pero para hacer esto primero tiene que haber sido posible la traducción, así que el botón aparece junto con el textarea de python.

Traducción de código

Se utilizan dos áreas de escritura para la entrada y salida de código, para poder escribir y cargar código se utiliza un textarea, pues con este se puede no solo cargar código desde el explorador, sino que también se puede modificar manualmente, en cambio para la salida en código Python se tiene un pre, que aparece únicamente hasta que se traduce el código y tiene la peculiaridad que no se puede modificar, solamente es para mostrar la salida y poder guardarla con la extensión correspondiente.

- Textarea para java: se utiliza un textarea <textarea *id*="javaCode" *placeholder*="// Escribe o pega el código Java</textarea> que permite pegar y cargar el archivo .java, cabe recalcar que este se puede editar en cualquier momento, tanto por subir archivo, como manualmente.
- Textarea para Python: esta “escondido” se utiliza un <pre *id*="pythonCode"># El código Python aparecerá aquí después de la traducción</pre> para que el código de salida aparezca únicamente cuando ya se ha traducido el código.

Lógica del programa

Para este programa se utilizó HTML, CSS y JS, el frontend se trabajaron con HTML y CSS, y el backend únicamente con JS, para hacer el programa mas ordenado y modular, se utilizaron varias carpetas para el backend y una única carpeta para el frontend, de la siguiente manera.

Estructura del programa

Proyecto 2, traductor java -> python

|-Errores

| |-Error.js(clase error para el manejo de errores léxicos y sintácticos)

|-Sintaxis

| |-Parser.js (contiene el analizador sintáctico)

|-Lexico

| |-Lexer.js (contine el analizador léxico)

|-Template (aquí se maneja el frontend de la pagina)

| |-index.html

| |-style.css

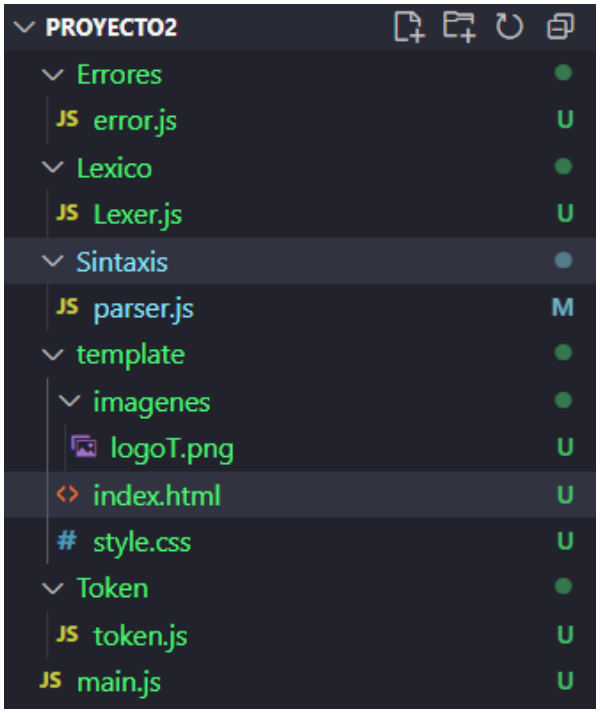
| |-imágenes

| |-logoT.png

|-Token

| |-token.js

|-main.js (maneja las llamadas entre el frontend y el backend)



Clases y funcionalidades

Token.js

Aquí se define un diccionario con los atributos válidos, lexemas, y el constructor del token.

El constructor del token defina que atributos e información lleva cada token, información que será necesaria para crear el analizador lexico.

```
export class Token{
    constructor(type, value, line, column){
        this.type = type;
        this.value = value;
        this.line = line;
        this.column = column;
    }
}
```

Diccionario para las palabras reservadas: aquí están todas las palabras que el txt usa para identificar la estructura que debe llevar el código de entrada .java.

```
export const ReservedWords = {
    "public": "PUBLIC",
    "class": "CLASS",
    "static": "STATIC",
    "void": "VOID",
    "main": "MAIN",
    "String": "STRING_TYPE",
    "args": "ARGS",
    "int": "INT_TYPE",
    "double": "DOUBLE_TYPE",
    "char": "CHAR_TYPE",
    "boolean": "BOOLEAN_TYPE",
    "true": "TRUE",
    "false": "FALSE",
    "if": "IF",
    "else": "ELSE",
    "for": "FOR",
    "while": "WHILE",
```

Diccionario de símbolos: símbolos que vienen en el txt, que son válidos, y que sirven para leer el archivo.

```
export const Symbols = {
    "{": "LLAVE_IZQ",
    "}": "LLAVE_DER",
    "(": "PAR_IZQ",
    ")": "PAR_DER",
    "[": "CORCHETE_IZQ",
    "]": "CORCHETE_DER",
    ";": "SEMICOLON",
    ",": "COMMA",
    ".": "DOT",
    "=": "EQUAL",
    "+": "PLUS",
    "-": "MINUS",
    "*": "MULTIPLY",
    "/": "DIVIDE",
    "==": "EQUAL_EQUAL",
```

Error.js

Esta clase contiene la estructura que se debe almacenar cuando se encuentre algún error léxico o sintáctico. Devolviendo el tipo de error obtenido, el valor, símbolo o el carácter que provoca el error, el mensaje de error, la línea y columna donde se encuentran los mismos.

```
export class Error{
    constructor(type, value, message, line, column){
        this.type = type;
        this.value = value;
        this.message = message;
        this.line = line;
        this.column = column;
    }
}
```

Lexico.js

Aquí se defina la clase Lexer, aquí se maneja el analizador léxico, donde se importa la clase token y error, mismas que serán utilizadas para almacenar errores y para analizar los tokens.

```
import { Token, ReservedWords } from "../Token/token.js";
import { Error } from "../Errores/error.js";

// LEXER
export class Lexer {
    // constructor
    constructor(texto) {
        this.texto = texto;
        this.pos = 0;
        this.linea = 1;
        this.columna = 1;
        this.tokens = [];
        this.errors = [];
        this.recorrido = [];
    }
```

Avanzar() determina un incremento en la columna y en la fila para seguir leyendo el texto para Analizar() utiliza la información de los tokens para analizar el texto, validando los tokens línea por línea y definiendo si hay algún error y que tipo de token son.

```
avanzar() {
    this.pos++;
    this.columna++;
}

analizar() {
    while (this.pos < this.texto.length) {
        let char = this.texto[this.pos];

        // ignora saltos de linea
        if (char === " " || char === "\t") { this.avanzar(); continue; }
        if (char === "\n") { this.linea++; this.columna = 1; this.avanzar();

        // inicializado en 0
        if (char === "/") {
            // Verificar si es un comentario de una línea '//'
            if (this.pos + 1 < this.texto.length && this.texto[this.pos + 1]
                this.recorrerComentario();
                continue;
```

Aquí en el analizador, se aplican las funciones detalladas mas adelante, para poder analizar el texto y se obtienen los errores.

Se tienen funciones para recorrer números, ID, caracteres, cadena, símbolo y comentarios.

```
recorrerNumero() {
    let inicioCol = this.columna;
    let buffer = "";
    let puntosDecimales = 0; // Contador para los puntos decimales
    let posPunto = -1; // Para recordar la posición del primer punto

    while (this.pos < this.texto.length) {
```

Parser

Este es el analizador sintáctico, aquí se va a hacer precisamente la traducción aquí se valida que el bloque de código tenga bien definida la estructura para poder ser traducido, de lo contrario muestra error y no se traduce. Primero se tiene la clase constructor, para guardar los errores sintácticos, el código java y el código de salida Python.

```
import { Error } from "../Errores/error.js";

export class Parser {
    constructor(tokens) {
        this.tokens = tokens;
        this.pos = 0;
        this.errors = [];
        this.pythonCode = "";
        this.indent = "";
        this.hasSyntaxError = false;
        this.symbolTable = new Set();
    }
```

Para lograr analizar el texto y traducirlo, se tiene una serie de métodos y funciones auxiliares, como peek, que sirve para **ver el próximo token** sin consumirlo. Es útil para tomar decisiones condicionales: por ejemplo, si el siguiente token es un (, entonces se espera una expresión.

```
//--- Métodos Auxiliares ---
peek() {
    if (this.pos >= this.tokens.length) {
    return { type: "Sintaxis", value: "Sintaxis", line: 0,
    }
    return this.tokens[this.pos];
}

match(type, value = null) {
    if (this.pos >= this.tokens.length) return false;
    const current = this.tokens[this.pos];
    if (current.type === type) {
    if (value === null || current.value === value) {
        this.pos++;
        return true;
    }
    }
    return false;
```

Hay dos métodos fundamentares que son error y analizar, en error, se manda y almacena el tipo de error obtenido, y en análisis se utilizan todas las funciones siguientes, para determinar si se puede o no hacer la traducción a código Python y si hay algún error, no se puede hacer.

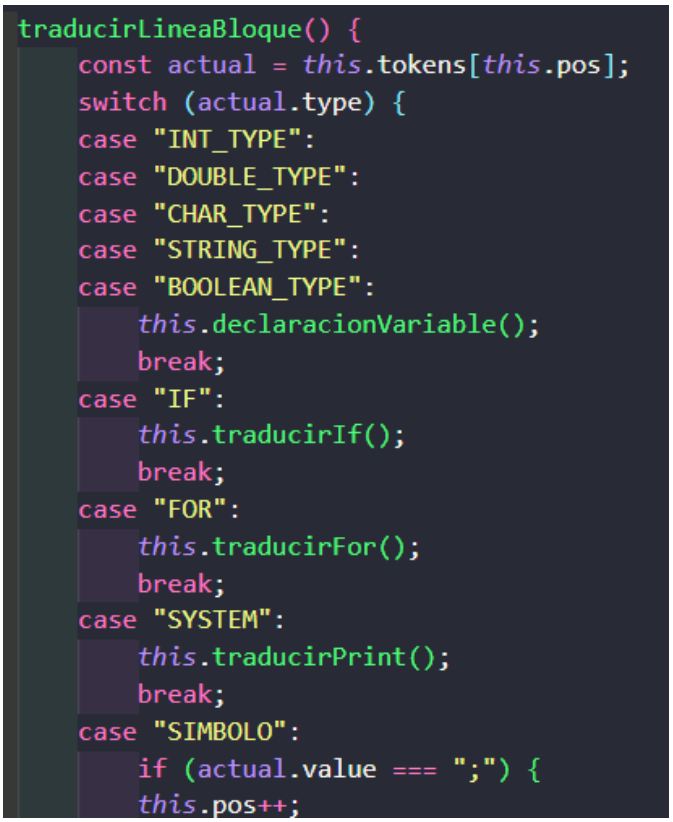
```
reportError(message, line, column) {
    this.hasSyntaxError = true;
    this.errors.push(new Error("Sintáctico", "EOF", message, line, column));
}

// lógica del parser
analizar() {
    // Validar la estructura de la clase primero
    this.validarEstructuraClase();

    if (this.hasSyntaxError) {
    return {
        errors: this.errors,
        python: "// Código con errores. No se puede traducir."
    };
    }
```

Para poder hacer este análisis, se necesitan funciones como:

1. BuscarMetodoPublicStaticVoid: que valida si existe el método public static void, propio del lenguaje java, que lo hace un bloque valido.
2. TraducirCuerpoMetodo: verifica que el bloque de código tenga corchetes de apertura y de cierrte.
3. analizarBloque: utiliza otros métodos como analizar sentencia, para ir analizando cada tipo de dato si es if, for, system, símbolo o comentario, para ir validando cada uno.
4. declaracionVariables: cerifica las variables declaradas y si están bien declaradas, si tiene el símbolo “=”, valor de asignación, punto y coma, si están en booleano, que tenga solo true o flase.
5. Traductores de condicionales: son los métodos como traducirIF(), traducirFor(), traducirPrint().
6. TraducirLineaBloque: Este se encarga de utilizando todo, ir traduciendo en si a código Python.



```
traducirLineaBloque() {
    const actual = this.tokens[this.pos];
    switch (actual.type) {
    case "INT_TYPE":
    case "DOUBLE_TYPE":
    case "CHAR_TYPE":
    case "STRING_TYPE":
    case "BOOLEAN_TYPE":
        this.declaracionVariable();
        break;
    case "IF":
        this.traducirIf();
        break;
    case "FOR":
        this.traducirFor();
        break;
    case "SYSTEM":
        this.traducirPrint();
        break;
    case "SIMBOLO":
        if (actual.value === ";") {
        this.pos++;
```

Main

Aquí se maneja toda la interacción con el usuario y la lógica del sistema primero se utiliza un DOM para inicializar comportamientos de una interfaz web una vez que el DOM está completamente cargado, odas las líneas usan document.getElementById('') para obtener referencias a elementos HTML por su id. Estas referencias se guardan en constantes para poder manipular esos elementos más adelante. Aquí obtiene el texto del textarea y el pre, entre otros.

```
document.addEventListener('DOMContentLoaded', () => {
    // Referencias a los elementos del DOM
    const javaCodeTextarea = document.getElementById('javaCode');
    const pythonCodePre = document.getElementById('pythonCode');
    const resultSection = document.getElementById('resultSection');
    const translateBtn = document.getElementById('translateBtn');
    const viewTokensBtn = document.getElementById('viewTokensBtn');
    const tokensModal = document.getElementById('tokensModal');
    const tokensReportDiv = document.getElementById('tokensReport');
    const closeTokensModal = document.getElementById('closeTokensModal');
    // Referencias para el menú "Archivo"
    const newFileLink = document.getElementById('newFile');
    const openFileLink = document.getElementById('openFile');
    const fileInput = document.getElementById('fileInput');
    // para el menú "Ayuda"
    const showHelpLink = document.getElementById('showHelp');
    const helpModal = document.getElementById('helpModal');
    const closeHelpModal = document.getElementById('closeHelpModal');
```

Se utiliza un try catch que maneja el analizador léxico y sintáctico para traducir el código y generar los reportes debidos. Si se da un error, no se traduce el código.

```
try {
    // primero análisis Léxico
    const lexer = new Lexer(codigoJava);
    const tokens = lexer.analizar();
    const erroresLexicos = lexer.errors;

    console.log("Tokens generados:", tokens);

    // segundo análisis Sintáctico
    const parser = new Parser(tokens);
    const resultadoParser = parser.analizar();
    const erroresSintacticos = resultadoParser.errors;
    const codigoPython = resultadoParser.python;

    console.log("Errores sintácticos encontrados:", erroresSintacticos);
```

Se generan las tablas de errores sintácticos, léxicos y tokens, de forma dinámica.

```
if (erroresSintacticos.length > 0) {
    reporteHTML += '<h3>🔴 Errores Sintácticos</h3>';
    reporteHTML += `
        <table border="1" style="width:100%; border-collapse: collapse; margin-bottom: 20px;">
            <thead>
                <tr style="background-color: □#444;">
                    <th style="padding: 8px; text-align: left;">#</th>
                    <th style="padding: 8px; text-align: left;">Error</th>
                    <th style="padding: 8px; text-align: left;">Descripción</th>
                    <th style="padding: 8px; text-align: left;">Línea</th>
                    <th style="padding: 8px; text-align: left;">Columna</th>
                </tr>
            </thead>
            <tbody>`;
    erroresSintacticos.forEach((error, index) => {
```

Se tienen EventListeners para subir archivo, nuevo archivo, guardar archivo y mostrar el apartado de ayuda.

```
// Funcionalidad para "Mostrar ayuda"
showHelpLink.addEventListener('click', (e) => {
    e.preventDefault(); // Evita que el enlace recargue la página
    helpModal.style.display = 'block';
});

// Funcionalidad para cerrar el modal de ayuda
closeHelpModal.addEventListener('click', () => {
    helpModal.style.display = 'none';
});

//Cerrar el modal de ayuda si se hace clic fuera de él
window.addEventListener('click', (event) => {
    if (event.target === helpModal) {
        helpModal.style.display = 'none';
    }
});
// Para pruebas iniciales, traducimos el código de ejemplo al carg
```