MANUAL TECNICO

JavaBridge - Traductor Java a Python

Por: José Fernando Ramirez Ambrocio

Carnet: 202400195

ÍNDICE

MANUAL TÉCNICO

I. Introducción:

JavaBridge es una aplicación web desarrollada en JavaScript que realiza la traducción automática de un subconjunto de código Java a Python. El sistema integra análisis léxico y sintáctico manual, generación de reportes HTML y una interfaz web completa para la gestión del proceso de traducción.

Este proyecto fue desarrollado para el curso de Lenguajes Formales y de Programación, demostrando la aplicación práctica de principios de construcción de compiladores, autómatas finitos y gramáticas formales.

II. Objetivos

Desarrollar un traductor Java a Python con análisis léxico y sintáctico manual que genere código Python ejecutable a partir de código Java básico.

III. Dirigido

Este manual técnico está dirigido a:

- **Estudiantes de lenguajes formales** que deseen entender la implementación de analizadores
- **Desarrolladores** interesados en construcción de compiladores
- **Evaluadores del curso** que requieran verificar el cumplimiento técnico
- **Mantenimiento futuro** del sistema JavaBridge

IV. Especificación técnica

IV.1 Requisitos de Hardware

- Procesador: 1 GHz o superior
- Memoria RAM: 2 GB mínimo
- Almacenamiento: 10 MB de espacio libre
- Resolución: 1024x768 px o superior

IV.1 Requisitos de Software

- Sistema Operativo: Windows 10/11, macOS, Linux
- Node.js (v18 o superior)
- npm o yarn
- MySQL Server (v8.0 o superior) o entorno compatible (XAMPP, WAMP)
- Editor de código: Visual Studio Code (recomendado)
- Navegador web moderno: Chrome, Firefox, Edge

- Git (para clonar y gestionar el repositorio)

V. Lógica del Programa

V.1 Estructura del Programa

V.2 Flujo del Programa:

- **Fase 1: Análisis Léxico (Lexer.js)**

```
1. Inicialización: pos=0, line=1, column=1
2. Por cada carácter en el texto:
   - Identificar tipo (espacio, símbolo, letra, dígito, etc.)
   - Ejecutar rutina específica según tipo
   - Generar token o error léxico
   - Actualizar posición (linea, columna)
3. Retornar: {tokens: [], errors: []}
```

- **Fase 2: Análisis Sintáctico (Parser.js)**

```
1. PROGRAMA() → Verificar: 'public' 'class' ID '{' MAIN '}'
2. MAIN() → Verificar: 'public' 'static' 'void' 'main' '(' 'String' '[' ']' 'args' ')' '{' SENTENCIA
S '}'
3. SENTENCIAS() → Procesar cada sentencia hasta '}'
4. SENTENCIA() → DECLARACION | ASIGNACION | PRINT | ';'
5. Construcción incremental del AST
```

- **Fase 3: Traducción (Traductor.js)**

```
1. Traducir estructura principal (clase, main)
2. Por cada sentencia en el AST:
   - Aplicar regla de traducción específica
   - Manejar conversiones de tipos y sintaxis
   - Generar código Python equivalente
3. Formatear salida con comentarios y estructura
```

- **Fase 4: Generación de Reportes (app.js)**

```
1. Tabla de Tokens: Lista todos los tokens reconocidos
2. Tabla Errores Léxicos: Errores de caracteres inválidos
3. Tabla Errores Sintácticos: Errores de estructura gramatical
4. Actualización en tiempo real de la interfaz
```

VI. Clases y Métodos

VI.1 Componentes Principales

VI.1.1 Modelo (Backend / API)

- Clase Lexer:

```
class Lexer {
    constructor(texto)
    analizar() → {tokens: Token[], errors: Error[]}
    recorrerIdentificador() → void
    recorrerNumero() → void
    recorrerCadena() → void
    recorrerCaracter() → void
    procesarSimbolo() → boolean
    agregarErrorLexico() → void
}
```

- Clase Parser:

```
class Parser {
    constructor(tokens)
    analizar() → {ast: Object, errors: Error[]}
    PROGRAMA() → Object
    MAIN() → Object
    SENTENCIAS() → Object[]
    SENTENCIA() → Object
    DECLARACION() → Object
    ASIGNACION() → Object
    PRINT() → Object
    EXPRESION() → Object
}
```

- Clase Traductor:

```
class Traductor {
    constructor()
    traducir(ast) → {codigo: string, errores: Error[]}
    traducirMain(main) → void
    traducirSentencias(sentencias) → void
    traducirSentencia(sentencia) → void
    traducirExpresion(expresion) → string
}
```

- Clase Token:

```
class Token {
    constructor(type, value, line, column)
    // Propiedades: type, value, line, column
}
```

- Clase Error:

```
class Error {
    constructor(tipo, valor, descripcion, linea, columna)
    // Propiedades: tipo, valor, descripcion, linea, columna
}
```

VI.1.2 Métodos Principales

- **Métodos de Control (app.js)**

```
// Gestión del proceso completo
generarTraduccion() → void
verTokens() → void

// Manejo de archivos
abrirArchivo() → void
guardarJava() → void
guardarPython() → void

// Generación de reportes
generarReporteTokens(tokens) → void
mostrarErroresLexicos(errores) → void
mostrarErroresSintacticos(errores) → void

// Utilidades
actualizarContadorTokens(cantidad) → void
limpiarReportes() → void
escapeHtml(texto) → string
```

- **Gramática Implementada (BNF)**

```
<PROGRAMA>  ::= "public" "class" <IDENTIFICADOR> "{" <MAIN> "}"
<MAIN>      ::= "public" "static" "void" "main" "(" "String" "[" "]" "args" ")" "{" <SENTENCIAS> "}"
<SENTENCIAS> ::= <SENTENCIA> <SENTENCIAS> | ε
<SENTENCIA> ::= <DECLARACION> | <ASIGNACION> | <PRINT> | ";"
<DECLARACION> ::= <TIPO> <IDENTIFICADOR> ("=" <EXPRESION>)? ";"
<ASIGNACION> ::= <IDENTIFICADOR> "=" <EXPRESION> ";"
<PRINT>     ::= "System" "." "out" "." "println" "(" <EXPRESION> ")" ";"
<EXPRESION> ::= <TERMINO> (<OPERADOR> <TERMINO>)*
<TERMINO>   ::= <IDENTIFICADOR> | <LITERAL> | "(" <EXPRESION> ")"
<TIPO>      ::= "int" | "double" | "String" | "char" | "boolean"
```

VII. Conclusiones

Logros Alcanzados

- Implementación exitosa de analizador léxico mediante AFD manual
- Parser recursivo funcional para gramática Java básica
- Sistema de traducción Java→Python operativo
- Generación de reportes HTML completos
- Interfaz web integrada y responsive