

Manual Técnico - JavaBridge

Proyecto 2 - Lenguajes Formales y de Programación

Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Información del Proyecto

- Nombre: JavaBridge - Traductor de Java a Python
 - Estudiante: Christian Javier Rivas Arreaga
 - Carnet: 202303204
 - Curso: Lenguajes Formales y de Programación
 - Sección: B
 - Semestre: Segundo Semestre 2025
 - Fecha: Octubre 2025
-

1. Introducción

1.1 Descripción del Proyecto

JavaBridge es un traductor automático de código Java a Python que implementa análisis léxico y sintáctico de forma manual (sin uso de expresiones regulares ni librerías de parsing). El sistema procesa un subconjunto específico de Java y genera código Python equivalente.

1.2 Objetivos

- Implementar un AFD (Autómata Finito Determinista) para reconocimiento de tokens
- Desarrollar un parser manual mediante gramática libre de contexto
- Traducir constructos de Java a Python preservando la semántica
- Generar reportes HTML profesionales
- Proporcionar una interfaz web moderna y funcional

1.3 Tecnologías Utilizadas

Backend:

- Node.js v18+
- Express 4.18.2

- CORS 2.8.5

Frontend:

- Vue.js 3.4.21
- Vue Router 4.3.0
- Tailwind CSS 3.4.1
- Axios 1.6.7
- Vite 5.1.4

2. Análisis Léxico

2.1 Autómata Finito Determinista (AFD)

El analizador léxico está implementado en `backend/src/lexer/Lexer.js` mediante un AFD de 37 estados que reconoce todos los tokens del subconjunto de Java.

2.1.1 Tabla de Tokens

Token	Patrón	Descripción
PUBLIC	<code>public</code>	Palabra reservada
CLASS	<code>class</code>	Palabra reservada
STATIC	<code>static</code>	Palabra reservada
VOID	<code>void</code>	Palabra reservada
MAIN	<code>main</code>	Palabra reservada
STRING	<code>String</code>	Palabra reservada
ARGS	<code>args</code>	Palabra reservada
INT	<code>int</code>	Tipo de dato
DOUBLE	<code>double</code>	Tipo de dato
CHAR	<code>char</code>	Tipo de dato
BOOLEAN	<code>boolean</code>	Tipo de dato
TRUE	<code>true</code>	Literal booleano

FALSE	false	Literal booleano
IF	if	Estructura de control
ELSE	else	Estructura de control
FOR	for	Estructura de control
WHILE	while	Estructura de control
SYSTEM	System	Palabra reservada
OUT	out	Palabra reservada
PRINTLN	println	Palabra reservada
IDENTIFICADOR	[A-Za-z_][A-Za-z0-9_]*	Identificador de variable
NUMERO_ENTERO	[0-9]+	Literal numérico entero
NUMERO_DECIMAL	[0-9]+\.[0-9]+	Literal numérico decimal
CADENA	"..."	Literal de cadena
CARACTER	'.'	Literal de carácter
LLAVE_IZQ	{	Símbolo
LLAVE_DER	}	Símbolo
PAREN_IZQ	(Símbolo
PAREN_DER)	Símbolo
CORCHETE_IZQ	[Símbolo
CORCHETE_DER]	Símbolo
PUNTO_COMA	;	Símbolo
COMA	,	Símbolo
PUNTO	.	Símbolo
IGUAL	=	Operador de asignación
MAS	+	Operador aritmético

MENOS	-	Operador aritmético
POR	*	Operador aritmético
DIVIDE	/	Operador aritmético
IGUALDAD	==	Operador relacional
DIFERENTE	!=	Operador relacional
MAYOR	>	Operador relacional
MENOR	<	Operador relacional
MAYOR_IGUAL	>=	Operador relacional
MENOR_IGUAL	<=	Operador relacional
INCREMENTO	++	Operador unario
DECREMENTO	--	Operador unario

2.1.2 Diagrama de Estados del AFD

Ver archivo `docs/AFD_Diagrama.dot` para el diagrama completo en formato Graphviz.

Estados principales:

- S0: Estado inicial
- S1-S3: Reconocimiento de identificadores y palabras reservadas
- S4-S6: Reconocimiento de números (enteros y decimales)
- S7-S8: Reconocimiento de cadenas
- S9-S10: Reconocimiento de caracteres
- S11-S36: Reconocimiento de símbolos y operadores
- S17-S20: Reconocimiento de comentarios

2.1.3 Explicación de Estados Clave

Estado S0 (Inicial):

- Punto de entrada del autómata
- Clasifica el primer carácter para determinar la transición
- Ignora espacios en blanco, tabs y saltos de línea

Estados S1-S3 (Identificadores):

- S1: Reconoce el primer carácter (letra o underscore)

- S2: Acumula caracteres alfanuméricos
- S3: Estado final, verifica si es palabra reservada

Estados S4-S6 (Números):

- S4: Reconoce dígitos para números enteros
- S5: Detecta punto decimal
- S6: Acumula decimales para números flotantes

Estados S7-S8 (Cadenas):

- S7: Detecta comilla doble de apertura
- S8: Acumula caracteres hasta comilla de cierre
- Maneja escape de caracteres especiales

Estados S17-S20 (Comentarios):

- S17: Detecta / inicial
- S18: Comentario de línea //
- S19-S20: Comentario de bloque /* */

2.2 Implementación del Lexer

```
class Lexer {
  constructor(input) {
    this.input = input;
    this.position = 0;
    this.line = 1;
    this.column = 1;
    this.tokens = [];
    this.errors = [];
  }

  analyze() {
    while (this.position < this.input.length) {
      this.#S0(); // Estado inicial
    }
    this.tokens.push(new Token('EOF', '', this.line, this.column));
    return {
      tokens: this.tokens,
      errors: this.errors
    };
  }
}
```

Método de transición de estados:

- Cada estado está implementado como un método privado `#S{número}()`

- Los métodos verifican el carácter actual y determinan la siguiente transición
- Se utiliza `CharacterUtils` para validar tipos de caracteres sin regex

3. Análisis Sintáctico

3.1 Gramática Libre de Contexto (BNF)

```
<PROGRAMA> ::= <CLASE>
```

```
<CLASE> ::= "public" "class" IDENTIFICADOR "{" <METODO_MAIN> "}"
```

```
<METODO_MAIN> ::= "public" "static" "void" "main" "(" "String" "[" "]" "args"
")" "{" <SENTENCIAS> "}"
```

```
<SENTENCIAS> ::= <SENTENCIA> <SENTENCIAS> | ε
```

```
<SENTENCIA> ::= <DECLARACION>
                | <ASIGNACION>
                | <IMPRESION>
                | <IF_ELSE>
                | <FOR>
                | <WHILE>
                | <INCREMENTO>
                | <DECREMENTO>
```

```
<DECLARACION> ::= <TIPO> IDENTIFICADOR ";"
                | <TIPO> IDENTIFICADOR "=" <EXPRESION> ";"
```

```
<TIPO> ::= "int" | "double" | "char" | "String" | "boolean"
```

```
<ASIGNACION> ::= IDENTIFICADOR "=" <EXPRESION> ";"
```

```
<EXPRESION> ::= <TERMINO> <EXPR_PRIMA>
```

```
<EXPR_PRIMA> ::= <OP_SUMA> <TERMINO> <EXPR_PRIMA> | ε
```

```
<TERMINO> ::= <FACTOR> <TERMINO_PRIMA>
```

```
<TERMINO_PRIMA> ::= <OP_MULT> <FACTOR> <TERMINO_PRIMA> | ε
```

```
<FACTOR> ::= IDENTIFICADOR
                | NUMERO_ENTERO
                | NUMERO_DECIMAL
                | CADENA
                | CARACTER
                | "true"
                | "false"
                | "(" <EXPRESION> ")"
```

```
<OP_SUMA> ::= "+" | "-"
```

`<OP_MULT> ::= "*" | "/"`

`<OP_RELACIONAL> ::= "==" | "!=" | ">" | "<" | ">=" | "<="`

`<CONDICION> ::= <EXPRESION> <OP_RELACIONAL> <EXPRESION>`

`<IMPRESION> ::= "System" "." "out" "." "println" "(" <EXPRESION> ")" ";"`

`<IF_ELSE> ::= "if" "(" <CONDICION> ")" "{" <SENTENCIAS> "}" <ELSE_OPCIONAL>`

`<ELSE_OPCIONAL> ::= "else" "{" <SENTENCIAS> "}" | ϵ`

`<FOR> ::= "for" "(" <FOR_INIT> ";" <CONDICION> ";" <FOR_UPDATE> ")" "{" <SENTENCIAS> "}"`

`<FOR_INIT> ::= <TIPO> IDENTIFICADOR "=" <EXPRESION>`

`<FOR_UPDATE> ::= IDENTIFICADOR "++" | IDENTIFICADOR "--"`

`<WHILE> ::= "while" "(" <CONDICION> ")" "{" <SENTENCIAS> "}"`

`<INCREMENTO> ::= IDENTIFICADOR "++" ";"`

`<DECREMENTO> ::= IDENTIFICADOR "--" ";"`

3.2 Explicación de Producciones

3.2.1 <PROGRAMA>

- Producción raíz de la gramática
- Todo programa Java válido debe contener exactamente una clase
- Ejemplo: `public class MiClase { ... }`

3.2.2 <CLASE>

- Define la estructura de una clase Java
- Debe tener modificador `public`, palabra clave `class`, un identificador y un método `main`
- Ejemplo: `public class Calculadora { ... }`

3.2.3 <METODO_MAIN>

- Método principal con firma obligatoria
- Firma exacta: `public static void main(String[] args)`
- Contiene todas las sentencias del programa
- Ejemplo: `public static void main(String[] args) { int x = 5; }`

3.2.4 <SENTENCIAS>

- Secuencia de cero o más sentencias
- Puede ser vacía (producción ϵ)
- Ejemplo: `int x = 5; x = x + 1; System.out.println(x);`

3.2.5 <DECLARACION>

- Define una variable con o sin inicialización
- Tipos soportados: int, double, char, String, boolean
- Ejemplo: `int numero = 10; 0 double pi;`

3.2.6 <EXPRESION>

- Evaluación de operaciones aritméticas
- Respetar precedencia de operadores (* / antes que + -)
- Ejemplo: `a + b * c` se evalúa como `a + (b * c)`

3.2.7 <CONDICION>

- Expresión booleana para estructuras de control
- Usa operadores relacionales: ==, !=, >, <, >=, <=
- Ejemplo: `x > 5, a == b`

3.2.8 <IF_ELSE>

- Estructura condicional con else opcional
- Ejemplo: `if (x > 0) { ... } else { ... }`

3.2.9 <FOR>

- Bucle con inicialización, condición y actualización
- Ejemplo: `for (int i = 0; i < 10; i++) { ... }`

3.2.10 <WHILE>

- Bucle condicional
- Ejemplo: `while (x > 0) { x--; }`

3.3 Implementación del Parser

```
class Parser {
    constructor(tokens) {
        this.tokens = tokens;
        this.current = 0;
        this.errors = [];
        this.symbolTable = new Map();
    }

    parse() {
        try {
            const ast = this.#parseProgram();
            return {
                success: this.errors.length === 0,
                ast: ast,
                errors: this.errors,
                symbolTable: this.symbolTable
            };
        } catch (error) {
            return {
```



```

        success: false,
        ast: null,
        errors: this.errors,
        symbolTable: this.symbolTable
    };
}
}
}

```

Métodos principales:

- `#parseProgram()`: Producción raíz
- `#parseClass()`: Valida estructura de clase
- `#parseMainMethod()`: Valida método main
- `#parseStatement()`: Despacha a parsers específicos
- `#parseExpression()`: Maneja expresiones aritméticas
- `#parseCondition()`: Procesa condiciones booleanas

4. Traducción Java → Python

4.1 Reglas de Traducción

4.1.1 Tipos de Datos

Java	Python	Valor por Defecto
<code>int</code>	<code>int</code>	<code>0</code>
<code>double</code>	<code>float</code>	<code>0.0</code>
<code>char</code>	<code>str</code>	<code>' '</code>
<code>String</code>	<code>str</code>	<code>""</code>
<code>boolean</code>	<code>bool</code>	<code>False</code>

4.1.2 Declaraciones

Java:

```

int numero = 5;
double pi = 3.14;

```

Python:

```

numero = 5
pi = 3.14

```

4.1.3 Estructuras de Control

IF-ELSE:

Java:

```
if (x > 0) {  
    System.out.println("Positivo");  
} else {  
    System.out.println("Negativo");  
}
```

Python:

```
if x > 0:  
    print("Positivo")  
else:  
    print("Negativo")
```

FOR:

Java:

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

Python:

```
for i in range(0, 10):  
    print(i)
```

WHILE:

Java:

```
while (x > 0) {  
    x--;  
}
```

Python:

```
while x > 0:  
    x -= 1
```

4.1.4 Impresión

Java:

```
System.out.println(variable);
```

Python:

```
print(variable)
```

4.2 Implementación del Traductor

```
class Translator {
  constructor(ast) {
    this.ast = ast;
    this.indentLevel = 0;
  }

  translate() {
    return this.#translateMain(this.ast.mainMethod);
  }

  #indent() {
    return ' '.repeat(this.indentLevel);
  }
}
```

5. Reportes HTML

5.1 Reporte de Tokens

Genera tabla HTML con:

- Número de token
- Tipo de token
- Valor (lexema)
- Línea y columna

Archivo: backend/src/reports/ReportGenerator.js

5.2 Reporte de Errores Léxicos

Muestra:

- Carácter no reconocido
- Ubicación (línea, columna)
- Descripción del error

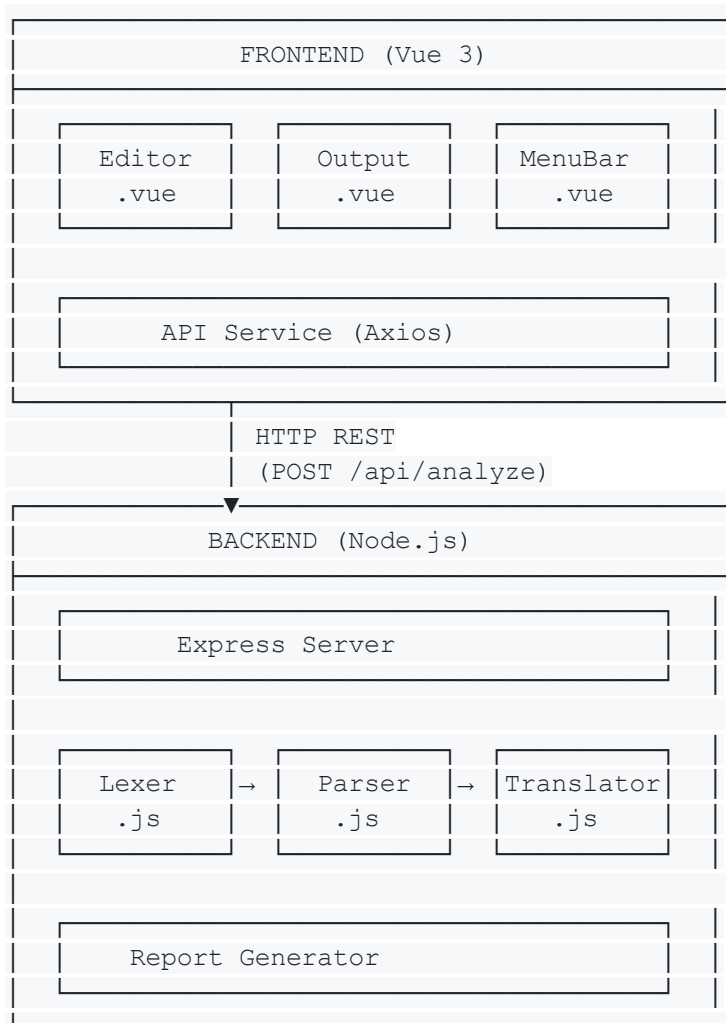
5.3 Reporte de Errores Sintácticos

Detalla:

- Token inesperado
- Token esperado
- Ubicación del error
- Descripción contextual

6. Arquitectura del Sistema

6.1 Diagrama de Componentes



6.2 Flujo de Procesamiento

1. Entrada: Usuario escribe/carga código Java
 2. Análisis Léxico: Tokenización mediante AFD
 3. Análisis Sintáctico: Validación de estructura
 4. Traducción: Generación de código Python
 5. Salida: Código Python o reportes de error
-

7. Pruebas Realizadas

7.1 Casos de Prueba Válidos

Caso 1: Declaraciones básicas

```
public class Test {  
    public static void main(String[] args) {  
        int x = 5;  
        double y = 3.14;  
        String nombre = "Juan";  
    }  
}
```

Caso 2: Estructuras de control

```
public class Test {  
    public static void main(String[] args) {  
        int x = 10;  
        if (x > 5) {  
            System.out.println("Mayor");  
        }  
        for (int i = 0; i < 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

7.2 Casos de Prueba Inválidos

Caso 1: Errores léxicos

```
public class Test {  
    public static void main(String[] args) {  
        int x = 5; @#$  
    }  
}
```

Resultado: Caracteres @, #, \$ no reconocidos







Caso 2: Errores sintácticos

```
public class Test {  
    public static void main(String[] args) {  
        int x = 5 // Falta punto y coma  
    }  
}
```

Resultado: Se esperaba ;

8. Conclusiones

8.1 Logros Obtenidos

-  Implementación exitosa de AFD con 37 estados
-  Parser recursivo descendente funcional
-  Traducción correcta de Java a Python
-  Reportes HTML profesionales
-  Interfaz web moderna y responsiva
-  Sin uso de regex ni librerías de parsing

8.2 Limitaciones

- Solo soporta el subconjunto de Java especificado
- No maneja clases múltiples ni herencia
- Arrays limitados (solo en firma de main)
- No soporta operadores lógicos (&&, ||, !)

8.3 Recomendaciones

- Ampliar gramática para soportar más constructos
- Implementar optimizaciones en el traductor
- Agregar más validaciones semánticas
- Mejorar mensajes de error

9. Referencias

- Compiladores: Principios, Técnicas y Herramientas - Aho, Lam, Sethi, Ullman
- Documentación oficial de Vue.js: <https://vuejs.org/>
- Documentación oficial de Express: <https://expressjs.com/>
- Documentación oficial de Node.js: <https://nodejs.org/>

Elaborado por: Christian Javier Rivas Arreaga

Carnet: 202303204

Fecha: Octubre 2025