Selvin Raúl Chuquiej Andrade

202405516

LFP B+

MANUAL TECNICO

Proyecto2

Descripción general del funcionamiento:

El analizador léxico inicia en el estado S0, que representa el estado inicial del AFD.

Dependiendo del tipo de carácter leído (letra, dígito, símbolo, comilla, etc.), realiza una transición hacia otro estado.

Cada estado tiene un propósito específico:

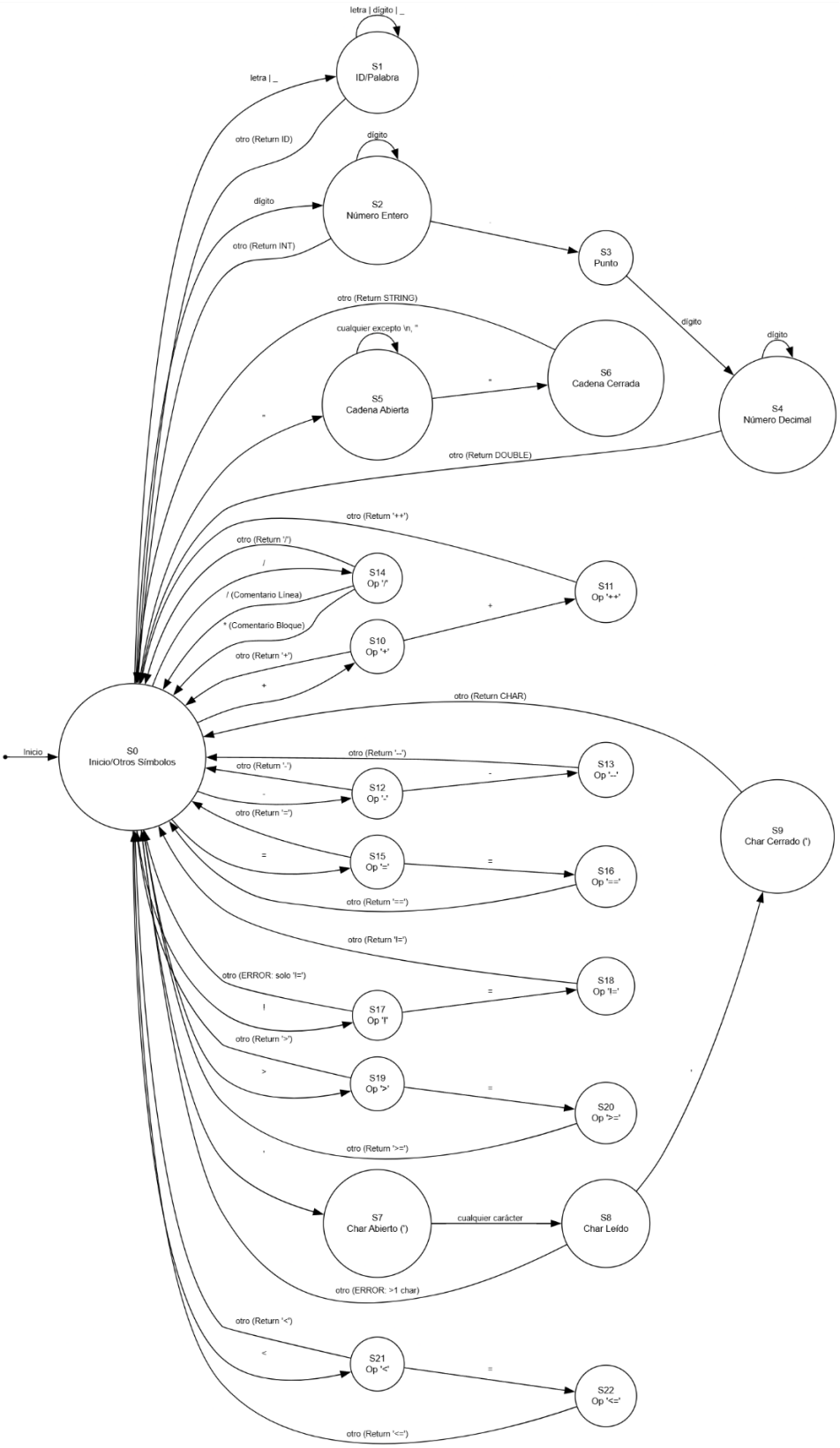
Estado	Descripción
S0	Estado inicial. Ignora espacios, salta comentarios, y decide qué tipo de token analizar.
S1	Identifica identificadores y palabras reservadas (public, class, int, etc.).
S2-S4	Reconocen números enteros y decimales.
S5-S6	Reconocen cadenas de texto ("Hola"). Detectan errores si no se cierran.
S7-S9	Reconocen caracteres ('a'). Detectan errores si están mal formados
S10-S13	Manejan operadores +, ++, -, --.
S14	Detecta operador / o comentarios (//, /* ... */).
S15-S22	Reconocen operadores relacionales y de asignación (=, ==, !=, >, >=, <, <=).

Tabla de Tokens:

Tipo	Ejemplo	Patrón	Descripción
PUBLIC	public	public	Palabra reservada para definir visibilidad.
CLASS	class	class	Palabra reservada para declarar clases.
STATIC	static	static	Define miembros estáticos.
VOID	void	void	Tipo de retorno vacío.
MAIN	main	main	Nombre obligatorio del método principal.

STRING_TYPE	String	String	Tipo de dato cadena.
ARGS	args	args	Nombre del arreglo de argumentos.
INT_TYPE	int	int	Tipo de dato entero.
DOUBLE_TYPE	double	double	Tipo de dato decimal.
CHAR_TYPE	char	char	Tipo de dato carácter.
BOOLEAN_TYPE	boolean	boolean	Tipo de dato lógico.
TRUE / FALSE	true / false	`true	false`
IF / ELSE	if / else	`if	else`
FOR / WHILE	for / while	`for	while`
SYSTEM, OUT, PRINTLN	System.out.println	System\.out\.println	Sentencia de impresión.
IDENTIFIER	variable1	[A-Za-z_][A-Za-z0-9_]*	Identificador de variable o clase.
INTEGER	123	[0-9]+	Números enteros.
DECIMAL	12.34	[0-9]+\.[0-9]+	Números con parte fraccionaria.
STRING	“hola”	"(.)*"	Cadenas entre comillas dobles.
CHAR	‘a’	'.'	Carácter entre comillas simples.
BOOLEAN	True, false	`true	false`
LLAVE_IZQ / LLAVE_DER	{, }	`{	}`
PAR_IZQ / PAR_DER	(,)	`(	)`
CORCHETE_IZQ / CORCHETE_DER	[.]	`[	]`
SEMICOLON	;	;	Fin de instrucción.
COMMA	,	,	Separador.
DOT	.	.	Acceso a miembros.
EQUAL	=	=	Asignación.
EQUAL_EQUAL / NOT_EQUAL	==, !=	`==	!=`
GREATER / LESS / GREATER_EQUAL / LESS_EQUAL	>, <, >=, <=	`>	<
PLUS / MINUS / MULTIPLY / DIVIDE	+, -, *, /	[+\-*/]	Operadores aritméticos.
INCREMENT / DECREMENT	++, --	`++	--`

AFD:


letra | dígito | _
S1
ID/Palabra
letra | _
otro (Return ID)
dígito
S2
Número Entero
dígito
otro (Return INT)
.
S3
Punto
otro (Return STRING)
cualquier excepto \n, "
dígito
dígito
S6
Cadena Cerrada
S5
Cadena Abierta
"
"
S4
Número Decimal
otro (Return DOUBLE)
otro (Return '++')
otro (Return '/')
/
/ (Comentario Línea)
* (Comentario Bloque)
otro (Return '+')
+
S14
Op '/'
S11
Op '++'
+
S10
Op '+'
otro (Return CHAR)
Inicio
S0
Inicio/Otros Símbolos
otro (Return '--')
otro (Return '-')
-
otro (Return '=')
S13
Op '--'
S12
Op '-'
S9
Char Cerrado (')
=
S15
Op '='
=
S16
Op '=='
otro (Return '==')
otro (Return '!=')
otro (ERROR: solo '!=')
!
S17
Op '!'
=
S18
Op '!='
otro (Return '>')
>
S19
Op '>'
=
S20
Op '>='
otro (Return '>=')
'
S7
Char Abierto (')
cualquier carácter
S8
Char Leído
otro (ERROR: >1 char)
otro (Return '<')
<
S21
Op '<'
=
S22
Op '<='
otro (Return '<=')

Utilizacion de un parser descendente recursivo, también conocido como recursive-descent parser. Cada no terminal de la gramática se implementa como una función JavaScript (parseProgram, parseMainMethod, parseStatement, etc.), lo cual refleja directamente las reglas de la gramática.

Funcionamiento general

1. Entrada: una lista de tokens producida por el analizador léxico.
2. Salida:
    - Si el código es válido → un árbol sintáctico (AST).
    - Si hay errores → una lista de errores sintácticos con línea y columna.
3. Método de análisis:
    - El parser verifica recursivamente la estructura:
        - Clase principal
        - Método main
        - Sentencias dentro del cuerpo del main

Gramática tipo 2 (BNF)

```
// PROGRAMA COMPLETO
<PROGRAMA> ::= "public" "class" <IDENTIFICADOR> "{" <MAIN> "}"

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

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

<SENTENCIA> ::= <DECLARACION>
              | <ASIGNACION>
              | <IF>
              | <WHILE>
              | <FOR>
              | <PRINT>
              | <BLOQUE>
              | <EXPR_STMT>
              | ";"

<BLOQUE> ::= "{" <SENTENCIAS> "}"

<DECLARACION> ::= <TIPO> <LISTA_VARS> ";"

<LISTA_VARS> ::= <VAR_DECL> ("," <VAR_DECL>)*

<VAR_DECL> ::= <IDENTIFICADOR> ("=" <EXPRESION>)?

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

<IF> ::= "if" "(" <EXPRESION> ")" <SENTENCIA> ("else" <SENTENCIA>)?

<WHILE> ::= "while" "(" <EXPRESION> ")" <SENTENCIA>

<FOR> ::= "for" "(" <FOR_INIT> ";" <EXPRESION> ";" <FOR_UPDATE> ")" <SENTENCIA>

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

<FOR_UPDATE> ::= <IDENTIFICADOR> ("++" | "--")

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

<EXPR_STMT> ::= <EXPRESION> ";"

<EXPRESION> ::= <TERMINO> (("==" | "!=" | ">" | "<" | ">=" | "<=") <TERMINO>)*

<TERMINO> ::= <FACTOR> (("+" | "-") <FACTOR>)*

<FACTOR> ::= <PRIMARIO> (("*" | "/") <PRIMARIO>)*

<PRIMARIO> ::= <IDENTIFICADOR>
             | <LITERAL>
             | "(" <EXPRESION> ")"
             | <UNARY_PRE>
             | <UNARY_POST>

<UNARY_PRE> ::= ("++" | "--") <IDENTIFICADOR>

<UNARY_POST> ::= <IDENTIFICADOR> ("++" | "--")

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

<LITERAL> ::= <ENTERO> | <DECIMAL> | <CARACTER> | <CADENA> | <BOOLEANO>

<IDENTIFICADOR> ::= [a-zA-Z_][a-zA-Z0-9_]*

<ENTERO> ::= [0-9]+

<DECIMAL> ::= [0-9]+ "." [0-9]+

<CARACTER> ::= "'" [^'] "'"

<CADENA> ::= '"' [^"]* '"'

<BOOLEANO> ::= "true" | "false"
```

Conclusiones:

- La creación manual del analizador léxico (con autómatas) y sintáctico (con gramáticas) permitió comprender de manera tangible los fundamentos de funcionamiento de los compiladores, llevando la teoría a la práctica.
- El proyecto materializó los conceptos de análisis sintáctico al construir un traductor operativo que convierte código Java a Python, evidenciando cómo mantener el significado del programa entre ambos lenguajes.
- La organización del sistema en componentes independientes (lexer, parser, manejo de errores) resultó en un código más legible y fácil de mantener, gracias a una arquitectura bien definida.