MANUEL TECNICO

Escaner

El escáner esta distribuido en mas de una clase, siendo la principal la de análisisLexico, las clases ubicadas en esta sección son:

Token: Sirve para la creación de tokens que serán usados dentro del análisis Lexico.

Error: Sirve para la creación de errores que puedan presentarse si en el análisis léxico algún caracter no cumpla con las condiciones para ser considerado token.

AnalisisLexico:

En este sección se encuentra todo el análisis léxico del traductor, siendo esta la primera fase para encontrar errores y enviar los tokens que si sean validos para el programa.

Las palabras reservadas que si se toman en cuenta para el análisis son:

public, class, static, void, main, String, args, int, double, char, boolean, true, false, if, else, for, while, System, out, println

SÍMBOLOS: {, }, (,), [,], ;, ,, ., =, +, -, *, /, ==, !=, >, =, <=, ++, --

IDENTIFICADORES:

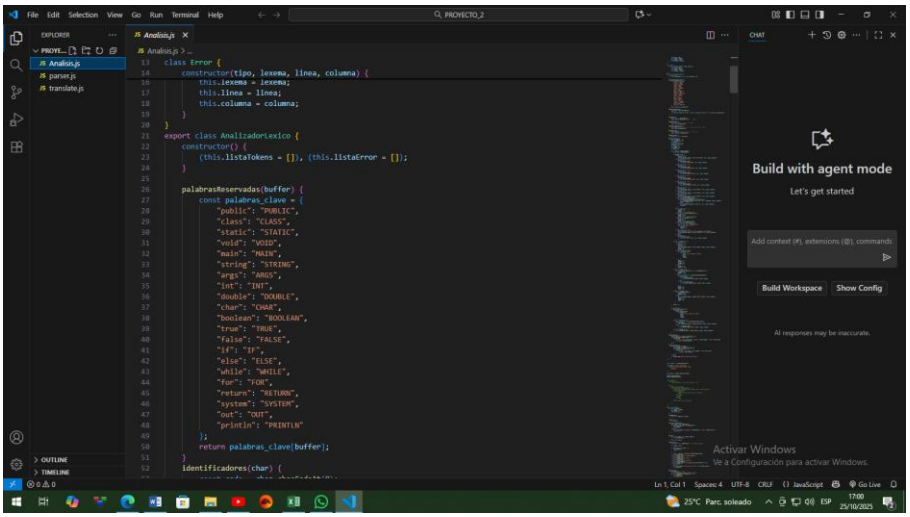
Patrón: [A-Za-z_][A-Za-z0-9_]* Ejemplos válidos: variable, _temp, contador1, miVariable Ejemplos inválidos: 1variable, class, for

LITERALES:

Enteros: 123, 0, -45 Decimales: 12.34, 0.0, -3.14 Caracteres: 'a', '1', ' ' Cadenas: "hola", "", "texto con espacios" Booleanos: true, false

COMENTARIOS: Línea: // texto hasta fin de línea Bloque: /* texto multilínea */

ESPACIOS EN BLANCO: Espacios, tabs, saltos de línea son ignorados excepto dentro de literales.



Muestra del código dentro del análisisLexico

```
export class AnalizadorLexico {
    palabrasReservadas(buffer) {
        const palabras_clave = {
        };
        return palabras_clave[buffer];
    }
    identificadores(char) {
        const code = char.charCodeAt(0);
        return (
            (code >= 65 && code <= 90) || (code >= 97 && code <= 122) || "áéíóúÁÉÍÓÚñÑ".includes(char)
        );
    }

    esDigito(c) {
        const code = c.charCodeAt(0);
        return code >= 48 && code <= 57;     // 0-9
    }
    operadores(char) {
        const ops = ["+", "-", "*", "/", "="];
        return ops.includes(char);
    }
    esOperadorCompuesto(char) {
        const ops2 = ["==", "!=", "<", ">", "<=", ">=", "++", "--"];
        return ops2.includes(char);
    }
    esSimbolo(char) {
        const simbolos = ["{", "}", "(", ")", "[", "]"]
        return simbolos.includes(char);
    }
```

Los identificadores contienen todos los caracteres en código ASCII del 65 al 90 para tomar en cuenta todas las letras del abecedario, además se incluye algunos caracteres especiales.

esDigito: aquí se toman en cuenta los valores de números reales desde el 0 al 9

operadores: los operadores esenciales para realizar operaciones aritméticas.

esOperadoresCompuesto: aquí se tomaran en cuenta todos los operadores especiales que sirven para denotar igualdades, desigualdades, comparaciones, etc.

esSimbolo: en esSimbolo se toman en cuenta las llaves, corchetes y paréntesis que están presentes en el código.

Función analizar

```
analizar(entrada) {
    this.listaTokens = [];
    this.listaError = [];
    let buffer = "";
    const centinela = "#";
    entrada += centinela;
    let linea = 1;
    let columna = 1;
    let estado = 0;
    let index = 0;
```

Dentro de la función analizar tendremos dos array para almacenar tokens y errores, el buffer que incrementara con los datos obtenidos y definimos las líneas y columnas para identificar de mejor

manera las posiciones de los tokens, en estado igualaremos a cero para que este pueda modificarse una vez se cumplan ciertas restricciones.

Iniciando un while que tomara el índice en cero, y verificara que este sea menor a la longitud de la entrada, dentro de el tendremos un char que almacenara el índice de la entrada para poder visualizar los datos en la entrada y estos puedan ser analizados.

Una vez definido estos valores, podremos iniciar el análisis del estado dentro de la entrada con un if que evaluara cuando este se encuentre en 0.

Dentro del if evaluaremos primero se dentro del texto se encuentra una llave que abre y cierra para agregar estos datos como tokens en la clase Token, siguiendo con corchetes que abren y cierran, luego paréntesis que abren y cierran.

```
while (index < entrada.length) {
    const char = entrada[index];

    if (estado === 0) {
        if (char === "{") {
            columna += 1;
            this.listaTokens.push(new Token("LLAVE_ABRE", char, linea, columna));
        } else if (char === "}") {
            columna += 1;
            this.listaTokens.push(
                new Token("LLAVE_CIERRA", char, linea, columna)
            );
        } else if (char === "[") {
            columna += 1;
            this.listaTokens.push(
                new Token("CORCHETE_ABRE", char, linea, columna)
            );
        } else if (char === "]") {
            columna += 1;
            this.listaTokens.push(
                new Token("CORCHETE_CIERRA", char, linea, columna)
            );
        } else if (char == "(") {
            columna += 1;
            this.listaTokens.push(
                new Token("PARENTESIS_ABRE", char, linea, columna)
            );
        } else if (char == ")") {
            columna += 1;
```

La siguiente parte del código verificaremos si se encuentran “;”, “:”,”,”,” como símbolos en donde se aumentara el índice y la columna, además de estos datos también se agregaran los valores que sean operadores y operadores compuestos (==, =>,>, etc).

Si en dado case el texto encuentre unas comillas de cierra, el estado pasara a ser 1 para identificar datos de tipo string, en caso de identificar caracteres del abecedario pasara al estado 2, en caso de digito se cambiara a estado 2.

Dentro del estado verificara si este se encuentra cerrado para agregar un nuevo token de cadena, de lo contrario marcara un error léxico de falta de comillas de cierre.

```
} else if (estado === 1) {
    // inside string
    if (char === '"') {
        columna += 1;
        buffer += '"';
        this.listaTokens.push(new Token("CADENA", buffer, linea, columna));
        buffer = "";
        estado = 0;
    } else if (char === "\n") {
        this.listaError.push(
            new Error(
                "Error Léxico: Cadena no cerrada",
                buffer,
                linea,
                columna
            )
        );
        buffer = "";
        linea += 1;
        columna = 1;
        estado = 0;
```

Dentro del estado 2 verificara si este es un carácter o digito, al evaluarlo con las palabras reservas dentro del buffer, estas agregar un nuevo token, de lo contrario lo identifiara con un token identificador.

```
} else if (estado === 2) {
    if (this.identificadores(char) || this.esDigito(char)) {
        columna += 1;
        buffer += char;
    } else {
        const tipo_token = this.palabrasReservadas(buffer);
        if (tipo_token) {
            this.listaTokens.push(
                new Token(tipo_token, buffer, linea, columna)
            );
        } else {
            this.listaTokens.push(
                new Token("IDENTIFICADOR", buffer, linea, columna)
            );
        }
        buffer = "";
        estado = 0;
        index -= 1;
    }
```

En el estado 3 verificara si este es un digito, de lo contrario lo tomara como un token de tipo ENTERO.

```
        } else if (estado === 3) {
            if (this.esDigito(char)) {
                columna += 1;
                buffer += char;
            } else {
                this.listaTokens.push(new Token("ENTERO", buffer, linea, columna));
                buffer = "";
                estado = 0;
                index -= 1;
            }
        }

        index += 1;
    }
```

Tendremos métodos adicionales para revelar todos los tokens encontrados o en su defecto los errores léxicos dentro del escaner.

```
imprimirTokens() {
    console.log("\n=== TOKENS ENCONTRADOS ===");
    this.listaTokens.forEach((token, i) => {
        console.log(
            `${i + 1}. Tipo: ${token.tipo}, Lexema: '${token.lexema}', Línea: ${token.linea
            }, Columna: ${token.columna}`
        );
    });
}
imprimirErrores() {
    if (this.listaError.length) {
        console.log("===ERRORES LEXICOS====");
        this.listaError.forEach((err, i) => {
            console.log(
                `${i + 1}. ${err.tipo}: '${err.lexema}', Linea: ${err.linea
                }, Columna: ${err.columna}`
            );
        });
    } else {
        console.log("\nNo se encontraron errores");
    }
}
```

PARSE:

La clase parse definirá un análisis sintactico evaluando la función para hallar errores definidos dentro de este código.

tokenActual: regreasara todo dentro de un array de tokens siempre y cuando cumpla con las condiciones.

Avanzar: sumare un digito en el index y continuara con el análisis sintactico del código.

Coincidir: En caso de coincidir la clase token este ejecutara un método llamado avanzar, devolviendo un valor booleando true y en caso contrario marca un error de tipo esperado.

```
export class Parser {
    constructor(tokens) {
        this.tokens = tokens;
        this.index = 0;
        this.errors = [];
    }

    tokenActual() {
        return this.tokens[this.index];
    }

    avanzar() {
        this.index++;
    }
    coincidir(tipoEsperado) {
        const token = this.tokenActual();
        if (token && token.tipo === tipoEsperado) {
            this.avanzar();
            return true;
        } else {
            this.errors.push(`Se esperaba '${tipoEsperado}' en línea ${token?.linea}`);
            return false;
        }
    }
```

Parse:

El método parse evalua un método instrucción siempre que el index sea menor a la cantidad de tokens obtenidos en el analizador léxico.

Dentro de instrucciones verificara que exista un texto public al inicio del código java, una vez hecho esto llamara a un método llamado función.

```
    parse() {
    console.log("Parseando...");
        while (this.index < this.tokens.length) {
            this.instruccion();
        }
        return this.errors;
    }
    instruccion() {
        const token = this.tokenActual(); // ← primero declaramos

        if (!token) return;

        if (token.tipo === "PUBLIC") {
            this.funcion();
            return;
        }
```

Dentro del método función se verificara que el texto cumpla con la sintaxis de un código java, empezando por public class “class” los corchetes y el código public static void main(String[args), una vez esto verificara que inicie en llave y que estas se encuentren cerrada, de no encontrar el public enviara un error.

```
funcion() {
    //public class Clase:
    if (this.coincidir("PUBLIC")) {
        if (this.coincidir("CLASS")) {
            if (!this.coincidir("IDENTIFICADOR")) return;
            this.coincidir("LLAVE_ABRE");
            this.coincidir("PUBLIC");
            this.coincidir("STATIC");
            this.coincidir("VOID");
            this.coincidir("MAIN");
            this.coincidir("PARENTESIS_ABRE");
            if (this.tokenActual()?.tipo === "IDENTIFICADOR") {
                this.avanzar(); // args
                this.coincidir("CORCHETE_ABRE");
                this.coincidir("CORCHETE_CIERRA");
                this.coincidir("ARGS");
            }
            this.coincidir("PARENTESIS_CIERRA");
            this.bloque(); // o instrucciones dentro de la clase o función
            this.coincidir("LLAVE_CIERRA");
            this.coincidir("LLAVE_CIERRA");
        } else if (!this.coincidir("CLASS")) {
            this.errors.push("Se esperaba 'class' después de 'public'");
            return;
        }
    }
}
```

Una vez se ejecute el método función, procederá a verificar que exista un método de declaración para verificar que cada variable este declara de tipo entero, double, boolean o char.

```
    if (["INT","DOUBLE","BOOLEAN", "CHAR"].includes(token.tipo)) {
        this.declaracion();
    } else if (token.tipo === "IDENTIFICADOR") {
        this.asignacion();
    } else if (token.tipo === "IF") {
        this.ifStatement();
    } else if (token.tipo === "WHILE") {
        this.whileStatement();
    } else if (token.tipo === "LLAVE_ABRE" && token.lexema === "{") {
        this.bloque();
    }  else if (token.tipo === "CENTINELA" && token.lexema === "#"){
```

En declaración se evaluara que este prosiga con un signo igual después del nombre de la variable y hará unb llamado a los métodos avanzar y expresión.

```
    declaracion() {
        this.avanzar(); // tipo
        if (!this.coincidir("IDENTIFICADOR")) return;
        if (this.tokenActual()?.lexema === "=") {
            this.avanzar();
            this.expresion();
        }
        this.coincidir("SIMBOLO"); // ;
    }
    asignacion() {
        this.avanzar(); // ID
        if (!this.coincidir("OP")) return;
        this.expresion();
        this.coincidir("SIMBOLO"); // ;
    }
    expresion() {
        this.termino();
        while (["+", "-", "*", "/"].includes(this.tokenActual()?.lexema)) {
            this.avanzar();
            this.termino();
```

También tenemos dos métodos adicionales para evaluar que el texto posea una expresión a ejecutar y termine en ; después de cada variable.

```
    declaracion() {
        this.avanzar(); // tipo
        if (!this.coincidir("IDENTIFICADOR")) return;
        if (this.tokenActual()?.lexema === "=") {
            this.avanzar();
            this.expresion();
        }
        this.coincidir("SIMBOLO"); // ;
    }
    asignacion() {
        this.avanzar(); // ID
        if (!this.coincidir("OP")) return;
        this.expresion();
        this.coincidir("SIMBOLO"); // ;
    }
    expresion() {
        this.termino();
        while (["+", "-", "*", "/"].includes(this.tokenActual()?.lexema)) {
            this.avanzar();
            this.termino();
```

En el método termino se evaluara que el valor de la variable sea de los tipos definidos dentro de las palabras reservadas (int,double, boolean, etc).

```
    termino() {
    const token = this.tokenActual();
    if (["ENTERO", "DOUBLE", "BOOLEAN", "CHAR", "CADENA"].includes(token?.tipo)) {
        this.avanzar();
    } else {
        this.errors.push(`Expresión inválida en línea ${token?.linea}`);
        this.avanzar();
    }
}
```

TRADUCTOR:

Dentro del traductor tendremos un contructor para tomar los tokens.

En el método declaración se avanzara en cada línea de código y tendremos una variable de tipo nombre para llamar el lexema de los tokens.

Si en cada declaración de variable se obtiene una variable esta hará un llamado al método expresión, y avanzara dentro del código.

```
class Traductor {
    constructor(tokens) {
        this.tokens = tokens;
        this.index = 0;
        this.codigoPython = "";
        this.indentacion = 0;
    }

    declaracion() {
        this.avanzar(); // tipo
        const nombre = this.tokenActual()?.lexema;
        this.avanzar(); // identificador

        let valor = "None";
        if (this.tokenActual()?.lexema === "=") {
            this.avanzar();
            valor = this.expresion();
        }

        this.avanzar(); // ;
        this.escribir(`${nombre} = ${valor}`);
    };
```

En el código instrucción se evaluara cada variable, condición if o while, empezando desde la ultima llave para iniciar el análisis por bloque y la creación de código en formato Python.

```
    instruccion() {
        const token = this.tokenActual();
        if (!token) return;

        if (["INT", "FLOAT", "BOOLEAN"].includes(token.tipo)) {
            this.declaracion();
        } else if (token.tipo === "IDENTIFICADOR") {
            this.asignacion();
        } else if (token.tipo === "IF") {
            this.ifStatement();
        } else if (token.tipo === "WHILE") {
            this.whileStatement();
        } else if (token.tipo === "SYM" && token.lexema === "{") {
            this.bloque();
        } else {
            this.avanzar(); // ignorar token no traducible
        }
    }
```

Árbol léxico:

Árbol léxico:

➔– [INT] 'int'

➔ [IDENTIFICADOR] 'x'

➔OP] '='

➔ [ENTERO] '5'

➔SYM] ';'

Árbol sintactico

<declaracion>

➔<tipo> → 'int'

➔ <identificador> → 'x'

➔ '='

➔ <expresion>

➔| └── <termino> → '5'

└➔ ';'

BNF

<programa> ::= <instruccion>*

<instruccion> ::= <declaracion>

| <asignacion>

| <if>

| <while>

| <bloque>

| <funcion>

<declaracion> ::= <tipo> <identificador> ["=" <expresion>] ";"

<asignacion> ::= <identificador> "=" <expresion> ";"

<if> ::= "if" "(" <expresion> ")" <instruccion>

<while> ::= "while" "(" <expresion> ")" <instruccion>

<bloque> ::= "{" <instruccion>* "}"

<funcion> ::= "public" "static" "void" <identificador> "(" [<identificador>] ")" <bloque>

| "public" "class" <identificador> <bloque>

<expresion> ::= <termino> { <operador> <termino> }*

<termino> ::= <identificador> | <numero>

<tipo> ::= "int" | "float" | "boolean"

<identificador> ::= token de tipo IDENTIFICADOR

<numero> ::= token de tipo ENTERO | FLOAT

<operador> ::= "+" | "-" | "*" | "/" | "==" | "!=" | "<" | ">" | "<=" | ">="

Resumen:

Desarrollar un sistema que permita traducir código fuente escrito en Java a su equivalente en Python, utilizando análisis léxico y sintáctico para garantizar la validez estructural del código antes de su conversión.

Componentes del sistema

1. **Análisis léxico (AnalizadorLexico)**

- **Entrada**: código fuente Java como texto.
- **Salida**: lista de tokens válidos y errores léxicos.
- **Reconoce**:
- Tipos de datos (int, float, boolean)
- Identificadores
- Operadores (=, +, ==, etc.)
- Símbolos ({, }, ;, (,))
- Literales (5, true, "texto")

2. **Análisis sintáctico (Parser)**

- **Entrada**: lista de tokens generados por el escáner.
- **Salida**: errores sintácticos o confirmación de estructura válida.
- **Reconoce estructuras como**:
- Declaraciones: int x = 5;
- Asignaciones: x = x + 1;
- Condicionales: if (x > 0) { ... }
- Ciclos: while (x < 10) { ... }
- Funciones: public static void main(...)
- Clases: public class Nombre { ... }

3. **Traductor (Traductor)**

- **Entrada**: tokens validados por el parser.
- **Salida**: código Python equivalente.
- **Traduce**:
- int x = 5; → x = 5
- if (x > 0) { ... } → if x > 0:
- while (x < 10) { ... } → while x < 10:
- public static void main(...) → def main():
- Bloques { ... } → indentación en Python