Manual técnico

Cristian Daniel Gomez Escobar
202107190

INFORMACION DEL SISTEMA

El sistema “TypeWise” se ha creado para que los estudiantes del curso de Organización de lenguajes y compiladores 1 creen un lenguaje de programación para los estudiantes de Introducción a la Programación y Computación1 aprendan a programar y tener conocimiento de todas las generalidades de un lenguaje de programación. El programa esta diseñado para ejecutarse en la nube usando un framework dedicado para ello y con la posibilidad de graficar varios reportes.

DATOS TECNICOS

LENGUAJE UTILIZADO

El programa fue desarrollado en el lenguaje de programación JavaScript y TypeScript en el framework React

IDE UTILIZADO

Visual Studio Code

SISTEMA OPERTIVO

El programa puede ser ejecutado en los sistemas operativos Windows, Linux/UNIX, MacOS, AIX, IMB i, iOS, iPadOS, OS/390, z/OS, RISC PS, Solaris, VMS y HP-UX

Requisitos del sistema

Windows

- windows 7, Windows 8, Windows 8.1, Windows 10 o una versión posterior
- Un procesador Intel Pentium 4 o superior compatible con SSE Mac, Para utilizar el navegador Chrome en Mac
- OS X El Capitan 10.11 o una versión posterior

Linux

- Ubuntu 14.04, Debian 8, openSUSE 13.3, Fedora Linux 24 o cualquier otra versión de 64 bits posterior a estas}
- Un procesador Intel Pentium 4 o superior compatible con SSE3

SERVER

Controller

Api.controller: Nos permite controlar las funciones a exportar de nuestro api

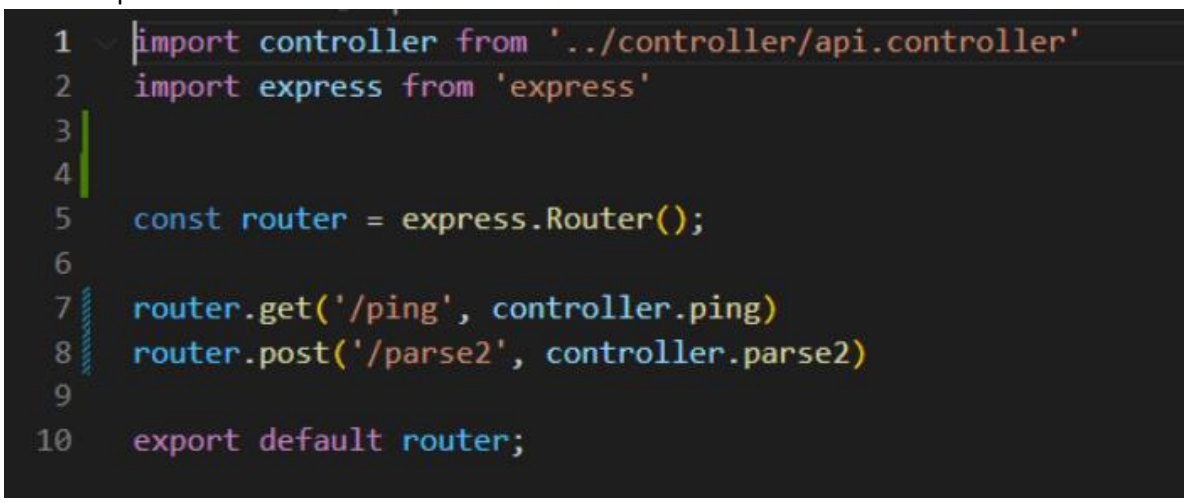
```
import * as health from './health/ping'
import * as parser2 from './parser/parser2'



export default {
    ...health,
    ...parser2
}
```

Parser: Esta funcione ejecuta la traducción de código enviado desde nuestro cliente y devuelve un JSON con los datos necesarios a mostrar al cliente

```
import { Response, Request } from "express";

import AST from '../../utils/Interprete/Ast/Ast'
import Nodo from '../../utils/Interprete/Ast/Nodo'
import Controlador from "../../utils/Interprete/Controlador"
import TablaSimbolos from "../../utils/Interprete/TablaSimbolos/TablaSimbolos"
import Errores from '../../utils/Interprete/Ast/Errores';

export const parse2 = (req: Request & unknown, res: Response): void => {

    var interprete = require('../../utils/Analizador/interprete').parser;
    const { peticion } = req.body
    let grafo = "digraph G {a -> b; b -> c; c -> a;}";

    try {
        let ast: AST = interprete.parse(peticion)
        let respuesta = "";
        let controlador = new Controlador()
        let ts_global = new TablaSimbolos(null);

        ast.ejecutar(controlador, ts_global)
        let ts_html = controlador.graficar_ts(controlador, ts_global)

        let ts_html_error = controlador.obtenererrores();

        let nodo_ast : Nodo = ast.recorrer();
        grafo = nodo_ast.GraficarSintactico();

        res.json({ consola: controlador.consola, errores: ts_html_error, graphviz: grafo, tablaSimbolos: ts_html })


    } catch (err) {
        console.log(err)
        res.json({consola: "Se ha producido un error", graphviz: grafo})
    }

```

Routes

Api.routes: son las rutas y métodos que dejamos disponibles para quien quiera acceder a nuestro api del servidor

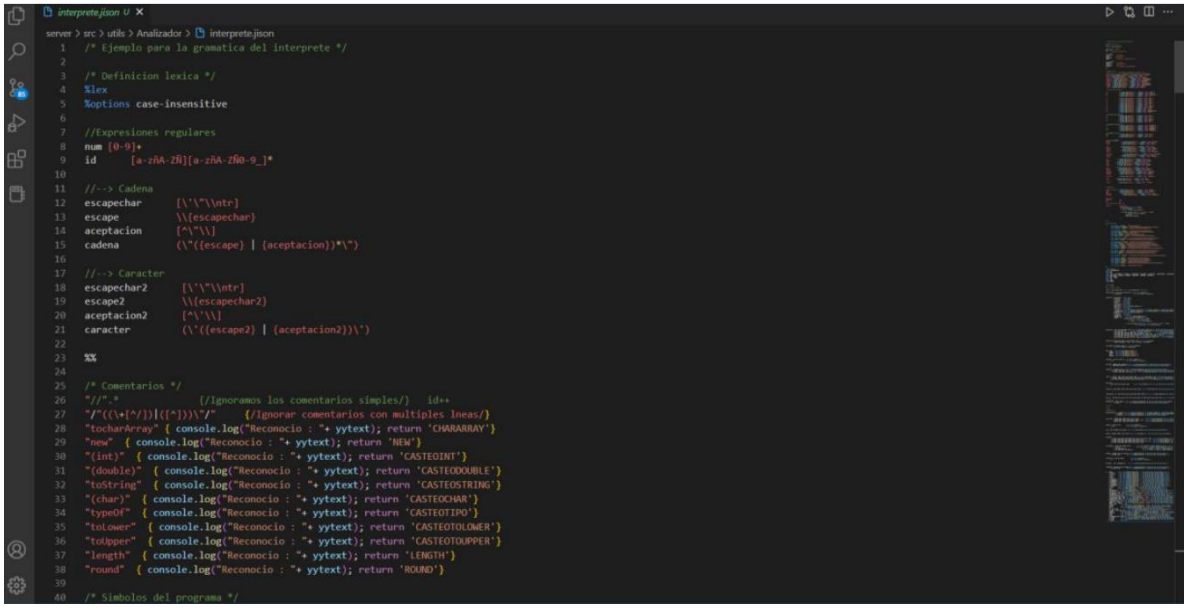


```
import controller from '../controller/api.controller'
import express from 'express'


const router = express.Router();

router.get('/ping', controller.ping)
router.post('/parse2', controller.parse2)

export default router;
```

Utils

Analizador

Interprete.jison: este archivo de texto plano incluye todas nuestras reglas léxicas y gramaticales a usar en nuestra ejecución de código, este a su vez genera un archivo interprete.js con sus métodos propios necesarios para la ejecución de código



```
/* Ejemplo para la gramatica del interprete */

/* Definicion lexica */
%lex
%options case-insensitive

//Expresiones regulares
num [0-9]+
id      [a-zñA-ZÑ][a-zñA-ZÑ0-9_]*

//--> Cadena
escapechar      [\'\"\\ntr]
escape          \\(escapechar)
aceptacion      [^\"\\]
cadena          (\"((escape) | (aceptacion))*\")

//--> Caracter
escapechar2      [\'\"\\ntr]
escape2          \\(escapechar2)
aceptacion2      [^\'\\]
caracter         (\'((escape2) | (aceptacion2))\')

%%

/* Comentarios */
"//".*                  {/Ignoramos los comentarios simples/}   id++
"/"((\+[^/])|([^]))\"/"     {/Ignorar comentarios con multiples lneas/}
"tocharArray" { console.log("Reconocio : "+ yytext); return 'CHARARRAY'}
"new"   { console.log("Reconocio : "+ yytext); return 'NEW'}
"(int)"  { console.log("Reconocio : "+ yytext); return 'CASTEOINT'}
"(double)"  { console.log("Reconocio : "+ yytext); return 'CASTEODOUBLE'}
"toString"  { console.log("Reconocio : "+ yytext); return 'CASTEOSTRING'}
"(char)"  { console.log("Reconocio : "+ yytext); return 'CASTEOCHAR'}
"typeOf"  { console.log("Reconocio : "+ yytext); return 'CASTEOTIPO'}
"toLower"  { console.log("Reconocio : "+ yytext); return 'CASTEOTOLOWER'}
"toUpper"  { console.log("Reconocio : "+ yytext); return 'CASTEOTOUPPER'}
"length"  { console.log("Reconocio : "+ yytext); return 'LENGTH'}
"round"  { console.log("Reconocio : "+ yytext); return 'ROUND'}

/* Simbolos del programa */
```

```
interpretejison U
server > src > utils > Analizador > interpretejison
204  instrucciones : instrucciones instruccion   { $$ = $1; $$.push($2); }
205            | instruccion                     { $$ = new Array(); $$.push($1); }
206            ;
207
208  instruccion : declaracion   { $$ =  $1; }
209            | startwith      { $$ = $1; }
210            | writeline      { $$ = $1; }
211            | asignacion     { $$ = $1; }
212            | sent_if        { $$ = $1; }
213            | sent_while     { $$ = $1; }
214            | sent_Dowhile   { $$ = $1; }
215            | BREAK PYC      { $$ = new detener.default(); }
216            | sent_switch    { $$ = $1; }
217            | sent_for       { $$ = $1; }
218            |  ID DECRE PYC  { $$ = new asignacion.default($1, new aritmetica.default(new identificador.default($1, @1.first_line, @1.last_column), '-', new primitivo.default(1,
219            | ID INCRE  PYC { $$ = new asignacion.default($1, new aritmetica.default(new identificador.default($1, @1.first_line, @1.last_column), '+', new primitivo.default(1, '
220            | CONTINUE PYC  { $$ = new continuar.default(); }
221            | funciones      { $$ = $1; }
222            | llamada PYC    { $$ = $1; }
223            | RETURN PYC        { $$ = new retorno.default(null); }
224            | RETURN e PYC      { $$ = new retorno.default($2); }
225            | error         { console.log("Error Sintactico: " + yytext
226                                    + " linea: " + this._$.first_line
227                                    + " columna: " + this._$.first_column);
228
229                                new errores.default("Sintactico", "No se esperaba el caracter "+ yytext ,
230                                            this.$.first_line ,this.$.first_column);
231                            }
232            ;
233
234  declaracion : tipo lista_ids IGUAL e PYC  { $$ = new declaracion.default($1, $2, $4,@1.first_line, @1.last_column);}
235            | tipo lista_ids PYC          { $$ = new declaracion.default($1, $2, null,  @1.first_line, @1.last_column);}
236            | tipo CORA CORC lista_ids IGUAL NEW  tipo CORA e CORC  PYC          { $$ = new declaracion.default($1, $4, null,  @1.first_line, @1.last_column,$7,$9);}
237            | tipo CORA CORC lista_ids IGUAL LLAVA listasimple  LLAVC  PYC        { $$ = new declaracion.default($1, $4, null,  @1.first_line, @1.last_column,$1,$4);}
238            | tipo CORA CORC CORA CORC lista_ids IGUAL NEW  tipo CORA e CORC  CORA  e CORC PYC        { $$ = new declaracion.default($1, $6, null,  @1.first_line, @1.last_column
239            | tipo CORA CORC  lista_ids IGUAL e PYC { $$ = new declaracion.default($1, $4, $6,  @1.first_line, @1.last_column);}
240            | tipo CORA CORC CORA CORC lista_ids  IGUAL LLAVA doublearray  LLAVC  PYC      { $$ = new declaracion.default($1, $6, null,  @1.first_line, @1.last_column,$1,$4,$7,$1
241            ;
242
```

Interprete

AST

Ast: esta clase ejecuta el codigo recibido gracias a la tabla de símbolos y con esa información arma el ast de todo el código

```
ejecutar(controlador: Controlador, ts: TablaSimbolos) {
    //1era pasada vamos a guardar las funciones y metodos del programa
    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof Funcion){
            let funcion = instruccion as Funcion;
            funcion.agregarFuncionTS(ts);
        }
    }
    //2 da pasada. ejecutar las declaraciones de variables
    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof Declaracion || instruccion instanceof Asignacion){
        //if(instruccion instanceof Declaracion ){

            instruccion.ejecutar(controlador,ts);
        }
    }
    //3ra pada. ejecutamos todas las demas instrucciones
    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof StartWith){
            instruccion.ejecutar(controlador,ts);
            break;
        }
    }
    for(let instruccion of this.lista_instrucciones){
        if(!(instruccion instanceof Declaracion) && !(instruccion instanceof Funcion) && !(instruccion instanceof StartWith)){
            instruccion.ejecutar(controlador,ts);
        }
    }

}
recorrer(): Nodo {
    let raiz = new Nodo("INICIO","");

    for(let inst of this.lista_instrucciones){
        raiz.AddHijo(inst.recorrer());
    }

    return raiz;
}
```

Lista_errores: es la lista general donde se guardan todos los errores léxicos, sintácticos y semánticos

```
import Errores from "./Errores"

export let lista_errores = {Errores: Array<Errores>()}

```

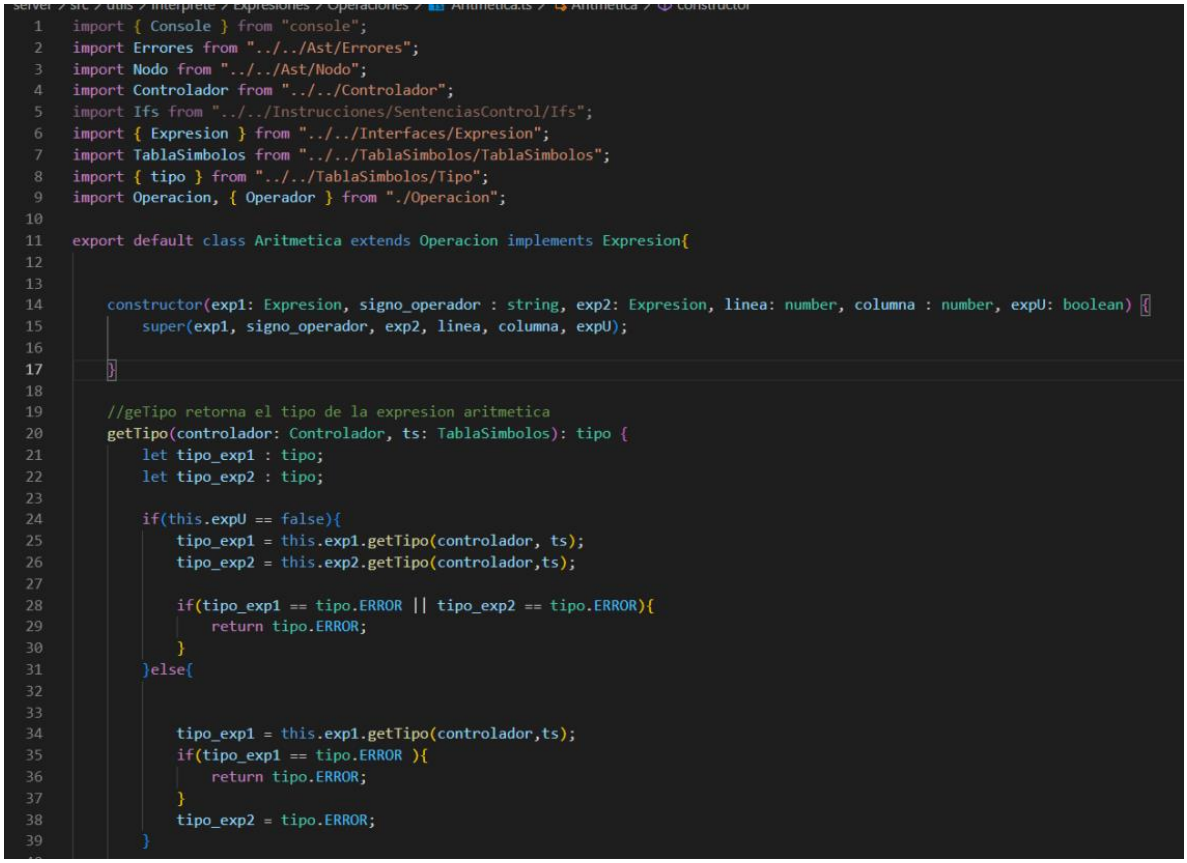
Nodo: esta clase contiene la información y métodos necesarios para la construcción de los nodos del AST

```
export default class Nodo{
    public token : string;
    public lexema : string;
    public hijos : Array<Nodo>;

    constructor(token : string, lexema : string) {
        this.token = token;
        this.lexema = lexema;
        this.hijos = new Array<Nodo>();
    }

    public AddHijo(nuevo :Nodo):void{
        this.hijos.push(nuevo);
    }

    public getToken():string{
        return this.token;
    }

    public GraficarSintactico():string{
        let grafica: string = `digraph G {\n\n${this.GraficarNodos(this, "0")} \n\n}`

        return grafica;
    }

    public GraficarNodos(nodo: Nodo, i: string):string{
        let k = 0;
        let r = "";
        let nodoTerm : string = nodo.token;
        nodoTerm = nodoTerm.replace("\"","");
        r = `node${i}[label = \"${nodoTerm}\"];\n`;

        for(let j = 0; j<= nodo.hijos.length - 1; j++){
            r = `${r}node${i} -> node${i}${k}\n`;
            r = r + this.GraficarNodos(nodo.hijos[j], ""+i+k);
            k = k + 1;
        }

        if(!(nodo.lexema.match(''))||!(nodo.lexema.match(""))){
            let nodoToken = nodo.lexema;
```

Expresiones

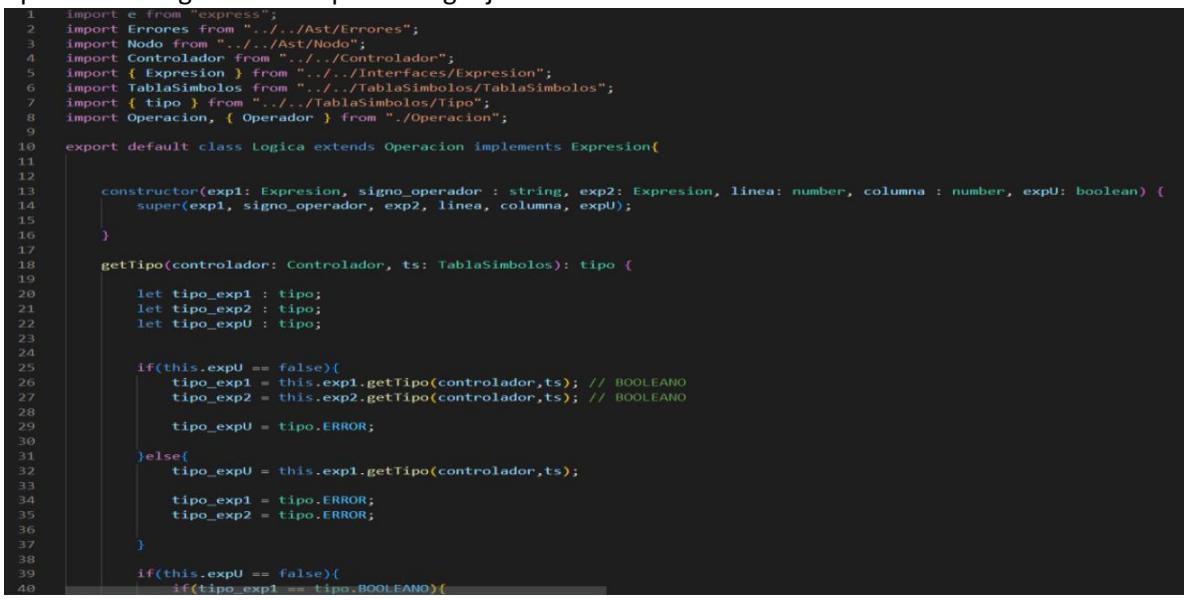
Operaciones

Aritmética: esta clase contiene todos los métodos necesarios para la validación y ejecución de operaciones aritméticas validas por el lenguaje



```
import { Console } from "console";
import Errores from "../../Ast/Errores";
import Nodo from "../../Ast/Nodo";
import Controlador from "../../Controlador";
import Ifs from "../../Instrucciones/SentenciasControl/Ifs";
import { Expresion } from "../../Interfaces/Expresion";
import TablaSimbolos from "../../TablaSimbolos/TablaSimbolos";
import { tipo } from "../../TablaSimbolos/Tipo";
import Operacion, { Operador } from "./Operacion";

export default class Aritmetica extends Operacion implements Expresion{


    constructor(exp1: Expresion, signo_operador : string, exp2: Expresion, linea: number, columna : number, expU: boolean) {
        super(exp1, signo_operador, exp2, linea, columna, expU);

    }

    //geTipo retorna el tipo de la expresion aritmetica
    getTipo(controlador: Controlador, ts: TablaSimbolos): tipo {
        let tipo_exp1 : tipo;
        let tipo_exp2 : tipo;

        if(this.expU == false){
            tipo_exp1 = this.exp1.getTipo(controlador, ts);
            tipo_exp2 = this.exp2.getTipo(controlador,ts);

            if(tipo_exp1 == tipo.ERROR || tipo_exp2 == tipo.ERROR){
                return tipo.ERROR;
            }
        }else{


            tipo_exp1 = this.exp1.getTipo(controlador,ts);
            if(tipo_exp1 == tipo.ERROR ){
                return tipo.ERROR;
            }
            tipo_exp2 = tipo.ERROR;
        }
```

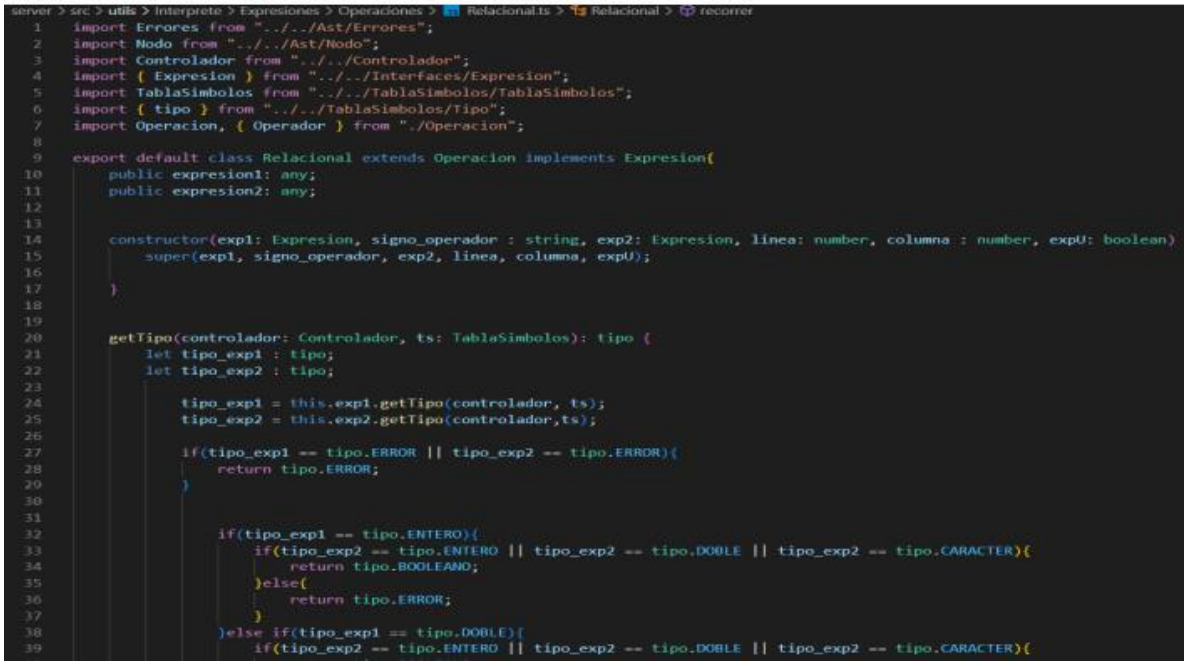
Logica: esta clase contiene todos los métodos necesarios para la validación y ejecución de operaciones lógicas validas por el lenguaje



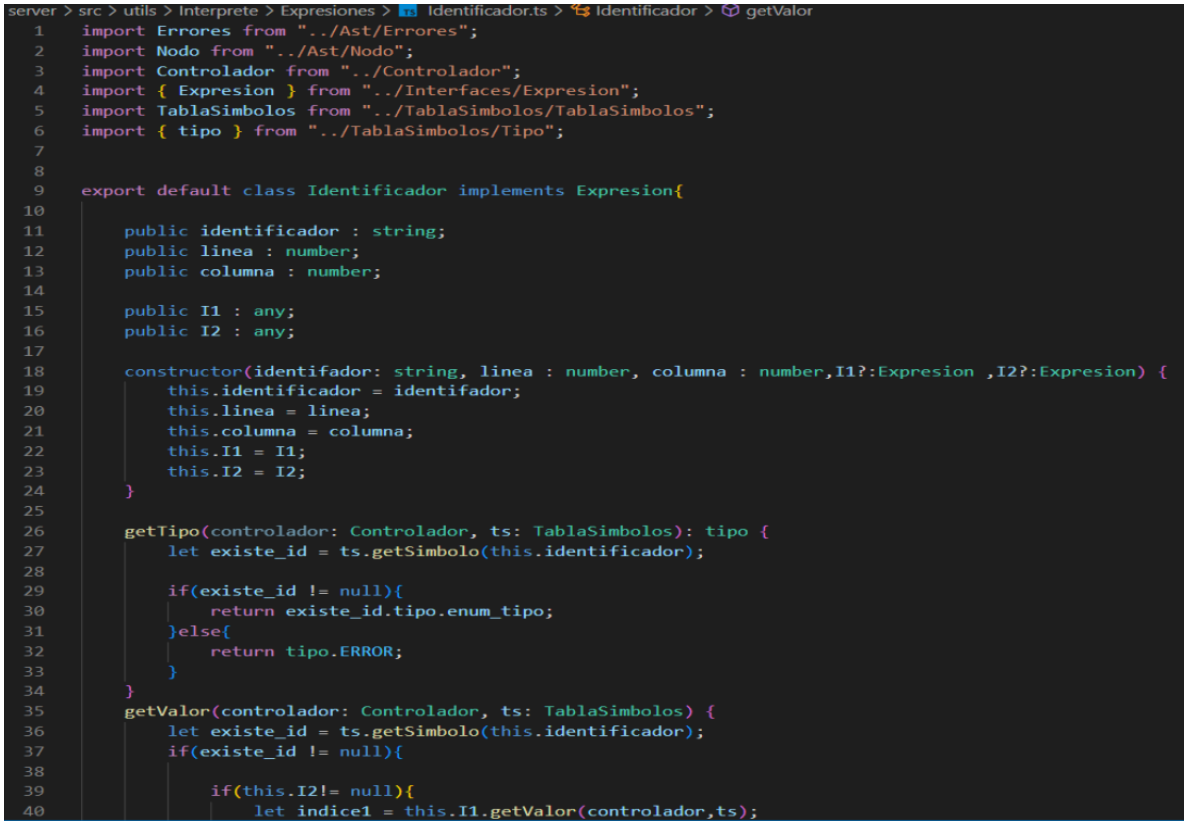
```
import e from "express";
import Errores from "../../Ast/Errores";
import Nodo from "../../Ast/Nodo";
import Controlador from "../../Controlador";
import { Expresion } from "../../Interfaces/Expresion";
import TablaSimbolos from "../../TablaSimbolos/TablaSimbolos";
import { tipo } from "../../TablaSimbolos/Tipo";
import Operacion, { Operador } from "./Operacion";

export default class Logica extends Operacion implements Expresion{


    constructor(exp1: Expresion, signo_operador : string, exp2: Expresion, linea: number, columna : number, expU: boolean) {
        super(exp1, signo_operador, exp2, linea, columna, expU);

    }

    getTipo(controlador: Controlador, ts: TablaSimbolos): tipo {

        let tipo_exp1 : tipo;
        let tipo_exp2 : tipo;
        let tipo_expU : tipo;


        if(this.expU == false){
            tipo_exp1 = this.exp1.getTipo(controlador,ts); // BOOLEANO
            tipo_exp2 = this.exp2.getTipo(controlador,ts); // BOOLEANO

            tipo_expU = tipo.ERROR;

        }else{
            tipo_expU = this.exp1.getTipo(controlador,ts);

            tipo_exp1 = tipo.ERROR;
            tipo_exp2 = tipo.ERROR;

        }

        if(this.expU == false){
            if(tipo_exp1 == tipo.BOOLEANO){
```

Operación: esta clase contiene todos los métodos necesarios para la validación y ejecución de las operaciones internas validas por el lenguaje

Relacional: esta clase contiene todos los métodos necesarios para la validación y ejecución de las operaciones relacionales validas por el lenguaje



Identificador: esta clase nos permite mantener el orden en la tabla de símbolos permitiendo que no salgan de los índices establecidos



Primitivo: esta clase almacena todas las variables de tipo primitivo de nuestro lenguaje

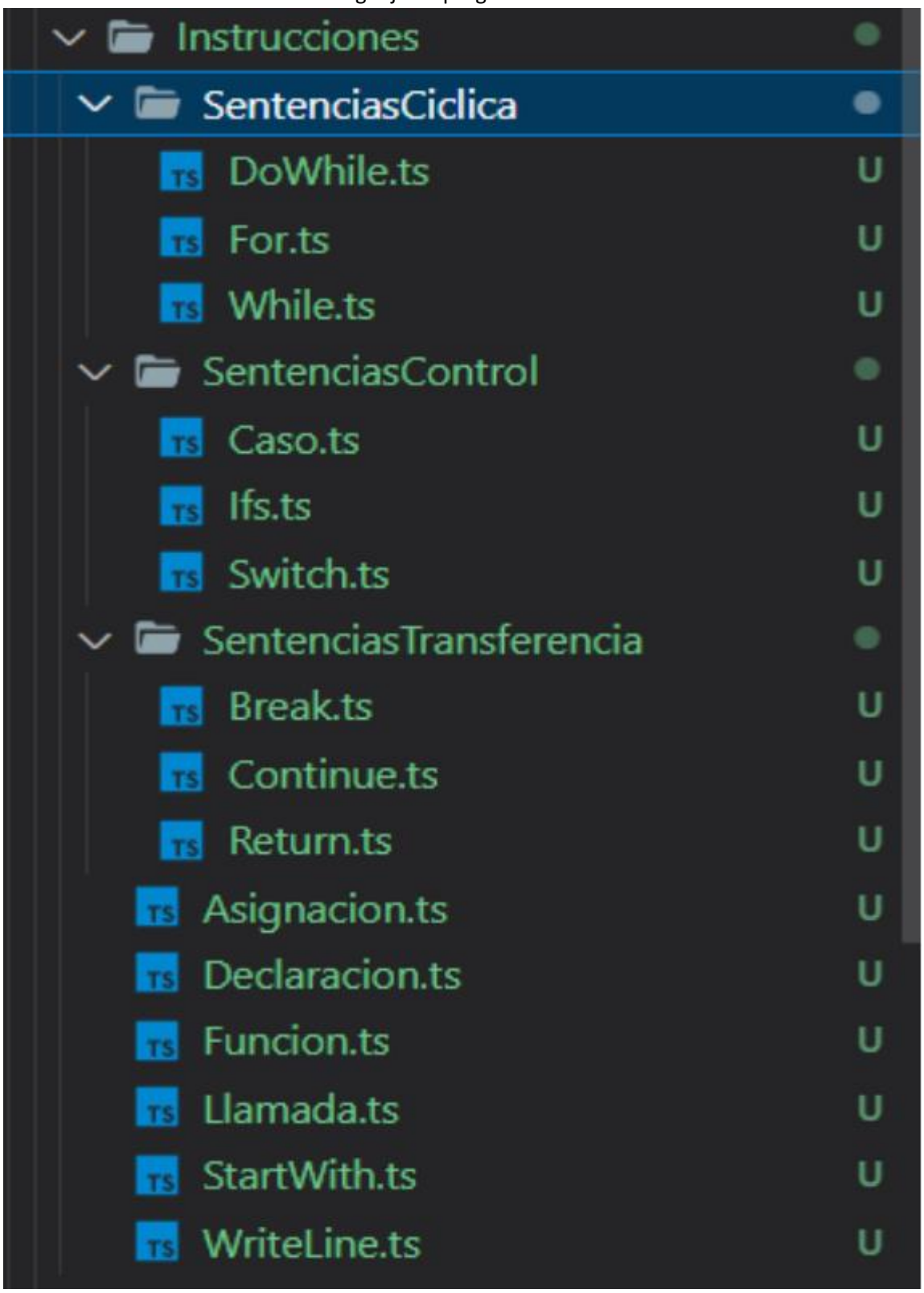
```typescript
import Nodo from "../Ast/Nodo";
import Controlador from "../Controlador";
import { Expresion } from "../Interfaces/Expresion";
import TablaSimbolos from "../TablaSimbolos/TablaSimbolos";
import Tipo, { tipo } from "../TablaSimbolos/Tipo";


export default class Primitivo implements Expresion{

    public valor_primitivo : any;
    public linea : number;
    public columna : number;
    public tipo : Tipo;

    constructor(valor_primitivo : any, tipo : string , linea : number, columna: number) {
        this.valor_primitivo = valor_primitivo;
        this.linea = linea;
        this.columna = columna;
        this.tipo = new Tipo(tipo);
    }

    getTipo(controlador: Controlador, ts: TablaSimbolos): tipo {
        return this.tipo.enum_tipo;
    }

    getValor(controlador: Controlador, ts: TablaSimbolos) {
        return this.valor_primitivo;
    }

    recorrer(): Nodo {
        let padre = new Nodo("Primitivo","");
        padre.AddHijo(new Nodo(this.valor_primitivo.toString(),""));

       return padre;
    }

}
```

Ternario: esta clase almacena todas las variables de tipo ternario de nuestro lenguaje

```ts
import Nodo from "../Ast/Nodo";
import Controlador from "../Controlador";
import { Expresion } from "../Interfaces/Expresion";
import TablaSimbolos from "../TablaSimbolos/TablaSimbolos";
import { tipo } from "../TablaSimbolos/Tipo";

export default class Ternario implements Expresion{

    public condicion : Expresion;
    public verdadero : Expresion;
    public falso : Expresion;
    public linea : number;
    public columna : number;

    constructor(condicion : Expresion, verdadero :Expresion, falso :Expresion, linea: number, columna: number) {
        this.condicion = condicion;
        this.verdadero = verdadero;
        this.falso = falso;
        this.linea = linea;
        this.columna = columna;
    }

    getTipo(controlador: Controlador, ts: TablaSimbolos): tipo {
        let valor_condicion = this.condicion.getValor(controlador,ts);

        if(this.condicion.getTipo(controlador, ts) == tipo.BOOLEANO){
            return valor_condicion ? this.verdadero.getTipo(controlador,ts) : this.falso.getTipo(controlador,ts);
        }else{
            return tipo.ERROR;
        }
    }
    getValor(controlador: Controlador, ts: TablaSimbolos) {
        let valor_condicion = this.condicion.getValor(controlador,ts);

        if(this.condicion.getTipo(controlador, ts) == tipo.BOOLEANO){
            return valor_condicion ? this.verdadero.getValor(controlador,ts) : this.falso.getValor(controlador,ts);
        }else{
            //reportamos error semantico
            return null;
        }
```

Instrucciones

Estas clases tienen los métodos y comprobaciones necesarios para la ejecución de código de cada una de las sentencias del lenguaje de programación



Usan una estructura similar cada una de ellas, que es un constructor con los parámetros necesarios para la ejecución de código, así como su lista interna de instrucciones

```
export default class While implements Instruccion{

    public condicion: Expresion;
    public lista_instrucciones : Array<Instruccion>;
    public linea : number;
    public columna : number;


    constructor(condicion : Expresion, lista_instrucciones: Array<Instruccion>, linea:number, columna:number) {
        this.condicion = condicion;
        this.lista_instrucciones = lista_instrucciones;
        this.linea = linea;
        this.columna = columna;
    }
```

Interfaces

Expresión: esta interfaz tiene los datos necesarios para la encapsulación de código para la traducción de código de las expresiones presentes en el lenguaje de programación

```
1   import Nodo from "../Ast/Nodo";
2   import Controlador from "../Controlador";
3   import TablaSimbolos from "../TablaSimbolos/TablaSimbolos";
4   import { tipo } from "../TablaSimbolos/Tipo";
5
6   /**
7    * @interface Expresion
8    * Las funciones dentro de esta clase solo estan declarados indicando su tipo, nombre y parametros
9    * Las clases que implementen esta interfaz le estaremos indicando al programa que seran Expresiones
10   * y deberan de implementar las mismas funciones declaradas aca
11   */
12  export interface Expresion{
13
14
15      /**
16       * @function getTipo retorna el tipo del valor de la expresion
17       * @param controlador llevamos el control de todo el programa
18       * @param ts accede a la tabla de simbolos
19       */
20      getTipo(controlador : Controlador, ts : TablaSimbolos) : tipo ;
21
22
23      /**
24       * @function getValor retorna el valor de la expresion
25       * @param controlador llevamos el control de todo el programa
26       * @param ts accede a la tabla de simbolos
27       */
28      getValor(controlador : Controlador, ts: TablaSimbolos):any;
29
30      /**
31       * @function recorrer crea y recorre el subarbol de la expresion
32       */
33      recorrer(): Nodo;
34
35  }
```

Expresión: esta interfaz tiene los datos necesarios para la encapsulación de código para la traducción de código de las expresiones presentes en el lenguaje de programación

```typescript
import Nodo from "../Ast/Nodo";
import Controlador from "../Controlador";
import TablaSimbolos from "../TablaSimbolos/TablaSimbolos";

export interface Instruccion {

    /**
     * @function ejecutar ejecuta las instrucciones
     * @param controlador llevamos el control de todo el programa
     * @param ts accede a la tabla de simbolos
     */
    ejecutar(controlador : Controlador, ts : TablaSimbolos) :any ;

    /**
     * @function recorrer crea y retorna el subarbol de la instruccion
     */
    recorrer(): Nodo;
}
```

Tabla de símbolos

Símbolo: en esta clase esta toda la información para guardar todos los símbolos en la tabla de símbolos

```typescript
import { threadId } from "worker_threads";
import Tipo from "./Tipo";

export default class Simbolo{

    public simbolo : number;
    public tipo : Tipo;
    public identificador : string;
    public valor : any;

    public lista_params : Array<Simbolo> | undefined;
    public metodo : boolean | undefined;
    public columna : number;
    public linea : number;


    public ambito : any;
    constructor(simbolo: number, tipo: Tipo, identificador: string, valor: any, linea:number ,columna :number, lista_params?:Array<Simbo
        this.simbolo = simbolo;
        this.tipo = tipo;
        this.identificador = identificador;
        this.valor = valor;
        this.lista_params = lista_params;
        this.metodo = metodo;
        this.ambito=ambito;
        this.linea = linea;
        this.columna = columna
    }

    setValor(valor:any,I1?:number ,I2?:number): void{
        if(I2!= null){
            if(I1!= null){
                if(I1 < this.valor.length || I2 < this.valor.length){
                    this.valor[I1][I2] = valor

                }else{
                    console.log("ERROR FUER ADE INDICE")
                }
            }
```

Tabla de símbolos: en esta clase esta toda la información de la tabla de símbolos así como sus métodos de acceso a cada una de sus funciones necesarias para la traducción de código

```
import Simbolo from "./Simbolo";

export default class TablaSimbolos{
    public name:any;
    public ant : TablaSimbolos;
    public tabla : Map<string, Simbolo>;
    public sig:any;

    constructor(ant : TablaSimbolos | any,name?:string|any) {
        this.ant = ant;
        this.tabla = new Map<string, Simbolo>();
        if(ant!= null){
            ant.sig = this;
        }
        this.name = name
    }

    agregar(id: string, simbolo : Simbolo){
        this.tabla.set(id.toLowerCase(), simbolo);
    }

    existe(id: string): boolean{
        let ts : TablaSimbolos = this;

        while(ts != null){
            let existe = ts.tabla.get(id.toLowerCase());

            if(existe != null){
                return true;
            }
            ts = ts.ant;
        }
        return false;
    }

    getSimbolo(id: string){
        let ts : TablaSimbolos = this;

        while(ts != null){
            let existe = ts.tabla.get(id.toLowerCase());
```

Tipo: en esta clase esta toda la información para el almacenamiento de los tipos de datos del lenguaje de programación

```
export enum tipo{
    ENTERO,
    DOBLE,
    BOOLEANO,
    CARACTER,
    CADENA,
    ERROR,
    VOID
}

export default class Tipo{
    public nombre_tipo : string;
    public enum_tipo : tipo;

    constructor(nombre_tipo : string) {
        this.nombre_tipo = nombre_tipo;
        this.enum_tipo = this.gettipo();
    }

    gettipo(): tipo{
        if(this.nombre_tipo == 'ENTERO'){
            return tipo.ENTERO;
        }else if(this.nombre_tipo == 'DOBLE'){
            return tipo.DOBLE;
        }else if(this.nombre_tipo == 'CADENA'){
            return tipo.CADENA;
        }else if(this.nombre_tipo == 'CARACTER'){
            return tipo.CARACTER;
        }else if(this.nombre_tipo == 'BOOLEANO'){
            return tipo.BOOLEANO;
        }else if(this.nombre_tipo == 'VOID'){
            return tipo.VOID;
        }else{
            return tipo.ERROR;
        }
    }

}
```

Controlador: en esta clase están todos los métodos necesarios para la ejecución de código que deben ser accedidos por múltiples clases a la vez

```
1  import Errores from "./Ast/Errores";
2  import Simbolo from "./TablaSimbolos/Simbolo";
3  import TablaSimbolos from "./TablaSimbolos/TablaSimbolos";
4
5
6  export default class Controlador{
7
8      public errores : Array<Errores>;
9      public consola : string;
10     public sent_ciclica : boolean;
11     constructor() {
12         this.errores = new Array<Errores>();
13         this.consola = "";
14         this.sent_ciclica = false;
15     }
16
17     obtenererrores(){
18         console.log(this.errores);
19         return this.errores;
20     }
21
22     print(cadena : string, tipo:boolean){
23         if(tipo){
24             this.consola = this.consola + cadena + " \r\n ";
25         }else{
26             this.consola = this.consola + cadena ;
27         }
28     }
29
30
31     append(cadena : string){
32         this.consola = this.consola + cadena + " \r\n ";
33     }
34
35
36      graficar_ts(controlador:Controlador, ts:TablaSimbolos):string{
37
38         var TextSalida = "";
39         while(ts != null){
40             ts.tabla.forEach((sim: Simbolo, key : string) =>{
```

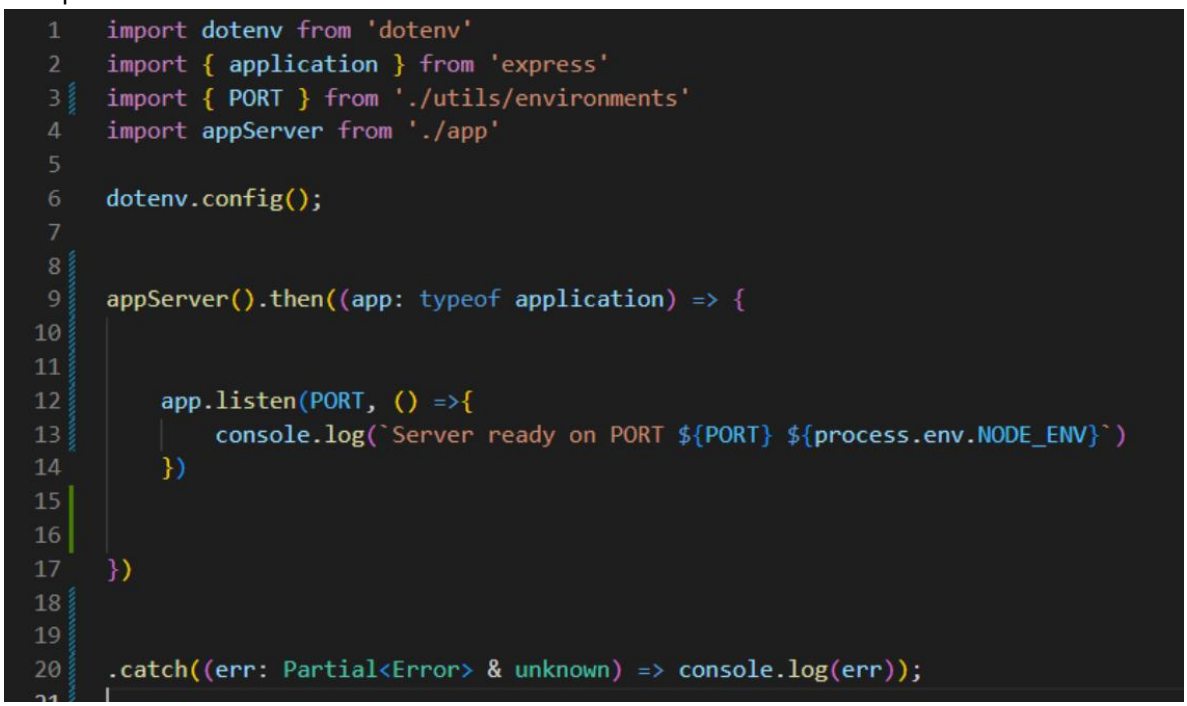
App: esta clase nos permite crear la aplicación que da inicio a la api

```ts
import express, { application, Request } from 'express';
import morgan from 'morgan';
import cors from 'cors';
import bodyParser from 'body-parser';
import api from './routes/api.routes'

const makeApp = async () : Promise<typeof application> => {
    const app = express();

    app.use(morgan('dev', {
        skip: (req: Request) => req.url === '/api/ping'
    }))

    app.use(cors())
    app.use(bodyParser.urlencoded({extended: false, limit: '100mb'}))
    app.use(bodyParser.json({limit: '100mb'}))

    app.use('/api', api)
    return app
}

export default makeApp;
```

Graphviz: en esta clase esta todo lo necesario para el graficado de graphviz

```ts
import { Digraph, Node, Edge, EdgeTargetTuple, attribute, toDot } from 'ts-graphviz';
import { CliRenderer } from "@diagrams-ts/graphviz-cli-renderer";

export class CDigraph extends Digraph {
  constructor(label: string) {
    super('G', {
      [attribute.label]: label,
    });
  }

  public async generate(){
    const render = CliRenderer({ outputFile: "./salida.svg", format: "svg" });
    await (async () => {
        try {
          await render(
            toDot(this)
          );
        } catch (error) {
          console.log(error);
        }
    })();
  }
}
export class CNode extends Node {
  constructor(id: number, label: string) {
    super(`node${id}`, {
      [attribute.label]: label
    });
  }
}

export class CEdge extends Edge {
  constructor(targets: EdgeTargetTuple, label: string) {
    super(targets, {
      [attribute.label]: label
    });
  }
}
```

Server: esta clase despliega nuestra aplicación en un puerto, listo para ser usado por cualquiera



```
import dotenv from 'dotenv'
import { application } from 'express'
import { PORT } from './utils/environments'
import appServer from './app'

dotenv.config();


appServer().then((app: typeof application) => {


    app.listen(PORT, () =>{
        console.log(`Server ready on PORT ${PORT} ${process.env.NODE_ENV}`)
    })


})


.catch((err: Partial<Error> & unknown) => console.log(err));
```

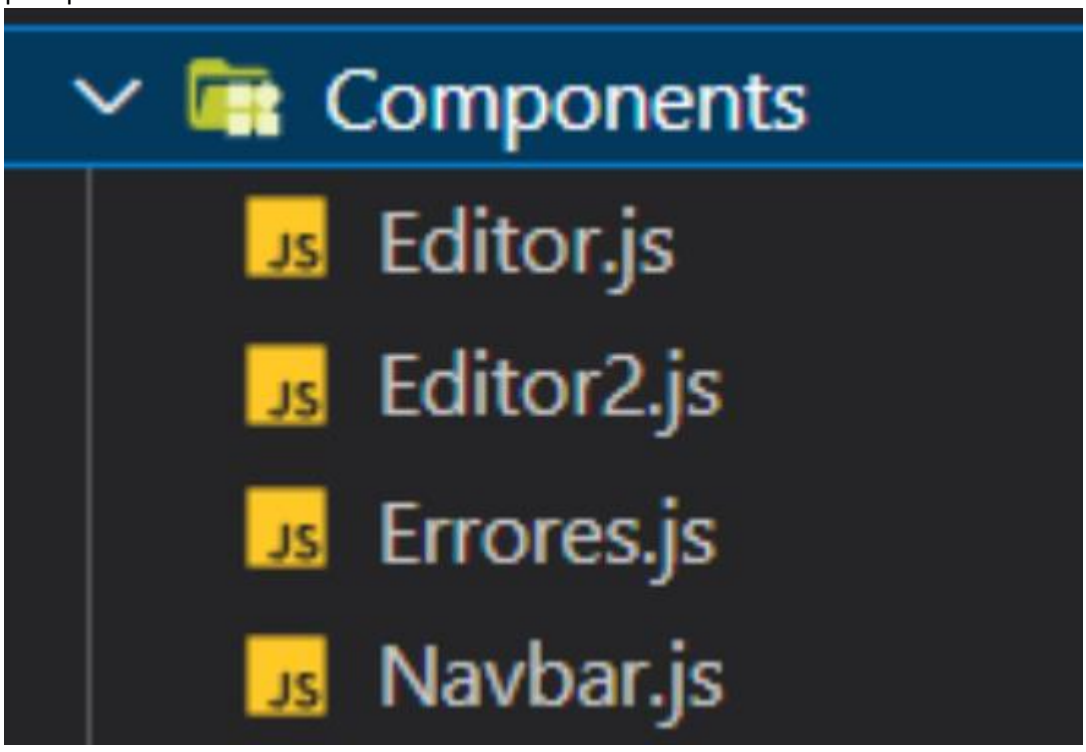
CLIENT

Services

Parser: esta clase se comunica con el api del server y manda los datos a traducir y recibe a su vez la respuesta del servidor para luego desplegarla al cliente

```
import axios from 'axios';

const instance = axios.create({
    baseURL: 'http://localhost:5002/api',
    timeout: 15000,
    headers: {
        'Content-Type': 'application/json',
    }
});

export const parse = async(value) => {
    const {data} = await instance.post('/parse2', {peticion: value});
    return data;
}

export const ping = async() => {
    const {data} = await instance.get('/ping');
    return data;
}
```

Components: Esta clase es la encargada de pintar cada uno de los componentes utilizados por aparte.



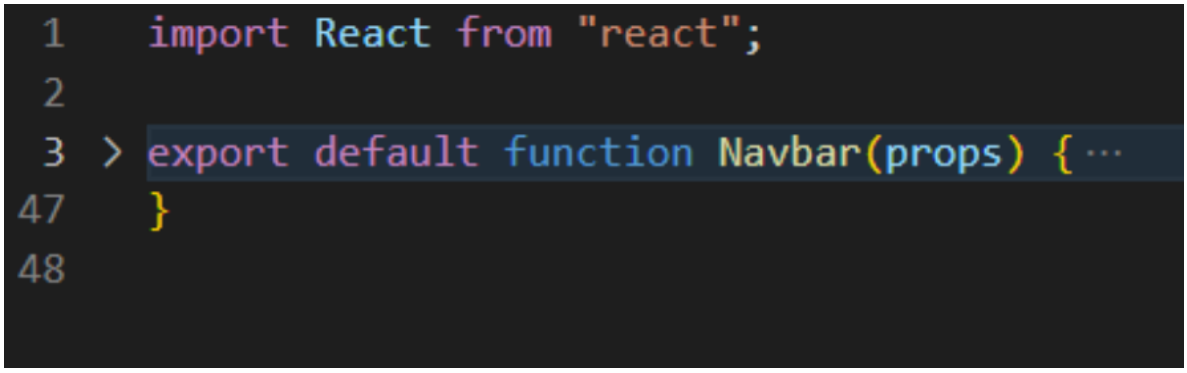
Editor2: Es el encargado de mostrar cada uno de los text áreas donde introduciremos nuestro código y veremos la consola de salida, así mismo tambien veremos cada uno de los botones a funcionar.

```
import React from "react";
import '../Styles/textArea.css'

export default function Editor(props) {
    const handlerChangeEditor = (evt) => {
      props.handlerChange(evt.target.value);
    };

    return (
      <>
        <div class="container">
          <label class="text-area" for="exampleFormControlTextarea1" style={{fontSize: "30px"}}>{props.text}</label>
          <textarea class="form-control" id="exampleFormControlTextarea1" rows="20" onChange={handlerChangeEditor} value={props.value}></textarea>
          {props.comp} {props.comp2}
        </div>
      </>
    );
  }
```

Errores: muestra una tabla de los errores léxicos y sintácticos del programa.

```
import React from "react";

export default function Errores(props) {

  return (
    <>
      <div className="container">
        <table class="table table-dark table-striped">
          <thead>
            <tr>
              <th scope="col">Tipo</th>
              <th scope="col">Descripcion</th>
              <th scope="col">Fila</th>
              <th scope="col">Columna</th>
            </tr>
          </thead>
          <tbody>
            {/* {props.valor.map((value, index) => {
              return (
                <tr key={index}>
                  <th>{value.tipoError}</th>
                  <td>{value.desc}</td>
                  <td>{value.fila}</td>
                  <td>{value.Columna}</td>
```

Navbar: Es la que nos muestra nuestra barra de navegación.



Pages

Index: nos muestra nuestra pagina principal, con todos los componentes unidos en un mismo archivo html.

Para recibir cada uno de los datos ya analizados desde nuestra api utilizamos la librería Service para poder tener la obtención de cada uno de los datos a utilizar.

```
20  const handlerPostParse = () => {
21    //alert(value)
22    Service.parse(value).then((response) => {
23      setResponse(response.consola);
24    });
25
26    Service.parse(value).then((response) => {
27      setArbolito(response.grafito.toString());
28      console.log(arbolito)
29    });
30
31    Service.parse(value).then((response) => {
32      setArr(response.ts_html_error);
33      console.log(arr)
34    });
35
36  };
```

Styles

Esta carpeta es utilizada para almacenar cada uno de los estilos utilizados en cada uno de los componentes.