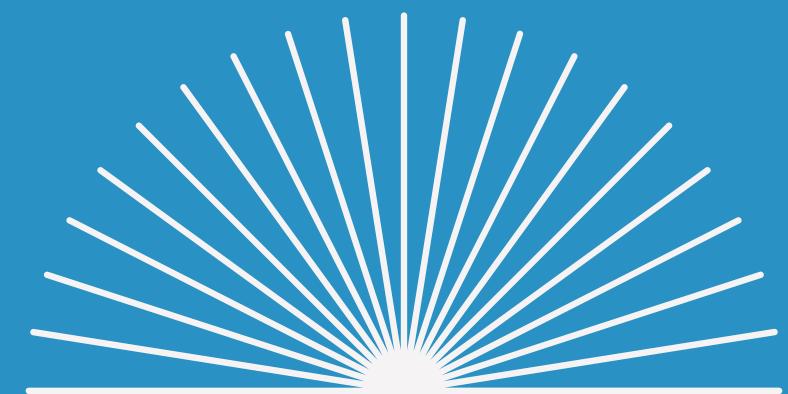


MANUAL TECNICO

PROYECTO 1 PENSUM USAC - MANUAL TECNICO





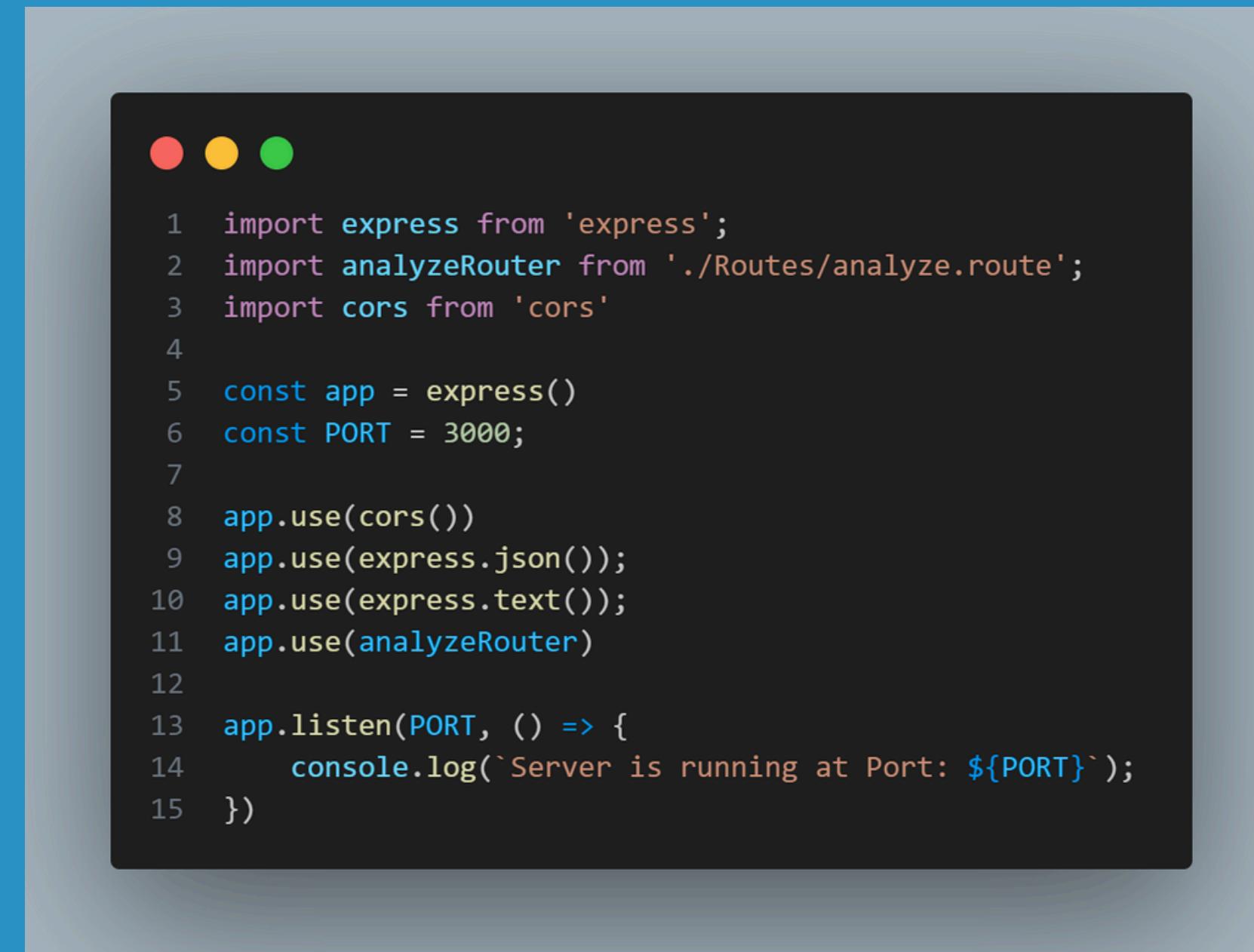
INTRODUCCION

Este es un manual tecnico que muestra como entender el codigo de programacion de la pagina web de Pensum USAC, mostrando sus funcionalidades y la logica que tiene en su backend.



BACKEND

index: El archivo principal del sistema de backend esta dado por el index que contiene lo siguiente en codigo; Hago uso de “cors” y tambien de express para el manejo mas facil de mi proyecto, express para los servicios y cors para poder usar esta api de backend en un entorno global, y uso express con json para el manejo de respuestas, tambien instancio el uso de las rutas que estan en otro archivo, por ultimo levanto el servidor en el puerto 3000.



```
● ● ●

1 import express from 'express';
2 import analyzeRouter from './Routes/analyze.route';
3 import cors from 'cors'
4
5 const app = express()
6 const PORT = 3000;
7
8 app.use(cors())
9 app.use(express.json());
10 app.use(express.text());
11 app.use(analyzeRouter)
12
13 app.listen(PORT, () => {
14   console.log(`Server is running at Port: ${PORT}`);
15 })
```

BACKEND

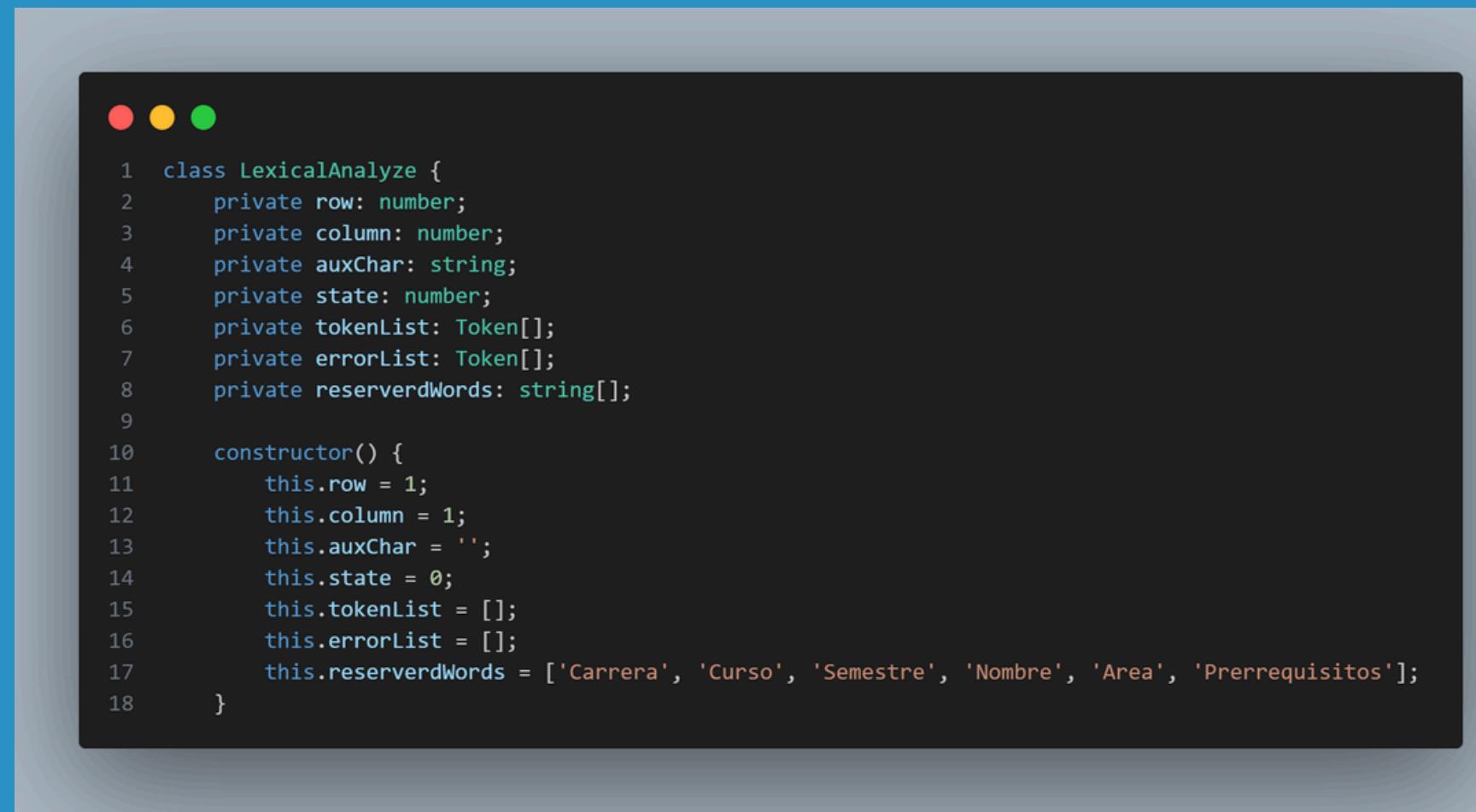
Token: En la parte del archivo de Token hago un constructor con sus variables locales privadas que me ayudaran a crear una estructura sobre que es un token y tambien un enum Type que me va servir para llevar un orden sobre que tipos de Token puedo tener en mi programa, tambien tengo dos Getters que me sirven en otra parte del codigo para poder acceder a los lexemas y los tipos de token.

```
1  enum Type {  
2      UNKNOWN,  
3      PAR_OPEN, // (   
4      PAR_CLOSE, // )  
5      SEMICOLON, // ;  
6      COLON, // :  
7      RESERVERD_WORD,  
8      NUMBER, // number  
9      STRING, // string  
10     BRACK_OPEN, // [  
11     BRACK_CLOSE, // ]  
12     BRACE_OPEN, // {  
13     BRACE_CLOSE, // }  
14     COMMA, // ,  
15 }
```

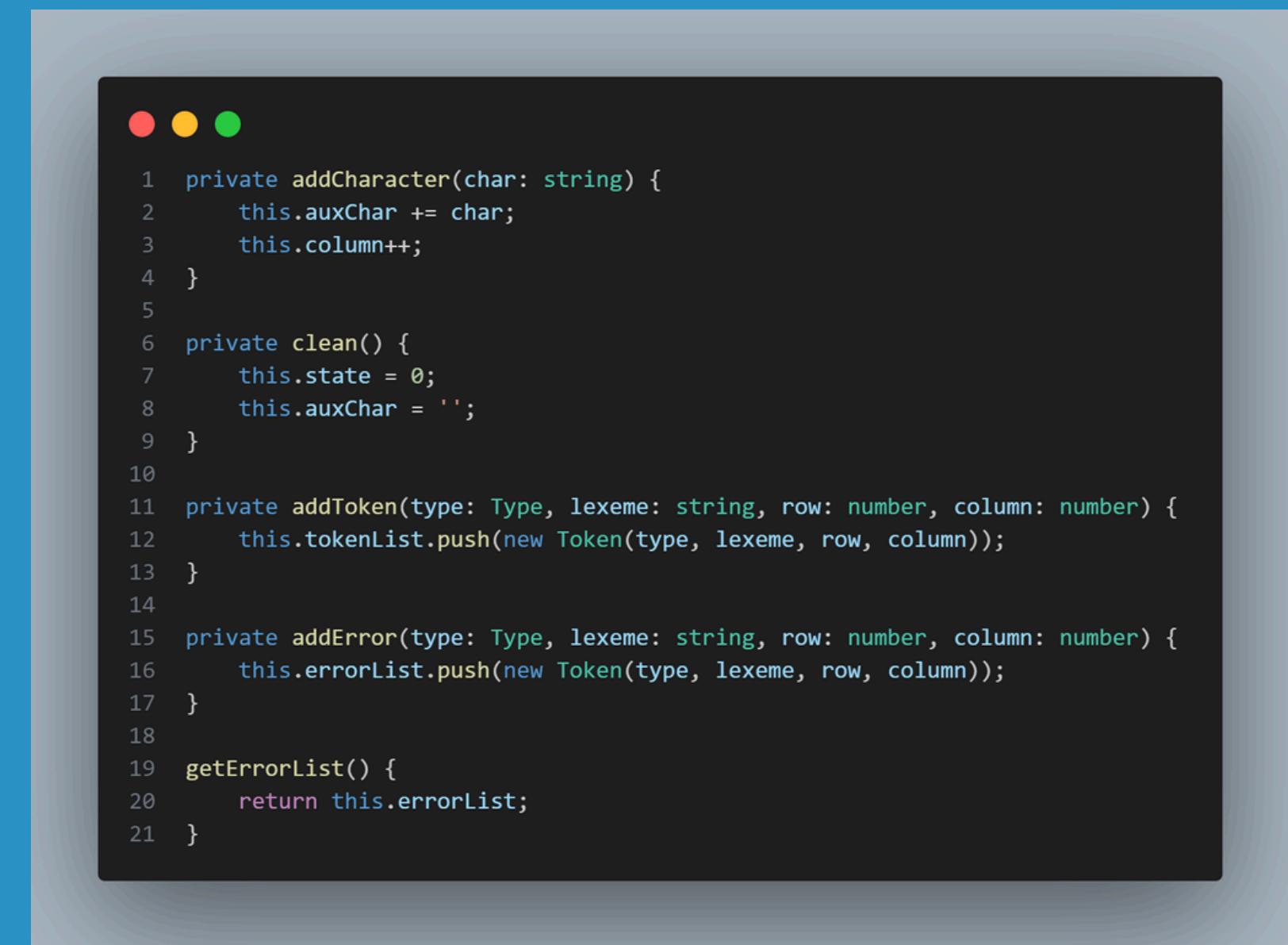
```
1  class Token {  
2      private row: number;  
3      private column: number;  
4      private lexeme: string;  
5      private typeToken: Type;  
6      private typeTokenString: string;  
7  
8      constructor(typeToken: Type, lexeme: string, row: number, column: number) {  
9          this.typeToken = typeToken;  
10         this.typeTokenString = Type[typeToken];  
11         this.lexeme = lexeme;  
12         this.row = row;  
13         this.column = column;  
14     }  
15     getLexeme(): string {  
16         return this.lexeme;  
17     }  
18     getType(): Type {  
19         return this.typeToken;  
20     }  
21 }  
22  
23 export { Type, Token };
```

BACKEND

LexicalAnalyzer: Esta parte del código es de las más importantes, aca es donde se maneja toda la lógica del AFD y como funciona el analizador léxico, lo que hago aca es primero instanciar variables privadas y un constructor que va consistir en tres arreglos, donde uno de los arreglos contiene las palabras reservadas o identificadores un estado para moverse, un carácter auxiliar y uno de fila y uno de columna para poder llevar un orden y un conocimiento. Por otra parte estan las funciones de addCharacter que agrega los caracteres actuales y los auxiliares para llevar orden, la función Clean que limpia caracteres y estados, la función de addToken que agrega los tokens a la lista de tokens y addError que agrega los errores a la lista de errores y por ultimo un getter que devuelve la lista de errores.



```
● ● ●
1 class LexicalAnalyzer {
2     private row: number;
3     private column: number;
4     private auxChar: string;
5     private state: number;
6     private tokenList: Token[];
7     private errorList: Token[];
8     private reservedWords: string[];
9
10    constructor() {
11        this.row = 1;
12        this.column = 1;
13        this.auxChar = '';
14        this.state = 0;
15        this.tokenList = [];
16        this.errorList = [];
17        this.reservedWords = ['Carrera', 'Curso', 'Semestre', 'Nombre', 'Area', 'Prerrequisitos'];
18    }
}
```



```
● ● ●
1     private addCharacter(char: string) {
2         this.auxChar += char;
3         this.column++;
4     }
5
6     private clean() {
7         this.state = 0;
8         this.auxChar = '';
9     }
10
11    private addToken(type: Type, lexeme: string, row: number, column: number) {
12        this.tokenList.push(new Token(type, lexeme, row, column));
13    }
14
15    private addError(type: Type, lexeme: string, row: number, column: number) {
16        this.errorList.push(new Token(type, lexeme, row, column));
17    }
18
19    getErrorList() {
20        return this.errorList;
21    }
}
```

BACKEND

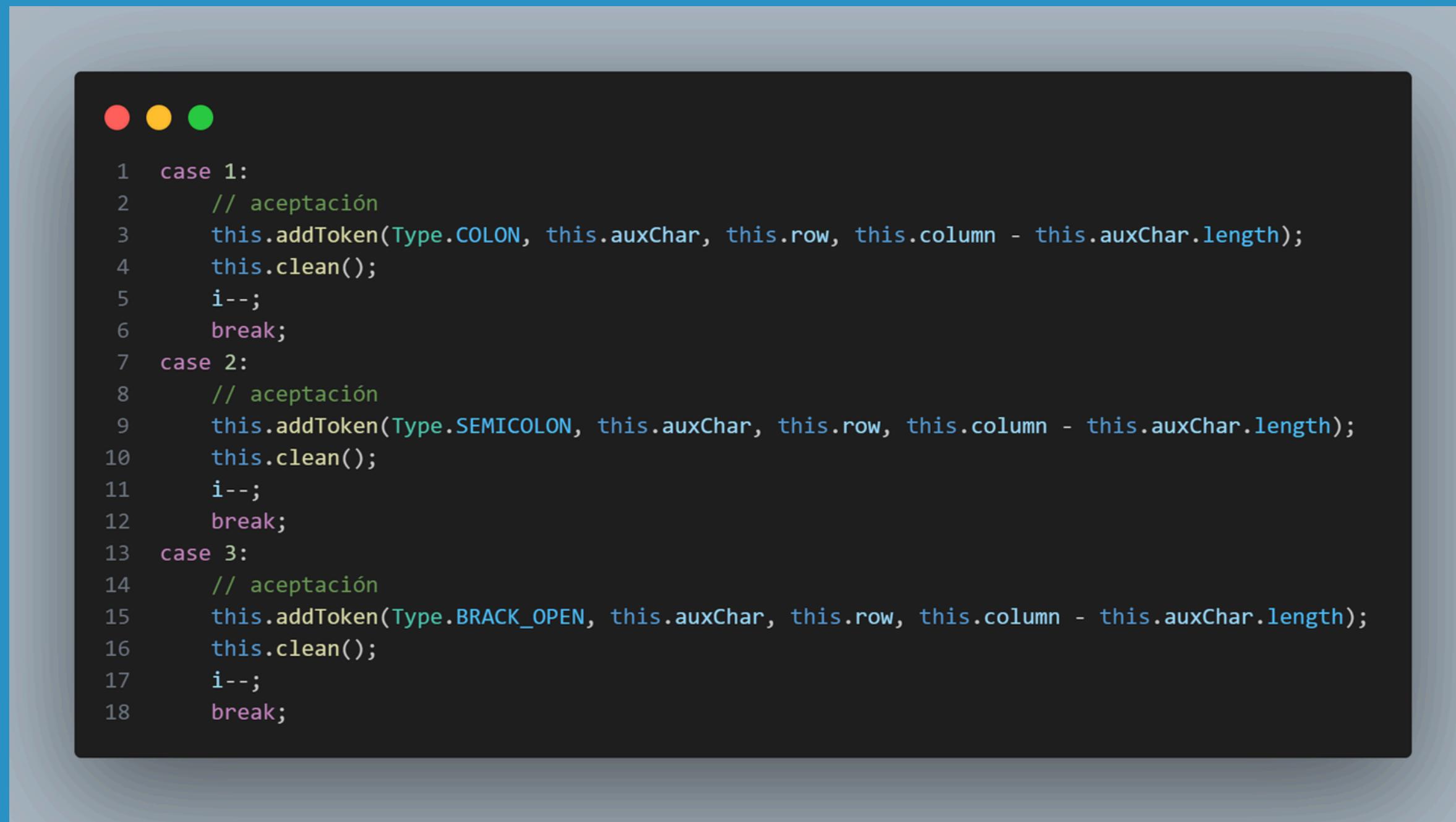
LexicalAnalyzer: Esta parte es muy tecnica, dicha funcion primero inicia con una instancia de un caracter y continua con un bucle for que va recorrer un condicional switch con determinados casos donde en el primer caso enumerado con el 0 va tener dentro otro switch el cual va tener todos lo inicios de palabras reservadas o transiciones principales de salen del estado cero en un AFD entonces dependiendo de en que parte del caso 0 inicia reconociendo lo mandara a ese estado, hay otros estados del estado 0 que ayudan por ejemplo para el espaciado y saltos de linea que son necesarios para saber las columnas y filas.

```
1  scanner(input: string) {  
2      input += '#';  
3      let char: string;  
4      for (let i: number = 0; i < input.length; i++) {  
5          char = input[i];  
6          switch (this.state) {  
7              case 0:  
8                  switch (char) {  
9                      case ':':  
10                         this.state = 1;  
11                         this.addCharacter(char);  
12                         break;  
13                     case ';':  
14                         this.state = 2;  
15                         this.addCharacter(char);  
16                         break;  
17                     case '[':  
18                         this.state = 3;  
19                         this.addCharacter(char);  
20                         break;
```

```
1  case '\t':  
2      this.column += 4;  
3      break;  
4  default:  
5      if (/[^a-zA-Z]/.test(char)) {  
6          // es una letra  
7          this.state = 9;  
8          this.addCharacter(char)  
9          continue;  
10     }  
11     if (/^\d/.test(char)) {  
12         // es un dígito  
13         this.state = 10;  
14         this.addCharacter(char);  
15         continue;  
16     }  
17     if (char == '#' && i == input.length - 1) {  
18         // Se termino el analisis  
19         console.log("Analyze Finished");  
20     } else {  
21         // Error Léxico  
22         this.addError(Type.UNKNOW, char, this.row, this.column);  
23         this.column++;  
24     }  
25 }
```

BACKEND

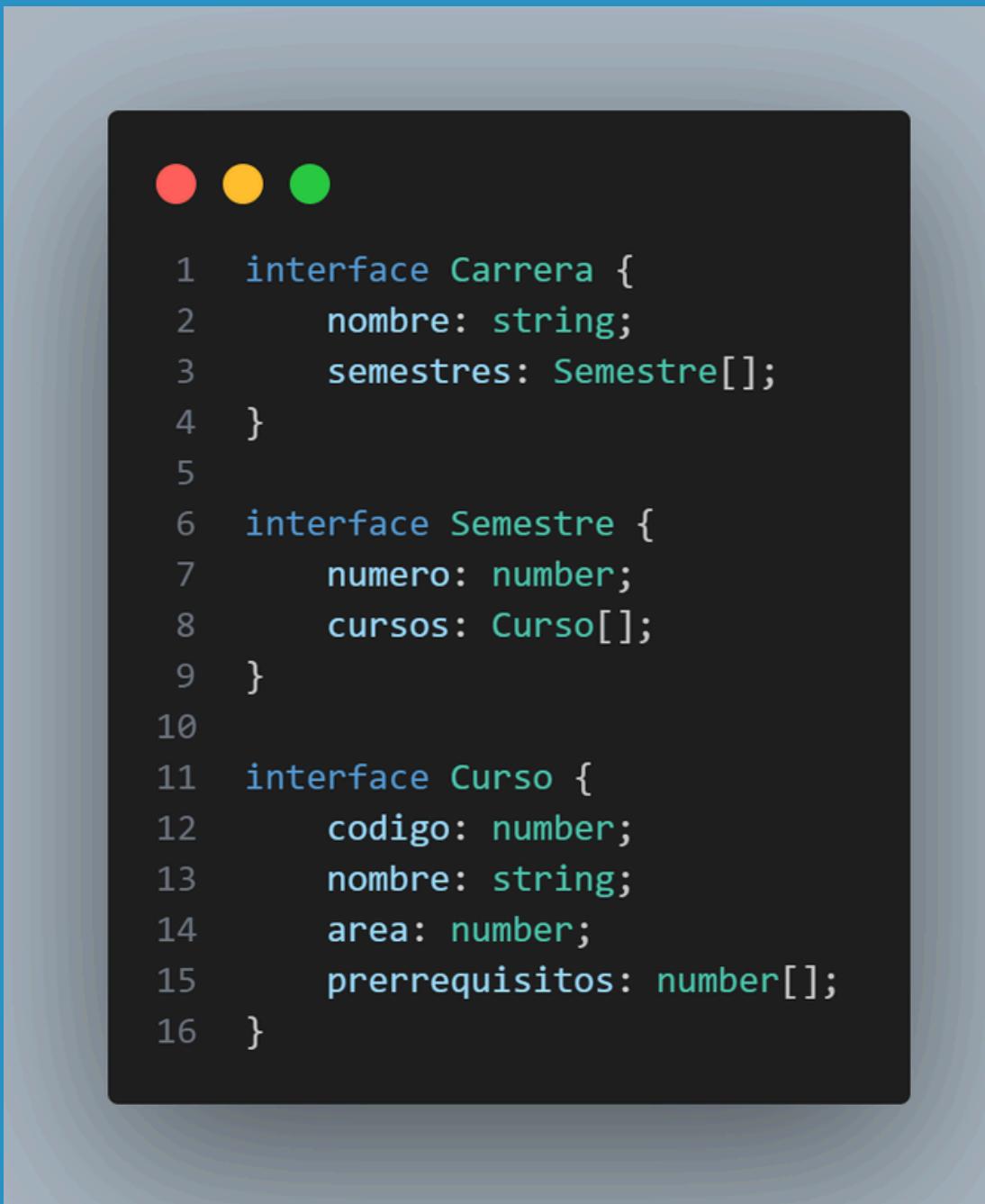
Al momento de dirigirse a uno de los estados fuera del principal que es el cero dependera abran 3 casos para recibir valores y almacenarlos, por ejemplo en el caso 1 al identificar en el estado 0 que es un parentesis abierto ya lo reconoce como Token entonces ira al estado 1 y agregara el token de ‘(‘ a la lista de Tokens luego hace funcion del Clean y regresa las lienas usadas de i para el siguiente caracter a reconocer



```
1 case 1:
2     // aceptación
3     this.addToken(Type.COLON, this.auxChar, this.row, this.column - this.auxChar.length);
4     this.clean();
5     i--;
6     break;
7 case 2:
8     // aceptación
9     this.addToken(Type.SEMICOLON, this.auxChar, this.row, this.column - this.auxChar.length);
10    this.clean();
11    i--;
12    break;
13 case 3:
14     // aceptación
15     this.addToken(Type.BRACK_OPEN, this.auxChar, this.row, this.column - this.auxChar.length);
16     this.clean();
17     i--;
18     break;
```

BACKEND

ProcessToken: En este archivo se crea unas interfaces de: Carrera, Semestre, Curso. Para tener una estructura de cada uno y manejarlos de mejor manera.



```
● ● ●

1 interface Carrera {
2     nombre: string;
3     semestres: Semestre[];
4 }
5
6 interface Semestre {
7     numero: number;
8     cursos: Curso[];
9 }
10
11 interface Curso {
12     codigo: number;
13     nombre: string;
14     area: number;
15     prerequisitos: number[];
16 }
```

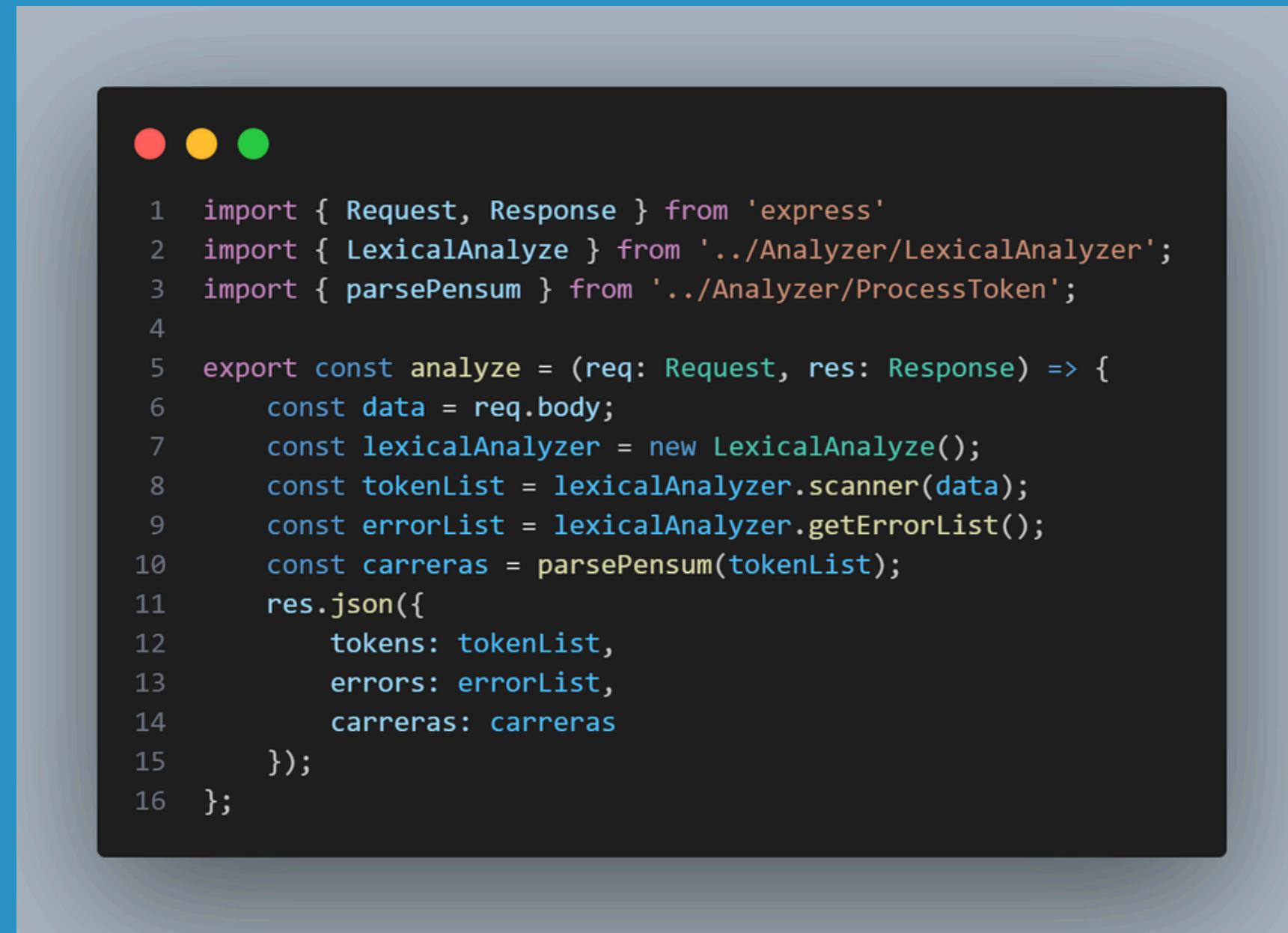
BACKEND

```
1 export function parsePensum(tokens: Token[]): Carrera[] {  
2     let index = 0;  
3     const carreras: Carrera[] = [];  
4     while (index < tokens.length) {  
5         const token = tokens[index];  
6         if (token.getLexeme() === "Carrera") {  
7             index += 2;  
8             const nombreCarrera = tokens[index++].getLexeme().replace(/\g, "");  
9             const carrera: Carrera = { nombre: nombreCarrera, semestres: [] };  
10            if (tokens[index].getType() === Type.BRACK_OPEN) index++;  
11            while (tokens[index] && tokens[index].getLexeme() === "Semestre") {  
12                index += 2; // Semestre + ':'  
13                const numeroSemestre = parseInt(tokens[index++].getLexeme());  
14                if (tokens[index].getType() === Type.BRACE_OPEN) index++; // '{'  
15                const semestre: Semestre = { numero: numeroSemestre, cursos: [] }; //Abrimos un semestre  
16                while (tokens[index] && tokens[index].getLexeme() === "Curso") {  
17                    index += 2; // Curso + ':'  
18                    const codigoCurso = parseInt(tokens[index++].getLexeme());  
19                    if (tokens[index].getType() === Type.BRACE_OPEN) index++; // '{'  
20                    index += 2; // Nombre + ':'  
21                    const nombreCurso = tokens[index++].getLexeme().replace(/\g, "");  
22                    if (tokens[index].getType() === Type.SEMICOLON) index++; // ';'  
23                    index += 2; // Area + ':'  
24                    const areaCurso = parseInt(tokens[index++].getLexeme());  
25                    if (tokens[index].getType() === Type.SEMICOLON) index++; // ';'  
26                    index += 2; // Prerrequisitos + ':'  
27                    if (tokens[index].getType() === Type.PAR_OPEN) index++; // '('  
28                    const prerequisitos: number[] = [];  
29                    while (tokens[index].getType() !== Type.PAR_CLOSE) {  
30                        if (tokens[index].getType() === Type.NUMBER) {  
31                            prerequisitos.push(parseInt(tokens[index].getLexeme()));  
32                        }  
33                        index++;  
34                    }  
35                    if (tokens[index].getType() === Type.PAR_CLOSE) index++; // ')'  
36                    if (tokens[index].getType() === Type.SEMICOLON) index++; // ';'  
37                    if (tokens[index].getType() === Type.BRACE_CLOSE) index++; // '}'  
38                    semestre.cursos.push({ codigo: codigoCurso, nombre: nombreCurso, area: areaCurso, prerequisitos });  
39                }  
40                if (tokens[index].getType() === Type.BRACE_CLOSE) index++; // '}'  
41                carrera.semestres.push(semestre);  
42            }  
43            if (tokens[index].getType() === Type.BRACK_CLOSE) index++;  
44            carreras.push(carrera);  
45        } else {  
46            index++;  
47        }  
48    }  
49    return carreras;  
50 }
```

ParsePensum: En este bloque de código se crea todo el proceso que hará que se devuelva segmentado por partes las carreras, semestres y cursos. Iniciamos un bucle while que recorre los tokens mandados, busca el lexema de carrera y guarda su nombre para meterlo en una interfaz instanciada, luego realiza otro while donde busca el lexema de semestre y al conseguirlo, guarda su número en una instancia de semestre y luego abre otro while donde busca la palabra curso y obtiene sus lexemas y valores de nombre, código, área y prerrequisitos, después de obtener todos los valores y revisar que ya no hayan cursos en el semestre entonces los cierra y los pusea a sus respectivas listas y por último retorna la carrera.

BACKEND

Controlador: Ahora lo que se hace aca es llamar a las funciones de LexicalAnalyzer y processTokens y luego se llaman a cada una de las funcinoes para recibir una respuesta luego de que se manden y reciban los tokens.



```
● ● ●

1 import { Request, Response } from 'express'
2 import { LexicalAnalyze } from '../Analyzer/LexicalAnalyzer';
3 import { parsePensum } from '../Analyzer/ProcessToken';
4
5 export const analyze = (req: Request, res: Response) => {
6     const data = req.body;
7     const lexicalAnalyzer = new LexicalAnalyzer();
8     const tokenList = lexicalAnalyzer.scanner(data);
9     const errorList = lexicalAnalyzer.getErrorList();
10    const carreras = parsePensum(tokenList);
11    res.json({
12        tokens: tokenList,
13        errors: errorList,
14        carreras: carreras
15    });
16}
```

BACKEND

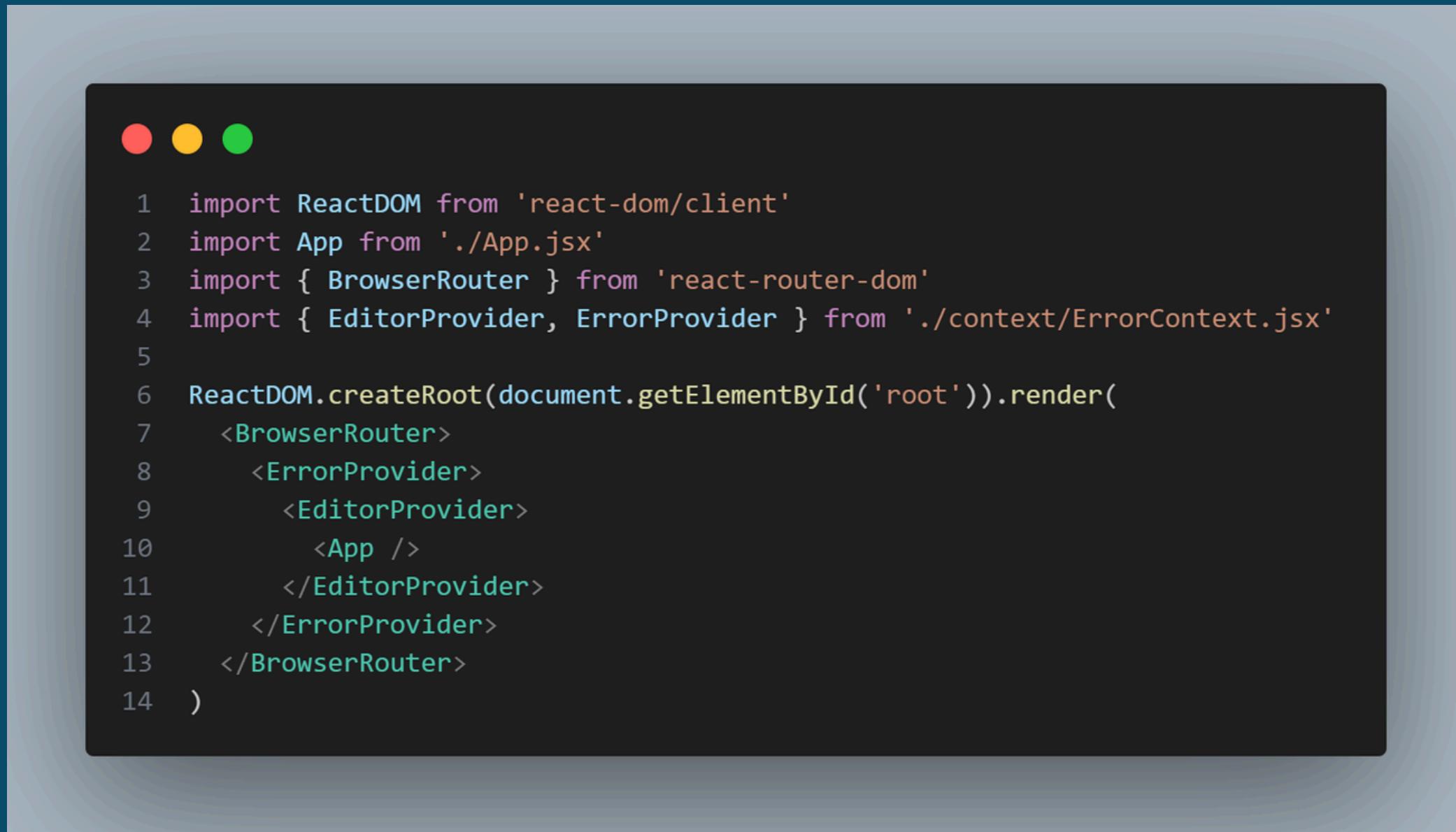
Route: Por ultimo en las rutas llamo a la funcion del controlador y le establezco una ruta en la cual se pueda usar en este caso '/analyze' y llamo a la funcion para que al entrar en esa ruta se hagan esas funciones.



```
1 import { analyze, } from '../Controllers/analyze.controller';
2 import { Router } from 'express';
3 const analyzeRouter = Router();
4
5 analyzeRouter.post('/analyze', analyze);
6
7 export default analyzeRouter;
```

FRONTEND

main: En esta parte solo se estructura como va estar utilizandoce la verificacion de rutas en la web, para que se pueda usar React de la mejor manera.



A screenshot of a terminal window with a dark background and three colored status icons (red, yellow, green) at the top. The terminal displays a script for setting up a React application. The script uses numbered comments (1 through 14) to indicate specific lines of code. The code imports necessary components from 'react-dom/client', 'App.js', 'react-router-dom', and 'context/ErrorContext.js'. It then creates a root element using ReactDOM.createRoot and renders a component tree that includes a 'BrowserRouter', 'ErrorProvider', 'EditorProvider', and 'App' component, all nested within 'EditorProvider' and 'ErrorProvider' components, and finally enclosed in a 'BrowserRouter'.

```
1 import ReactDOM from 'react-dom/client'
2 import App from './App.jsx'
3 import { BrowserRouter } from 'react-router-dom'
4 import { EditorProvider, ErrorProvider } from './context/ErrorContext.jsx'
5
6 ReactDOM.createRoot(document.getElementById('root')).render(
7   <BrowserRouter>
8     <ErrorProvider>
9       <EditorProvider>
10         <App />
11       </EditorProvider>
12     </ErrorProvider>
13   </BrowserRouter>
14 )
```

FRONTEND

App: Aca se crean las distintas rutas que se utilizan en el proyecto web.



The screenshot shows a code editor window with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. The main area contains the following code:

```
1 import { Route, Router, Routes } from 'react-router-dom'
2 import { Inicial } from './Components/Inicial'
3 import { Manuales } from './Components/Manuales'
4 import { Reporte } from './Components/Reporte'
5
6 function App() {
7
8   return (
9     <Routes>
10       <Route path='/' element = {<Inicial />}></Route>
11       <Route path='/Manuales' element = {<Manuales />}></Route>
12       <Route path='/Reporte' element = {<Reporte />}></Route>
13     </Routes>
14   )
15 }
16
17 export default App
```

FRONTEND

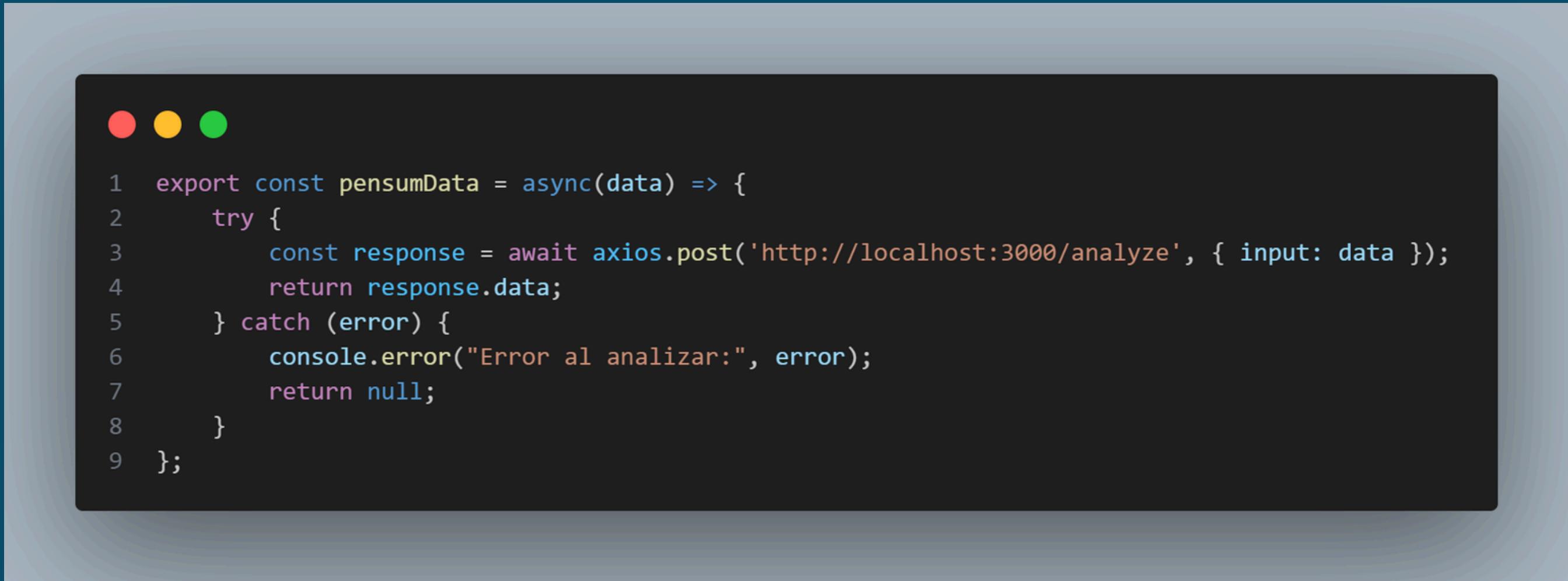
NavBar: NavBar: Aca se crea el Navbar que tiene la pagina web con sus rutas, donde la que mas tiene funcion es la de archivos que sirve para cargar un archivo al editor de texto.



```
1  export const PenumNavbar = ({ onFileClick }) => {
2      const [isResponsive, setIsResponsive] = useState(false);
3
4      const toggleNav = () => {
5          setIsResponsive(!isResponsive);
6      };
7      return (
8          <div className={`topnav${isResponsive ? " responsive" : ""}`}>
9              <ul className="nav nav-pills">
10                 <a className="a-style active">Penum USAC</a>
11                 <a className='a-style' href='/>Home</a>
12                 <Link to="/Reporte" className="a-style">Reporte Error</Link>
13                 <a className='a-style' href="/" onClick={(e) => {
14                     e.preventDefault();
15                     onFileClick();
16                 }}>Archivos</a>
17                 <a className='a-style' href="/Manuales">Manuales</a>
18                 <a className="icon" onClick={toggleNav}>
19                     <i className="fa fa-bars icono" />
20                 </a>
21             </ul>
22         </div>
23     )
24 }
```

FRONTEND

Api: Se importa la libreria de axios para manejar las peticiones al Backend, luego se crea la funcion que va llamar al controlador del backend donde se manda la ruta y los datos necesarios que seria el texto donde van los tokens y retorna los datos pero ya verificados.



The screenshot shows a dark-themed code editor window. At the top left, there are three circular icons: red, yellow, and green. Below them is a block of code:

```
1 export const pensumData = async(data) => {
2     try {
3         const response = await axios.post('http://localhost:3000/analyze', { input: data });
4         return response.data;
5     } catch (error) {
6         console.error("Error al analizar:", error);
7         return null;
8     }
9};
```

FRONTEND



```
1 export const Reporte = () => {
2     const { errors } = useState();
3     return (
4         <>
5             <PensumNavbar />
6             <div className="container mt-4">
7                 <h2>Reporte de Errores</h2>
8                 {errors.length === 0 ? (<p>No se encontraron errores.</p>) : (
9                     <table className="table">
10                         <thead>
11                             <tr>
12                                 <th>No.</th>
13                                 <th>Fila</th>
14                                 <th>Columna</th>
15                                 <th>Lexema</th>
16                                 <th>Token</th>
17                             </tr>
18                         </thead>
19                         <tbody>
20                             {errors.map((error, index) => (
21                                 <tr key={index}>
22                                     <td>{index + 1}</td>
23                                     <td>{error.row}</td>
24                                     <td>{error.column}</td>
25                                     <td>{error.lexeme}</td>
26                                     <td>{error.typeToken}</td>
27                                 </tr>
28                             )));
29                         </tbody>
30                     </table>
31                 )}
32             </div>
33         </>
34     );
35 };
```

Reporte: En esta parte del reporte primero se llama al NavBar para que aparezca en la pagina luego se empieza a hacer un mapeo de los errores si es que encuentra, si no entonces mostrara un texto que dice que no hay errores, despues se hace una tabla en la cual empieza a mostrar todos los errores que puedan haber.

FRONTEND

Inicial: Aca primero usamos estados de variables para las distintas cosas que tenemos en el codigo, luego tenemos la funcion de recargar la pagina para limpiar el editor, luego tambien esta la funcion de handleFileLoad que sirve para mostrar el input de archivos y permite cargar el archivo directo al textarea.



```
1  export const Inicial = () => {
2      const { editorText, setEditorText } = useEditor();
3      const [tokenList, setTokenList] = useState([]);
4      const [carreras, setCarreras] = useState([]);
5      const [carreraSeleccionada, setCarreraSeleccionada] = useState(null);
6      const [cursoSeleccionado, setCursoSeleccionado] = useState(null);
7      const [prerrequisitosRecursivos, setPrerrequisitosRecursivos] = useState([]);
8      const { setErrors } = useError();
9
10     const refreshPage = () => {
11         window.location.reload();
12     };
13
14     const handleFileLoad = (event) => {
15         const file = event.target.files[0];
16         if (!file) return;
17         const reader = new FileReader();
18         reader.onload = (e) => {
19             const content = e.target.result;
20             setEditorText(content);
21         };
22         reader.onerror = (e) => {
23             console.error("Error al leer el archivo:", e);
24         };
25         reader.readAsText(file);
26     };
}
```

FRONTEND

Ahora la siguiente función manda lo que se encuentra en el textarea para el api para que se conecte con el backend y luego continuar con las funciones como si encuentra un error, mandar la alerta y mostrar la lista de tokens, o si no encuentra errores entonces mostrar los tokens y las carreras.



```
1 const handlePensumData = async () => {
2     const result = await pensumData(editorText);
3     if (result) {
4         const errores = result.errors || [];
5         setErrors(errores);
6         if (errores.length > 0) {
7             setTokenList(result.tokens || []);
8             Swal.fire({
9                 icon: 'error',
10                title: 'Errores detectados',
11                text: 'Se detectaron errores. Puedes verlos en la sección de Reporte.',
12                confirmButtonText: 'OK'
13            })
14        } else {
15            setTokenList(result.tokens || []);
16            setCarreras(result.carreras || []);
17        }
18    }
19};
```

FRONTEND

La siguiente función es una función recursiva que busca los requisitos de la clase que se seleccione en la carrera que está, y usa un set para que así sea mejor su función para almacenar y que no hayan valores repetitivos y evitar bucles, luego esta la función que sirve para llamar a la función anterior y mandarle los datos necesarios para que funcione.



```
1 const encontrarPrerrequisitosRecursivos = (codigoCurso, carrera) => {
2     const encontrados = new Set();
3     const buscar = (codigo) => {
4         for (const semestre of carrera.semestres) {
5             for (const curso of semestre.cursos) {
6                 if (curso.codigo === codigo) {
7                     curso.prerrequisitos.forEach(pr => {
8                         if (!encontrados.has(pr)) {
9                             encontrados.add(pr);
10                            buscar(pr);
11                        }
12                    });
13                }
14            }
15        };
16    };
17    buscar(codigoCurso);
18    return Array.from(encontrados);
19};
20
21 const handleCursoClick = (codigoCurso) => {
22     setCursoSeleccionado(codigoCurso);
23     if (carreraSeleccionada) {
24         const prereqs = encontrarPrerrequisitosRecursivos(codigoCurso, carreraSeleccionada);
25         setPrerrequisitosRecursivos(prereqs);
26    }
27};
```

FRONTEND

En el return aparte de mostrar la parte donde se visualizan los tokens y el textarea, lo mas importante recae aqui, en este bloque de codigo se hace un mapeo para crear los botones necesarios por cada carrera que se detecte en el sistema y los crea con sus respectivos nombres encontrados.

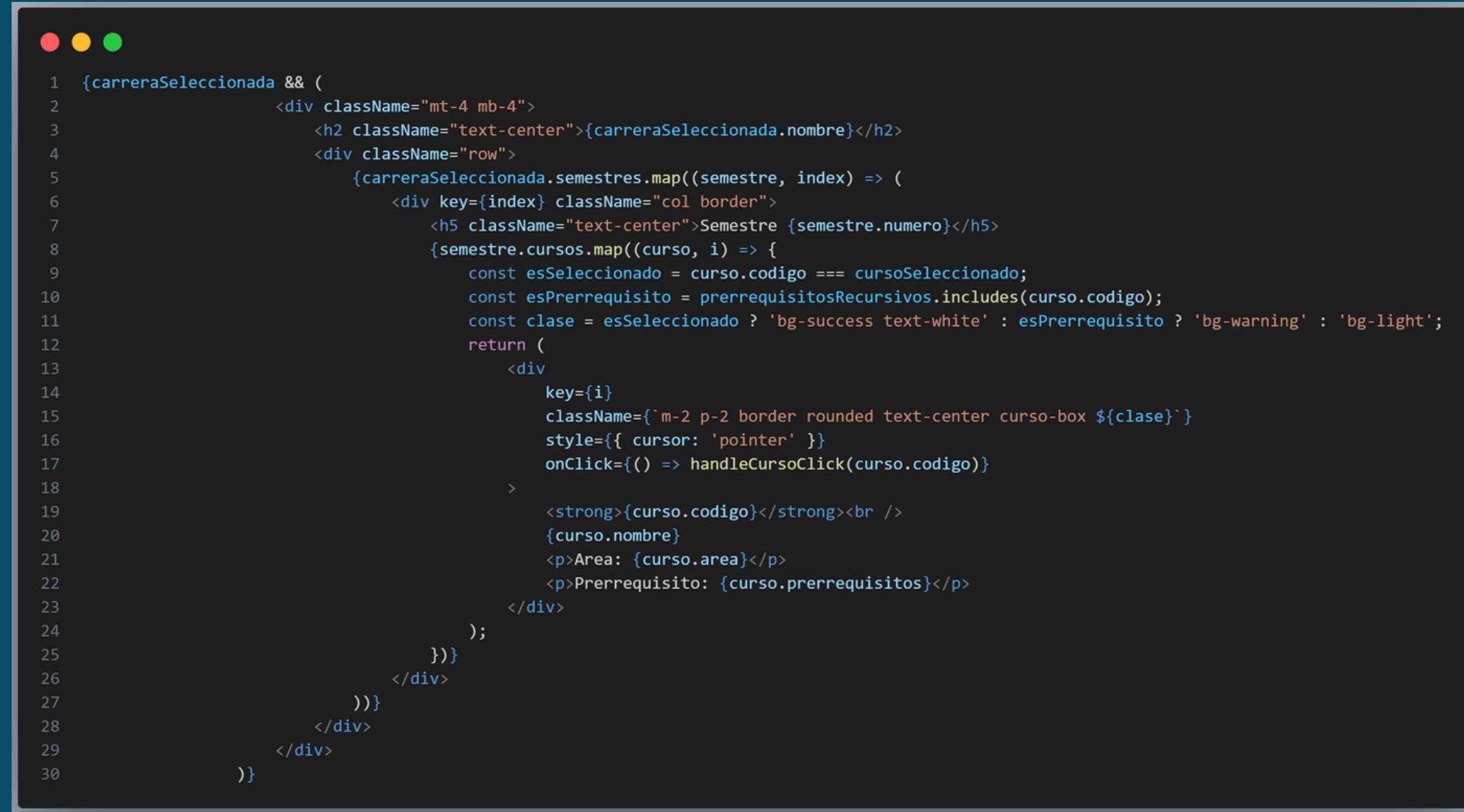


A screenshot of a code editor window showing a snippet of React code. The code is displayed in a dark-themed editor with line numbers on the left. The code itself is a component that maps over an array of careers, creating a button for each career name. The button has a primary class and an onClick event that sets the selected career and clears other state variables.

```
1 <div className="row mt-4">
2   <h1 className="text-center">Carreras</h1>
3   <div className="text-center">
4     {carreras.map((carrera, index) => (
5       <button
6         key={index}
7         className="btn btn-primary m-2"
8         onClick={() => {
9           setCarreraSeleccionada(carrera);
10          setCursoSeleccionado(null);
11          setPrerrequisitosRecursivos([]);
12        }}
13       >
14         {carrera.nombre}
15       </button>
16     )));
17   </div>
18 </div>
```

FRONTEND

En este bloque de código igual del return lo que se hace es otro mapeo, en donde toma la carrera seleccionada por el botón y muestra una tabla con div simulando botones donde en cada div se muestra el nombre del curso y sus otras áreas, esta se repite por cuantos cursos se encuentre y también hace llamado y envío de datos a otras funciones que ya se mostraron para poder hacer parte del código indicado.



```
1 {carreraSeleccionada && (
2     <div className="mt-4 mb-4">
3         <h2 className="text-center">{carreraSeleccionada.nombre}</h2>
4         <div className="row">
5             {carreraSeleccionada.semestres.map((semestre, index) => (
6                 <div key={index} className="col border">
7                     <h5 className="text-center">Semestre {semestre.numero}</h5>
8                     {semestre.cursos.map((curso, i) => {
9                         const esSeleccionado = curso.codigo === cursoSeleccionado;
10                        const esPrerrequisito = prerrequisitosRecursivos.includes(curso.codigo);
11                        const clase = esSeleccionado ? 'bg-success text-white' : esPrerrequisito ? 'bg-warning' : 'bg-light';
12                        return (
13                            <div
14                                key={i}
15                                className={` m-2 p-2 border rounded text-center curso-box ${clase}`}
16                                style={{ cursor: 'pointer' }}
17                                onClick={() => handleCursoClick(curso.codigo)}
18                            >
19                                <strong>{curso.codigo}</strong><br />
20                                {curso.nombre}
21                                <p>Area: {curso.area}</p>
22                                <p>Prerrequisito: {curso.prerrequisitos}</p>
23                            </div>
24                        );
25                    })
26                )
27            ))
28        )
29    )
30)}
```

CONCLUSIÓN

- Esas son todas las funciones que tiene la pagina web diseñada para el usuario y su uso de ayuda para elegir sus cursos para aprobar.
- Muestra cada una de las funciones tanto en Backend como en Frontend y como usarlas y por si se quiere modificar el codigo en algun momento.

Nombre: Alberto Moisés Gerardo Lémus Alvarado

Carne: 202400999

