202111718

Manual Técnico

Proyecto 2 Compi 1

Walter Javier Santizo Mazariegos

Objetivos

Objetivo General:

- Aplicar los conocimientos sobre análisis léxico y sintáctico de un compilador para desarrollar un intérprete sencillo funcional para el lenguaje especificado.

Objetivos Específicos:

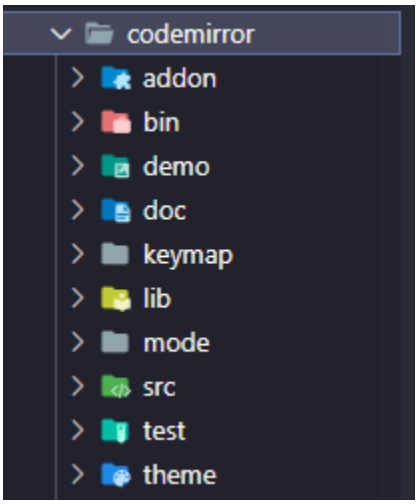
- Generar un analizadores léxicos y sintácticos utilizando la herramienta JISON.

- el manejo de errores léxicos y sintácticos correctamente durante la interpretación.

- La implementación del patrón de diseño intérprete para el manejo de acciones gramaticales en javascript.

Code Mirror

CodeMirror es un componente editor de código para la web. Puede ser utilizado en sitios web para implementar un campo de entrada de texto con soporte para muchas características de edición y cuenta con una interfaz de programación completa para permitir una mayor extensión.

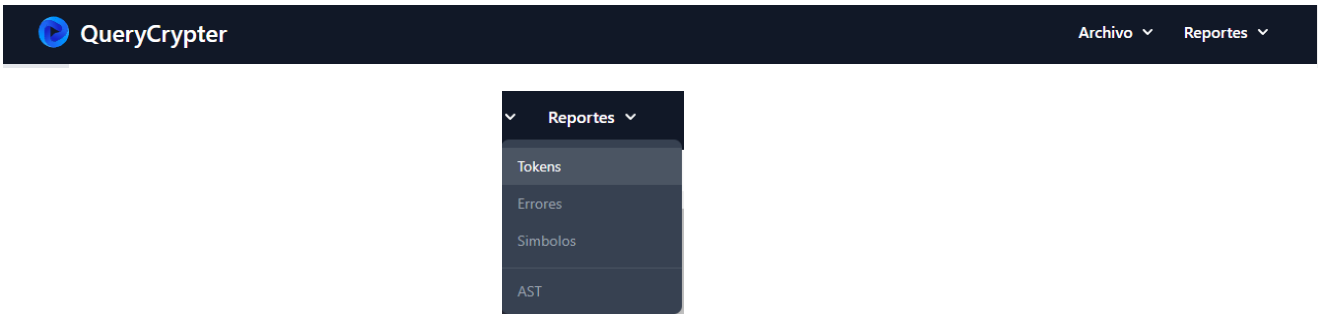


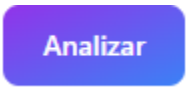
Se instalo de forma local el code mirror para utilizar la librería de forma completa para estilizar la consola de entrada y la consola de salida (numero de línea, tema, coloreo de fuente)

Tailwind

Tailwind CSS es un framework de diseño para la construcción rápida y eficiente de interfaces de usuario en sitios web y aplicaciones. A diferencia de otros frameworks como Bootstrap, Tailwind no proporciona componentes prediseñados, sino que se enfoca en proporcionar utilidades de clases CSS de bajo nivel para construir interfaces personalizadas.

Flowbite es la librería de componentes estilizados que se utilizaron para la construcción de la navbar y tablas de reportes así como los botones

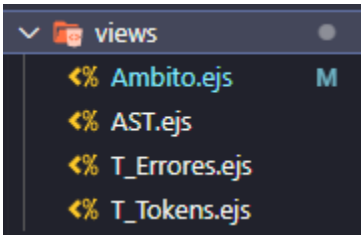




EJS

EJS, que significa "Embedded JavaScript", es un motor de plantillas para JavaScript que permite a los desarrolladores incorporar código JavaScript dentro de documentos HTML. EJS facilita la creación de vistas dinámicas en aplicaciones web, lo que significa que puedes generar HTML de forma dinámica en función de datos o variables.

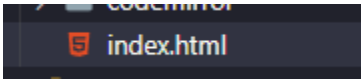
En lugar de incrustar todo el HTML en cadenas de texto en tu código JavaScript, puedes utilizar EJS para crear plantillas HTML con sintaxis especial. Estas plantillas pueden contener código JavaScript que se ejecutará en el servidor antes de enviar la respuesta al cliente.



Para un mayor dinamismo se uso ejs para renderizar los reportes del lado del servidor y servirlos por medio de la api rest

Frontend

Se utilizo una sola pagina web servida desde el live server en el puerto 5000 donde se utilizo la etiqueta script para llamar la api rest y traer los datos necesarios del backend




```
<!-- ================================ CODEMIRROR ========================== -->
<script src="codemirror/lib/codemirror.js"></script>
<script src="codemirror/mode/javascript/javascript.js"></script>
<script src="codemirror/mode/sql/sql.js"></script>
<script src="codemirror/addon/edit/closebrackets.js"></script>
<link rel="stylesheet" href="codemirror/lib/codemirror.css">
<link rel="stylesheet" href="codemirror/theme/dracula.css">

<!--flowbite cdn-->
<link href="https://cdnjs.cloudflare.com/ajax/libs/flowbite/1.8.1/flowbite.min.css" rel="stylesheet" />
<!--Tailwind cdn-->
<script src="https://cdn.tailwindcss.com"></script>
```

Ya que hicimos el proyecto localmente solo debemos colocar en el header los cdn de flowbyte y tailwind además de importar las librerías javascript y css de code mirrror

```
let editor = CodeMirror.fromTextArea(document.getElementById("editor1"),{
        lineNumbers         : true,
        mode                : "sql",
        theme               : "dracula",
        autoCloseBrackets   : true,
        tabSize: 4,
        //lineWrapping : true,
        styleActiveLine     : true
    });
```

De esta forma se estiliza el elemento textarea le debemos poner un id que lo identifique

```
<div id="container-content">
  <section id="selected-content">
    <textarea id="editor1"></textarea>
  </section>
```

De la misma manera se debe editar la consola de salida

```
fetch('http://localhost:4000/analizar',{
  method: 'POST',
  body: JSON.stringify({
    entrada: editor.getValue()
  }),
  headers: {
    'Content-Type': 'application/json'
  }
})  .then(res => {
      // Esperar a que la promesa de res.json() se resuelva
      return res.json();
    })
    .then(data => {
      let texto = "";
      data.salida.forEach((element, index) => {
        texto += element;
        if (index < data.salida.length - 1) {
          texto += "\n"; // Agregar salto de línea si no es
        }
      });
      consola.setValue(texto);
    })
    .catch(error => {
      // Capturar errores
      console.error('Error:', error);
```

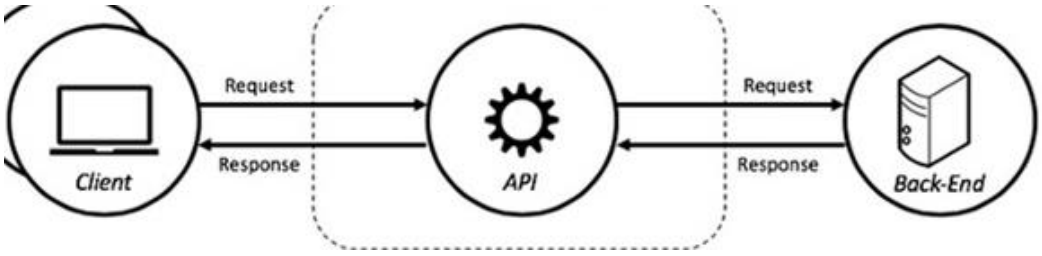
Para consumir el api rest desde el evento listener del botón ingresamos el siguiente código que manda a traer la data desde el api para posteriormente colocarla en la consola de salida

```
fetch('http://localhost:4000/TablaToken',{
    method: 'GET',
    headers: {
      'Content-Type': 'application/json'
    }
  })  .then(res => {
        // Esperar a que la promesa de res.json() se resuelva y devolver el resultado
        console.log(res.url)
        //abrir en una nueva pestaña
        window.open(res.url, '_blank');
      })
      .then(data => {

      })
      .catch(error => {
        // Capturar errores
        console.error('Error:', error);
      });
```

En el caso de las plantillas que se renderizan del lado del servidor las servimos de esta forma abriendo la pagina web en otra pestaña

API

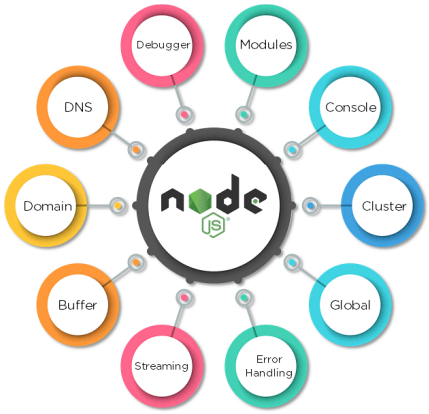
Una API REST (Representational State Transfer) es un conjunto de reglas y convenciones para el diseño de servicios web que utiliza el protocolo HTTP para la comunicación. Las APIs RESTful permiten a los sistemas interactuar entre sí a través de la web de una manera simple y estandarizada.



Para realizar la api se utilizaron las siguientes herramientas

Node js

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.



Los comandos a ejecutar para tener el ambiente de desarrollo de node son los siguientes

```
npm init
npm install express
npm install ejs
npm install --save-dev @types/node
npm install nodemon --save-dev
npm install -g jison
```

Una vez tenemos instalado los paquetes procedemos a crear los scripts en nuestro packaje.json

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js",
  "generar": "jison ./analizador/gramatica.jison -o ./analizador/parser.js",
  "dev": "nodemon index.js"
},
```

Los 2 mas importantes son el generar y el dev el generar se encarga de generar la gramática en código js por medio de la herramienta jison mientras que el comando dev se encarga de ejecutar nodemon que escucha los archivos en busca de un cambio para así reiniciar el servidor

```
> npm run generar

> server@1.0.0 generar
> jison ./analizador/gramatica.jison -o ./analizador/parser.js
```

```
[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Server running on port 4000
```

index.js

```
server > index.js > app.listen() callback
1  const app = require('./app.js');
2  const port = 4000;
3
4  app.listen(port, () => {
5      console.log(`Server running on port ${port}`);
6  });
7
8
```

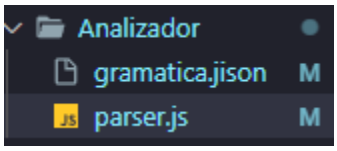
app.js

```
server > app.js > ...
1  const express=require('express');
2  const cors=require('cors');
3  const morgan=require('morgan');
4
5  const app=express();
```

De esta forma levantamos un api con node js

JISON

Jison toma una gramática libre de contexto como entrada y genera un archivo JavaScript capaz de analizar el lenguaje descrito por esa gramática. Luego puedes utilizar el script generado para analizar entradas y aceptarlas, rechazarlas o realizar acciones basadas en la entrada. Si estás familiarizado con Bison o Yacc, u otros clones similares, ya estás casi listo para empezar.



Por medio de la api mandamos a llamar el archivo parser que se encarga de analizar los caracteres de una gramática y producir salidas

```
const analizar = (req, res) =>{
```

En el controlador definimos la función

```
const express = require('express');
const router = express.Router();

//Imports controller
const indexController = require('../controller/index.controller.js')


// Rutas
router.get("/", indexController.index);
router.post("/analizar", indexController.analizar);
```

Desde las rutas mandamos a llamar la funcion

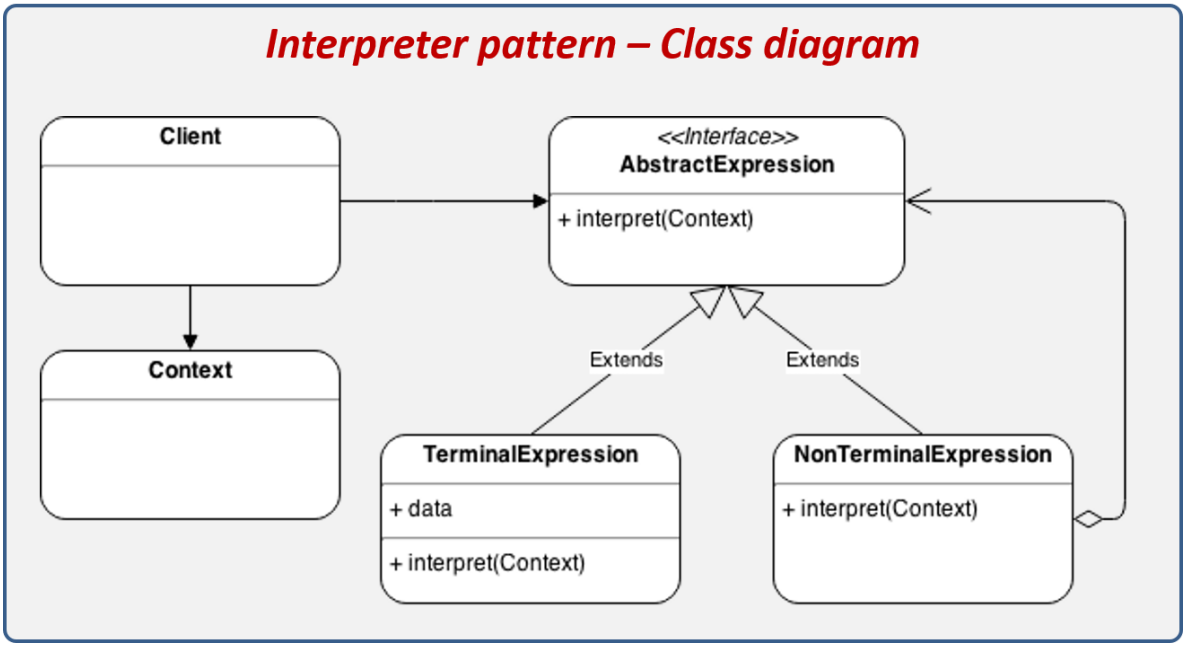
```
app.use("/", indexRoutes)
app.use("/analizar", indexRoutes)
```

Mediante el app usamos el endpoint

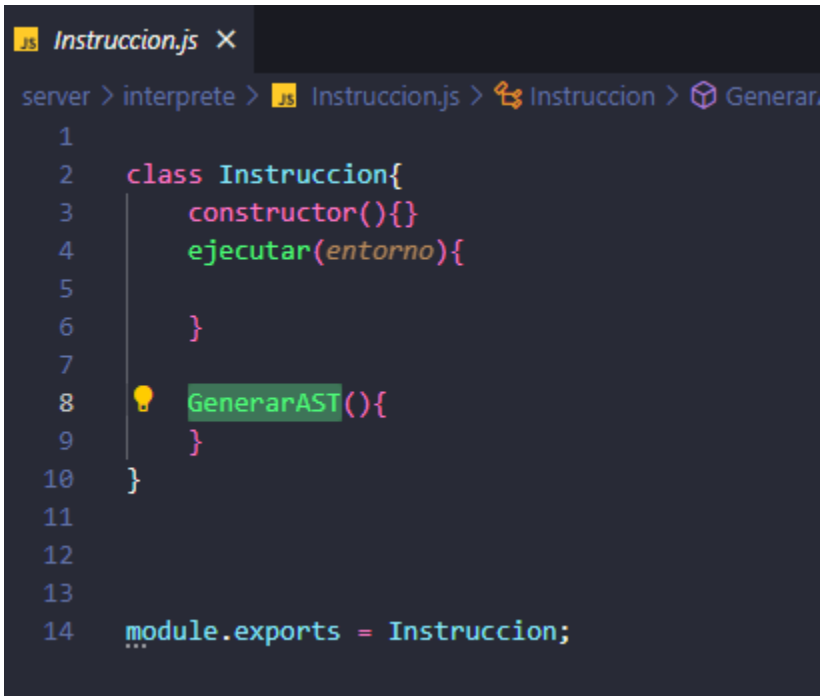
Patrón Interprete

El patrón intérprete (Interpreter Pattern en inglés) es un patrón de diseño de software que se utiliza para definir la gramática de un lenguaje y proporcionar un intérprete para interpretar sentencias en ese lenguaje. Es parte de la categoría de patrones de comportamiento.

En términos más simples, el patrón intérprete se utiliza para definir una gramática para un idioma y proporcionar un intérprete que interprete las oraciones de ese idioma. Este patrón es útil cuando se necesita interpretar expresiones complejas o realizar operaciones específicas en un lenguaje particular.



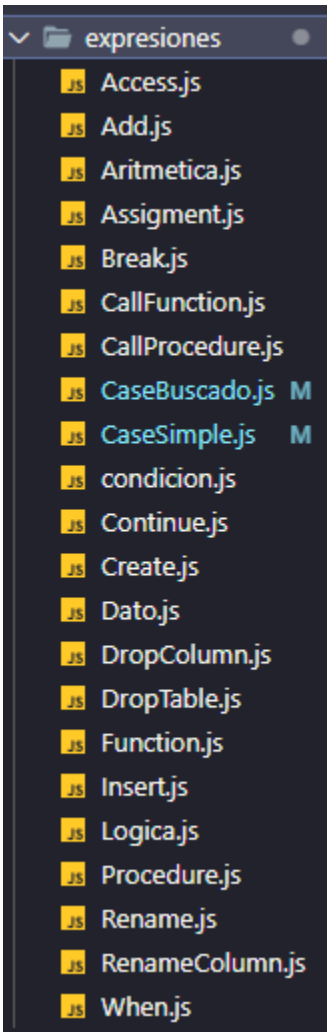
Como lo indica el patrón debemos de tener 1 clase abstracta que será el molde para todas las expresiones e instrucciones de nuestro programa



Esta clase se encargara de funcionar como plantilla para la clase Expresión e Instrucción que pueden ser mas de 2

Expresión

Una expresión es una combinación de valores, variables, operadores y llamadas a funciones que, cuando se evalúa, produce un resultado. Las expresiones son evaluadas para obtener un valor y pueden ser de diferentes tipos de datos, como números, cadenas de texto, booleanos, objetos, etc. Las expresiones se utilizan en programación para realizar cálculos y manipulaciones de datos.



Por ejemplo

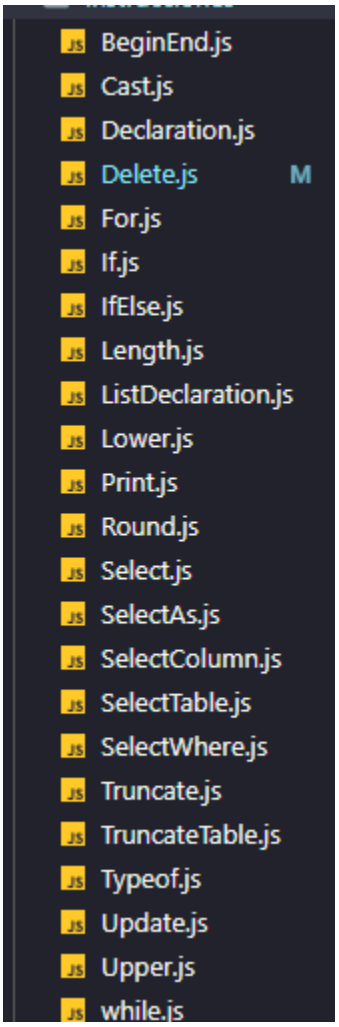
```
class Aritmetica extends Instruccion {
    constructor(izquierda, operador, derecha, linea, columna) {
        super();
        this.izquierda = izquierda;
        this.derecha = derecha;
        this.operador = operador;
        this.linea = linea;
        this.columna = columna;
        this.valor = null;
    }
```

Esta es una expresión Aritmética que se encarga de operar por medio de la funcion ejecutar que es la siguiente

```
ejecutar(entorno) {
    let izquierda=this.izquierda.ejecutar(entorno);
    let derecha;



    if(this.derecha!=null){
        derecha=this.derecha.ejecutar(entorno);
    }

    switch(this.operador){
        case "+":
            if(izquierda.tipo=="INT" && derecha.tipo=="INT"){
                this.valor= izquierda.valor + derecha.valor;
                return new Dato(this.valor, "INT",this.linea,this.columna);
            }else if(izquierda.tipo=="DOUBLE" && derecha.tipo=="DOUBLE"){
                this.valor= izquierda.valor + derecha.valor;
                return new Dato(this.valor, "DOUBLE",this.linea,this.columna);
            }else if(izquierda.tipo=="INT" && derecha.tipo=="DOUBLE"){
                this.valor= izquierda.valor + derecha.valor;
                return new Dato(this.valor, "DOUBLE",this.linea,this.columna);
            }else if(izquierda.tipo=="DOUBLE" && derecha.tipo=="INT"){
```

Instrucción

Una instrucción, por otro lado, es una línea completa de código que realiza una acción. Las instrucciones son declaraciones completas que ejecutan una tarea en el programa. Las instrucciones pueden consistir en asignaciones de variables, llamadas a funciones, estructuras de control como bucles y condicionales, y otras operaciones que afectan el flujo del programa. Las instrucciones no tienen un valor por sí mismas y se utilizan para realizar acciones o cambiar el estado del programa.



Por ejemplo

```
class Print extends Instruccion {
    constructor(expresion) {
        super();
        this.expresion = expresion;
    }
```

Print recibe una expresión y la ejecuta de esta forma

```
ejecutar(entorno) {
    let expresion=this.expresion.ejecutar(entorno);
    ConsolaSalida.push(expresion.valor);
}
```

Views y Reportes

Como los reportes se renderizan del lado del servidor se utilizo ejs y la sintaxis ejs es la siguiente

```
<% for (let i = 0; i < Simbolos.length; i++) { %>
    <% for (var [key, value] of Simbolos[i].TablaSimbolos) { %>
    <tr class="bg-white border-b dark:bg-gray-800 dark:border-gray-700">
        <th scope="row" class="px-6 py-4 font-medium text-gray-900 whitespace-nowrap d
            <%=key%>
        </th>
        <td class="px-6 py-4">
            <%=value.valor%>
        <td class="px-6 py-4">
            <%=value.tipo%>
        </td>
        <td class="px-6 py-4">
            <%=Simbolos[i].nombre%>
        </td>
        <td class="px-6 py-4">
            <%=value.linea%>
        </td>
        <td class="px-6 py-4">
            <%=value.columna%>
        </td>
    </tr>
    <% } %>
```

```
const GenerarAST= (req, res) =>{

    let texto=`https://quickchart.io/graphviz?graph=digraph L{ordering="out" 0[label="instrucciones"]$
    {encodeURIComponent(cuerpo)}}`
    res.render('AST.ejs',{ imageUrl: texto});
}
```


```
const GenerarAST= (req, res) =>{

    let texto=`https://quickchart.io/graphviz?graph=digraph L{ordering="out" 0[label="instrucciones"]$
    {encodeURIComponent(cuerpo)}}`
    res.render('AST.ejs',{ imageUrl: texto});
}
```