Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 1

Primer Semestre 2024

Ing. Kevin Lajpop

Carlos Acabal

CompiScript+

Proyecto 2

Manual Técnico

César Fernando Sazo Quisquinay

202202906

Objetivo Principal:

Crear un sistema capaz de realizar distintas operaciones por medio de una sintaxis en específico, que permite funciones que la mayoría de los lenguajes de programación a día de hoy poseen, todo esto a partir de una colección de datos que se hacen por medio de un archivo de texto, para esto se utilizaran los conocimientos sobre analizadores léxicos, sintácticos y semánticos para poder ejecutar las tareas y que por medio del archivo del texto se puedan mandar las instrucciones.

El proyecto se realizó en el lenguaje de programación JavaScript utilizando distintas librerías listadas acá:

- Jison
- Node.js

Jison

Jison es una herramienta para analizar y generar analizadores léxicos y sintácticos. Básicamente, es un generador de analizadores gramaticales que toma una definición de una gramática (un conjunto de reglas y tokens) y produce un analizador que puede procesar texto según esas reglas. Se utiliza para crear compiladores, intérpretes o procesadores de texto que necesitan entender estructuras complejas de datos o lenguajes de programación.

Es un proyecto de código abierto y se considera el equivalente en JavaScript de Bison, una herramienta similar que es parte del proyecto GNU. Jison permite a los desarrolladores de JavaScript construir analizadores potentes y eficientes de una manera relativamente sencilla, usando gramáticas escritas en un estilo similar al utilizado por Bison y Yacc (Yet Another Compiler Compiler). Esto hace que sea muy útil para proyectos que necesitan analizar datos o lenguajes de programación dentro del entorno de JavaScript.

Node.js

Node.js es un entorno de ejecución de código abierto y multiplataforma para la programación en JavaScript. Permite a los desarrolladores ejecutar JavaScript en el servidor, fuera del navegador. Esto hace posible usar JavaScript para todo tipo de aplicaciones, desde servidores web hasta herramientas de desarrollo y aplicaciones de escritorio. Node.js utiliza el motor V8 de Google Chrome para interpretar JavaScript, lo que le permite ser muy rápido y eficiente en el manejo de operaciones asíncronas y basadas en eventos.

Un elemento fundamental que pose Node.js es NPM, que significa Node Package Manager, es el sistema de gestión de paquetes que acompaña a Node.js. Permite a los desarrolladores instalar y manejar las dependencias (bibliotecas y herramientas) necesarias para sus proyectos. NPM también facilita el compartir y distribuir código a través de paquetes, lo que permite a los desarrolladores reutilizar código de otros desarrolladores fácilmente y manejar las versiones de las dependencias de manera eficaz. Es una herramienta esencial para el desarrollo moderno de aplicaciones JavaScript, ofreciendo una vasta biblioteca de paquetes disponibles que se pueden integrar en cualquier proyecto.

Archivos destacados:

La principal columna del programa es el archivo .jison que contiene el análisis léxico junto con el análisis sintáctico, a comparación de otros analizadores que se guardan en 2 archivos diferentes. En la parte de arriba tenemos declarada toda la parte del análisis, léxico, es decir todo lo que va a reconocer el lenguaje, así poder guardarlo en tokens para el análisis sintáctico, básicamente lo que se hace es declarar la cadena de texto o expresión regular para poder guardarla en un token con cualquier nombre.

```
"=="                    { console.log("ENTRO A DOBLE IGUAL"); return 'ORIGUAL'; }
"!="                    { return 'ORDIF'; }
"<="                    { return 'ORMENORIGUAL'; }
">="                    { return 'ORMAYORIGUAL'; }
"||"                    { return 'OR'; }
"&&"                    { return 'AND'; }
"!"                     { return 'NOT'; }
"<<"                    { return 'ASIGN'; }
"++"                    { return 'INCREASE'; }
"--"                    { return 'DECREASE'; }
"int"                   { return 'INT'; }
"double"                { return 'DOUBLE'; }
"bool"                  { return 'BOOL'; }
"char"                  { return 'CHAR'; }
"std::string"           { return 'STD'; }
"std::toString"         { return 'STS'; }
"="                     { return 'IGUAL'; }
":"                     { return 'PUNTOS' }
"true"                  { return 'TRUE'; }
"false"                 { return 'FALSE'; }
","                     { return 'COMA'; }
```

```
// Expresiones Regulares
([a-zA-Z])[a-zA-Z0-9_]* { console.log('Token: ID, Valor: ' + yytext); return 'ID'; }    //Nombre de variables
[0-9]+("."[0-9]+)\b     { console.log('Token: DECIMAL, Valor: ' + yytext); return 'DECIMAL'; }
[0-9]+\b                { console.log('Token: ENTERO, Valor: ' + yytext); return 'ENTERO'; }
```

Luego de haber reconocido todos los caracteres, estos son guardados como tokens para poder seguir con la siguiente fase del compilador, que es la fase del análisis sintáctico, que recibe todos los tokens generados, está la veremos reflejada posteriormente al análisis léxico ya que ambas se encuentran en el mismo archivo, que es donde se declara el orden de los tokens para así poder reconocer una gramática, todas estas "producciones" son las instrucciones para realizar las funcionalidades del lenguaje.

```
init : entrada EOF  { console.log("Entrada procesada con éxito."); retorno = { instrucciones: $1 }; return retorno; }
;

entrada : entrada sentencia { $1.push($2); $$=$1; }
        | sentencia         { $$=[$1]; }
;

sentencia : declaracion_functions             { $$ = $1; }
        | declaracion_variable PUNTOYCOMA     { $$ = $1; }
        | declaracion_array PUNTOYCOMA        { $$ = $1; }
        | expresion PUNTOYCOMA                { $$ = $1; }
        | sent_transf PUNTOYCOMA              { $$ = $1; }
        | sent_if                             { $$ = $1; }
        | sent_switch                         { $$ = $1; }
        | sent_while                          { $$ = $1; }
        | sent_for                            { $$ = $1; }
        | sent_dowhile PUNTOYCOMA             { $$ = $1; }
        | print PUNTOYCOMA                    { $$ = $1; }
        | ID IGUAL expresion PUNTOYCOMA       { $$ = asignarValor($1, $3); }
        | EXECUTE llamada PUNTOYCOMA          { $2.execute = true; $$ = $2; }
        | error PUNTOYCOMA    { console.log("Error al procesar la entrada.");
    global.reportes.agregarError({tipo: "Sintactico", error: $1, linea: this._$.first_line, columna : this._$.first_column}); }
;
```

Luego de haber generado el archivo jison hay un aspecto importante a mencionar, y es que este no es un ejecutable de JavaScript como tal, por ende, se necesita la librería jison para poder ejecutar este archivo que genera el archivo JavaScript que si ser ejecutable para nuestro programa, esta librería esa accesible desde Node.js (npm) posteriormente de haberla instalado en el proyecto.

```
PS C:\Users\Cesar\Documents\Programas\2024\OLC1_Proyecto2_202202906> npm run jison

> olc1_proyecto2_202202906@1.0.0 jison
> jison Gramatica\gramatica.jison -o Gramatica\gramatica.js
```

Para permitir el manejo de tokens del analizador sintáctico, se opto por el uso de objetos, es decir, durante el análisis para cada instrucción se crea un objeto que ira subiendo en el árbol sintáctico que genera jison, hasta terminar el análisis y como resultado del análisis se obtendrá una lista de instrucciones que está conformado por objetos que permiten ser analizados y ejecutados por sus atributos.

```
function nuevaOpUnit(valor,tipoOperacion,linea,columna,comodin=false){
    let obj = {
        valor: valor,
        tipoOperacion: tipoOperacion,
        comodin: comodin,
        linea: linea,
        columna: columna
    }
    return obj;
}

function nuevaOpBinaria(valor1, valor2, tipoOperacion, linea, columna) {
    let obj = {
        valor1: valor1,
        valor2: valor2,
        tipoOperacion: tipoOperacion,
        linea: linea,
        columna: columna
    }
    return obj;
}

function nuevaOpTernaria(condicion, expresion1, expresion2, tipoOperacion, linea, columna) {
    let obj = {
        condicion: condicion,
        expresion1: expresion1,
        expresion2: expresion2,
        tipoOperacion: tipoOperacion,
        linea: linea,
        columna: columna
    }
    return obj;
}
```

Estos objetos contienen los atributos necesarios para llevar a cabo el tipo de operación que corresponda. Por cada elemento que exista en el lenguaje se creara un objeto para poder analizarlo.

La manera de crear un objeto en cada producción o ir subiendo el objeto:

```
expresion : expresion SUMA expresion        { $$ = nuevaOpBinaria($1, $3, 'SUMA', this._$.first_line, this._$.first_column+1) }
          | expresion RES expresion         { $$ = nuevaOpBinaria($1, $3, 'RESTA', this._$.first_line, this._$.first_column+1) }
          | expresion MULT expresion        { $$ = nuevaOpBinaria($1, $3, 'MULT', this._$.first_line, this._$.first_column+1) }
          | expresion DIV expresion         { $$ = nuevaOpBinaria($1, $3, 'DIV', this._$.first_line, this._$.first_column+1) }
          | expresion MOD expresion         { $$ = nuevaOpBinaria($1, $3, 'MOD', this._$.first_line, this._$.first_column+1) }
          | RES expresion %prec umenos              { $$ = nuevaOpBinaria($1, null, 'NEGATIVO', this._$.first_line, this._$.first_column+1)}
          | POW OPENPAREN expresion COMA expresion CLOSEPAREN     { $$ = nuevaOpBinaria($1, $3, 'POW', this._$.first_line, this._$.first_column+1) }
          | OPENPAREN tipo CLOSEPAREN expresion  { $$ = nuevaOpUnit($4, 'CASTEO', this._$.first_line, this._$.first_column+1, $2) }
          | actualizacion     { $$ = $1; }
          | native_function   { $$ = $1; }
          | op_relacional     { $$ = $1; }
          | op_logicos        { $$ = $1; }
          | llamada           { $$ = $1; }
          | valor             { $$ = $1; }
;

lista_expresion : lista_expresion COMA expresion  { $1.push($3); $$ = $1; }
          | expresion     { $$ = [$1]; }
;

op_relacional
          : expresion ORIGUAL expresion       { $$ = nuevaOpBinaria($1, $3, 'IGUALACION', this._$.first_line, this._$.first_column+1) }
          | expresion ORDIF expresion         { $$ = nuevaOpBinaria($1, $3, 'DIF', this._$.first_line, this._$.first_column+1) }
          | expresion ORMENOR expresion       { $$ = nuevaOpBinaria($1, $3, 'MENORQUE', this._$.first_line, this._$.first_column+1) }
          | expresion ORMENORIGUAL expresion  { $$ = nuevaOpBinaria($1, $3, 'MENORIGUALQUE', this._$.first_line, this._$.first_column+1) }
          | expresion ORMAYOR expresion       { $$ = nuevaOpBinaria($1, $3, 'MAYORQUE', this._$.first_line, this._$.first_column+1) }
          | expresion ORMAYORIGUAL expresion  { $$ = nuevaOpBinaria($1, $3, 'MAYORIGUALQUE', this._$.first_line, this._$.first_column+1) }
          | expresion INCOGNITA expresion PUNTOS expresion  { $$ = nuevaOpTernaria($1, $3, $5, 'IFSHORT', this._$.first_line, this._$.first_column+1)
;
```

El análisis sintáctico por regla general dará como resultado como un árbol sintáctico, en este proyecto se siguió un concepto similar, con la diferencia que este generara una lista de instrucciones, donde cada instrucción es un objeto, dando como resultado general una lista de objetos.

Clase AST

El constructor de esta clase nos genera un array de texto que representara el que se visualizara en consola y las instrucciones como tal.

Se poseen dos métodos, uno para ejecutar las instrucciones y otro para generar la salida de la consola, ambos a partir de los arreglos generados en el constructor.

En el método analizarInst, recibe una lista de instrucciones a ejecutar, de primera instancia tenemos un método que recorre todo el array de instrucciones **2 veces**, la primera vuelta es únicamente para reconocer todos los métodos, funciones y variables que se tienen, la segunda vuelta es la que realiza la ejecución como tal del lenguaje.

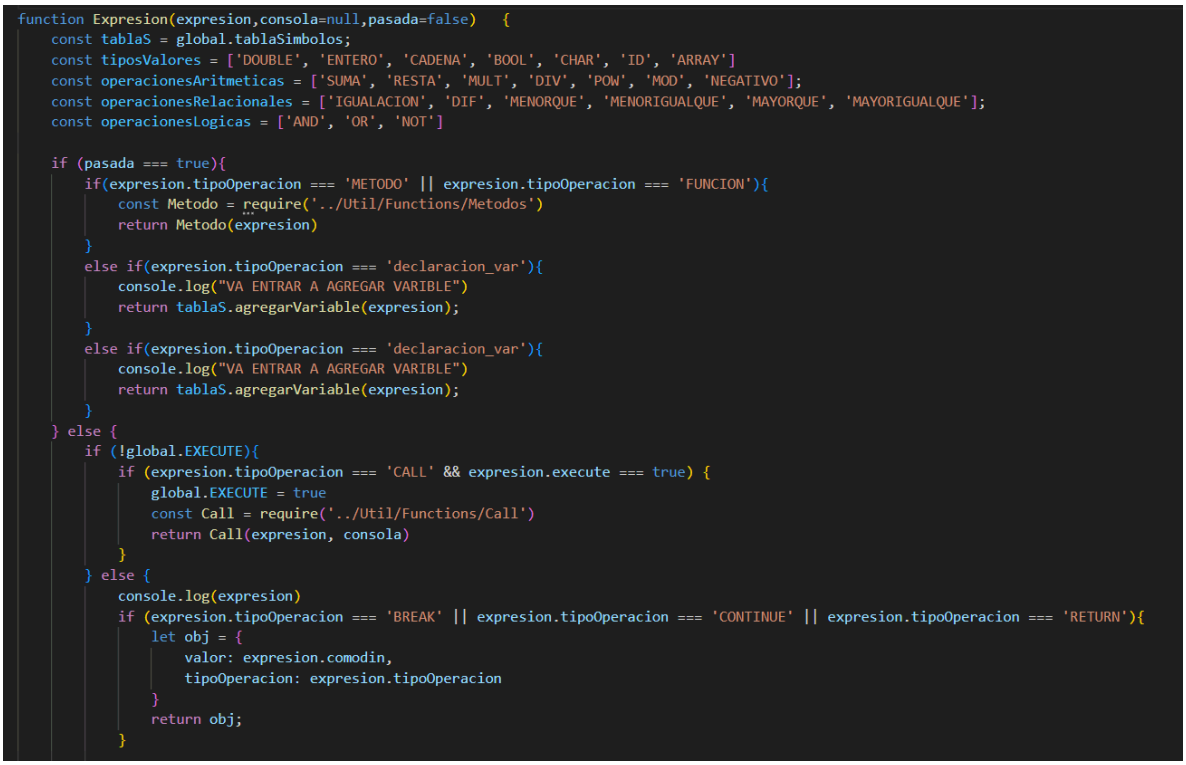
En el método getConsola, se tiene una variable de salida y se utiliza el arreglo de texto generado en el constructor, a partir de toda la ejecución se desglosa aquí por medio de un ciclo, alimentando así la variable de salida con cada iteración del ciclo.

```
class AST {

    constructor(instrucciones){
        this.instrucciones = instrucciones;
        this.consola = [];
    }

    analizarInst() {
        this.instrucciones.forEach(instruccion => {
            Expresion(instruccion,this.consola,true);
        });
        this.instrucciones.forEach(instruccion => {
            Expresion(instruccion,this.consola);
        });
    }

    getConsola(){
        console.log(this.consola);
        let salida = "";
        for (let i = 0; i < this.consola.length; i++){
            salida += this.consola[i].toString();
        }
        console.log(salida)
        return salida
    }

}
```

Función Expresión

Este es el pilar para la ejecución de todas las instrucciones, cada instrucción que en la clase AST se ejecutan por medio de recorrer la lista de instrucciones, se hacen en Expresión, que no exclusivamente es llamada desde la clase AST sino cada que se necesita realizar alguna operación anidada u obtener un valor en específico, ya que jison en el análisis sintáctico guarda los objetos siempre como texto, por lo que se necesita un parseo según el tipo de dato.

La idea principal de esta función es comparar todos los tipos de operación o tipo de valor que recibe cada que es llamada, permitiendo así realizar alguna operación básica como una operación aritmética o lógica hasta algo más complejo como algún ciclo o función y retornar el valor según cada operación, por ejemplo en una operación aritmética retornara un objeto con su valor y tipo resultante, en un método hará simplemente la ejecución, pero en una función retornara el valor resultante de todo el proceso de la función.



```
function Expresion(expresion,consola=null,pasada=false)   {
    const tablaS = global.tablaSimbolos;
    const tiposValores = ['DOUBLE', 'ENTERO', 'CADENA', 'BOOL', 'CHAR', 'ID', 'ARRAY']
    const operacionesAritmeticas = ['SUMA', 'RESTA', 'MULT', 'DIV', 'POW', 'MOD', 'NEGATIVO'];
    const operacionesRelacionales = ['IGUALACION', 'DIF', 'MENORQUE', 'MENORIGUALQUE', 'MAYORQUE', 'MAYORIGUALQUE'];
    const operacionesLogicas = ['AND', 'OR', 'NOT']

    if (pasada === true){
        if(expresion.tipoOperacion === 'METODO' || expresion.tipoOperacion === 'FUNCION'){
            const Metodo = require('../Util/Functions/Metodos')
            return Metodo(expresion)
        }
        else if(expresion.tipoOperacion === 'declaracion_var'){
            console.log("VA ENTRAR A AGREGAR VARIBLE")
            return tablaS.agregarVariable(expresion);
        }
        else if(expresion.tipoOperacion === 'declaracion_var'){
            console.log("VA ENTRAR A AGREGAR VARIBLE")
            return tablaS.agregarVariable(expresion);
        }
    } else {
        if (!global.EXECUTE){
            if (expresion.tipoOperacion === 'CALL' && expresion.execute === true) {
                global.EXECUTE = true
                const Call = require('../Util/Functions/Call')
                return Call(expresion, consola)
            }
        } else {
            console.log(expresion)
            if (expresion.tipoOperacion === 'BREAK' || expresion.tipoOperacion === 'CONTINUE' || expresion.tipoOperacion === 'RETURN'){
                let obj = {
                    valor: expresion.comodin,
                    tipoOperacion: expresion.tipoOperacion
                }
                return obj;
            }
```

```
    else if(expresion.tipoOperacion === 'CALL'){
        const Call = require('../Util/Functions/Call')
        return Call(expresion, consola)
    }
    else if(tiposValores.includes(expresion.tipoValor)) {
        const ValorExpresion = require("../Modelo/Valor");
        return ValorExpresion(expresion);
    }
    else if(expresion.tipoOperacion === 'CASTEO'){
        const casteo = require('../Util/NativeFunctions/Casteo');
        return casteo(expresion);
    }
    else if(expresion.tipoOperacion === 'declaracion_var'){
        console.log("VA ENTRAR A AGREGAR VARIBLE")
        return tablaS.agregarVariable(expresion);
    }
    else if(expresion.tipoOperacion === 'ASIGNACION'){
        let valor = Expresion(expresion.valor)
        console.log("VALOR DE ASIGNACIONES", valor)
        tablaS.asignarValor(expresion.id,valor,0,0)
    }
    else if(expresion.tipoOperacion === 'declaracion_array'){
        return tablaS.agregarArray(expresion);
    }
    else if(operacionesAritmeticas.includes(expresion.tipoOperacion)) {
        const Aritmetica = require("../Util/Aritmetica");
        return Aritmetica(expresion);
    }
    else if(operacionesRelacionales.includes(expresion.tipoOperacion)){
        const OpRelacional = require('../Util/Comparaciones/Relacionales')
        return OpRelacional(expresion)
    }
```

```
        tablaS.increasedecreaseValor(expresion.valor,expresion.tipoOperacion,expresion
    }
    else if(expresion.tipoOperacion === 'TOLOWER'){
        const tolower = require('../Util/NativeFunctions/Lower')
        return tolower(expresion)
    }
    else if(expresion.tipoOperacion === 'TOUPPER'){
        const toupper = require('../Util/NativeFunctions/Upper')
        return toupper(expresion)
    }
    else if(expresion.tipoOperacion === 'ROUND'){
        const round = require('../Util/NativeFunctions/Round')
        return round(expresion)
    }
    else if(expresion.tipoOperacion === 'LENGTH'){
        const Length = require('../Util/NativeFunctions/Length')
        return Length(expresion)
    }
    else if(expresion.tipoOperacion === 'TYPEOF'){
        const TypeOf = require('../Util/NativeFunctions/TypeOf')
        return TypeOf(expresion)
    }
    else if(expresion.tipoOperacion === 'TOSTRING'){
        const ToString = require('../Util/NativeFunctions/ToString')
        return ToString(expresion)
    }
    else if(expresion.tipoOperacion === 'CSTR'){
        console.log("ENTRO ACAJFLKSEJF")
        const c_str = require('../Util/NativeFunctions/C_str')
        return c_str(expresion)
    }
    else if(expresion.tipoOperacion === 'IFSHORT'){
        const OpTernario = require('../Util/Comparaciones/Ternario')
        return OpTernario(expresion);
    }
    else if(operacionesLogicas.includes(expresion.tipoOperacion)){
        const OpLogic = require('../Util/Comparaciones/Logicos')
        return OpLogic(expresion);
    }
```

Express y Cors

Para la conexión de la interfaz grafica (Frontend) y la funcionalidad del programa (Backend) se opto por utilizar el sistema nativo de node js apoyado de express y cors.

Express

Express es un marco de trabajo para Node.js que facilita la creación de aplicaciones web y API. Es minimalista, flexible y proporciona un conjunto robusto de características para desarrollar tanto aplicaciones web como móviles. Express te ayuda a manejar las solicitudes HTTP, configurar rutas, y más, de una manera muy eficiente y sencilla.

CORS (Cross-Origin Resource Sharing)

CORS es un mecanismo de seguridad que permite o restringe los recursos solicitados en un servidor web dependiendo de dónde se originó la solicitud. En términos simples, es una política que define si una solicitud de origen cruzado (es decir, una solicitud hecha desde un dominio diferente al del servidor) puede acceder a los recursos o no. Por defecto, los navegadores restringen las solicitudes de origen cruzado por motivos de seguridad.

Relación entre Express y CORS en la conexión Frontend-Backend

Cuando desarrollas una aplicación con un frontend (por ejemplo, una página web en React o Vue.js) y un backend (por ejemplo, un servidor en Express), normalmente estos componentes están alojados en diferentes dominios o puertos durante el desarrollo. Por ejemplo, el frontend podría estar corriendo en localhost:3000 y el backend en localhost:5000.

Aquí es donde CORS entra en juego. Si el backend no está configurado para aceptar solicitudes del dominio del frontend, el navegador bloqueará estas solicitudes debido a la política de seguridad del mismo origen. Esto puede resultar en errores que impiden que tu aplicación funcione correctamente.

Para resolver esto en Express, puedes usar un middleware llamado cors. Al aplicar este middleware, puedes configurar tu servidor Express para aceptar solicitudes de uno o más orígenes específicos, o permitir todas las solicitudes de origen cruzado de forma segura.

Ejecutar.js > ...

```
const express = require('express')
const path = require('path')
const app = express();
let cors = require('cors');
const parser = require("./Gramatica/gramatica.js");
const Procesador = require("./Gramatica/AST.js");
const TablaSimbolos = require('./Util/TablaSimbolos.js');
global.tablaSimbolos = new TablaSimbolos()
const Reportes = require('./Util/Reportes.js');
global.reportes = new Reportes();

app.use(cors());
app.use(express.json());
app.use(express.static(path.join(__dirname, 'public')));

// Archivo de la pagina como tal
app.get("/", (req, res) => {
    res.sendFile(path.join(__dirname, 'public', 'menu.html'));
});

// Post recibir el texto de la consola y analizarlo con el parser
app.post('/compile', (req, res) => {
    let tablaSimbolos = global.tablaSimbolos
    global.reportes.clear()
    tablaSimbolos.clear();
    var input = req.body.input;
    var result = parser.parse(input);
    console.log("Instrucciones: \n", result.instrucciones)
    var ast = new Procesador(result.instrucciones);
    ast.analizarInst();
    console.log(tablaSimbolos)
    global.reportes.generarTablaErrores()
    global.reportes.generarTablaSimbolos()
    global.reportes.generarReporteAST(result.instrucciones)
    res.send({ output: ast.getConsola() });
    });

// Se ejecutara cuando el servidor este activo
```

```
document.getElementById('crearArchivo').addEventListener('click', function() {
    document.getElementById('consola1').innerText = "";
});

document.getElementById("botonEjecutar").addEventListener("click", function() {
    const contenidoConsola = document.getElementById('consola1').innerText;
    fetch('http://localhost:3080/compile', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ input: contenidoConsola })
    })
    .then(response => response.json())
    .then(data => {
        // Respuesta del servidor
        document.getElementById('consola2').innerText = data.output;  // Mostrar la salida en la consola2
    })
    .catch(error => {
        console.error('Error:', error);
        document.getElementById('consola2').innerText = 'Error al procesar la solicitud.';
    });
});
```

```
// Funcion para abrir el buscador y poder seleccionar algun archivo .sc
document.getElementById('fileInput').addEventListener('change', function(e) {
    const file = e.target.files[0];
    if (file) {
        const reader = new FileReader();
        reader.onload = function(e) {
            const content = e.target.result;
            // Coloca el contenido en la consola de entrada
            document.getElementById('consola1').innerText = content;
        };
        reader.readAsText(file);
    }
});
```