Universidad de San Carlos de Guatemala

Laboratorio de Organización de Lenguajes y Compiladores 1

Sección C

Escuela de Ciencias y Sistemas

José Leonel López Ajvix

Carné: 202201211

Manual Técnico Compiscript +

Compiscript+ es un lenguaje de programación creado para ayudar a los estudiantes de IPC1 a familiarizarse con la programación. Para la creación de Compiscript+ fue usado lo siguiente:

- **Cliente:** Para el cliente fue usado React, así el usuario puede visualizar la aplicación por medio de una página web.
- **Servidor:** En la parte de servidor fue usado NodeJS, con Typescript, para poder usar paradigmas orientados a objetos y restricciones de lenguaje y tipado.
- **Generación Árbol AST:** Se usó la herramienta Graphviz, en el frontend para poder generar el grafo del árbol AST.
- **Framework de diseño:** Se usó el framework de diseño Bootstrap para poder crear una interfaz amigable para el usuario.
- **Creación de análisis léxico y sintáctico:** Fue usado el Framework Jison, para poder crear el analizador léxico y sintáctico.

Gramática y análisis léxico:

Como anteriormente fue mencionado, fue usado el framework Jison para poder generar el analizador léxico y sintáctico:

```
%lex

%options case-insensitive
%x string

%%

\s+                                 // Ignorar espacios en blanco
\/\/.* {}                             // Comentario de una linea
[/][*][^*]*[*]+([^/*][^*]*[*]+)*[/] {} // Comentario Multilinea


// TIPOS DE DATOS
//-----------------------------------------------------
"int"                               return 'R_INT';
"double"                             return 'R_DOUBLE';
"char"                              return 'R_CHAR';
"string"                            return 'R_STRING';
"bool"                              return 'R_BOOL';
//-----------------------------------------------------
```

```
%start inicio

%%

inicio : instrucciones EOF                  {return $1;}
;

instrucciones : instrucciones instruccion   {$1.push($2); $$=$1;}
              | instruccion                 {$$=[$1];}
;

instruccion : impresion             {$$=$1;}
            | declaracion R_PUNTOYCOMA          {$$=$1;}
            | asignacion  R_PUNTOYCOMA          {$$=$1;}
            | if                                {$$=$1;}
            | while                             {$$=$1;}
            | break                             {$$=$1;}
            | do_while                          {$$=$1;}
            | for                               {$$=$1;}
```

Aplicación cliente servidor:

Se usó una aplicación cliente servidor para poder levantar la aplicación. Aquí podremos ver en el Index como se levanta este servidor:

```
class servidor {
    public app: Application;

    constructor() {
        this.app = express();
        this.config();
        this.routes();
    }

    config(): any {
        this.app.set('port', process.env.PORT || 4000);
        this.app.use(morgan('dev'));
        this.app.use(express.urlencoded({ extended: false }));
        this.app.use(express.json());
        this.app.use(express.json({ limit: '50mb' }));
        this.app.use(express.urlencoded({ limit: '50mb' }));
        this.app.use(cors());
        this.app.use(bodyParser.urlencoded({ extended: true }));
    }

    routes(): void {
        this.app.use('/', indexRouter);
    }

    start(): void {
        this.app.listen(this.app.get('port'), () => {
            console.log('Servidor en puerto', this.app.get('port'));
        }
```

```
class controller{

    public interpretar(req: Request, res: Response){

        lista_errores = new Array<Errores>

        try{

            dotAst = "";

            let parser = require('../../gramatica/gramatica')
            let ast = new Arbol(parser.parse(req.body.message));
            let tabla = new tablaSimbolo();
            tabla.setNombre("Global");
            ast.setTablaGlobal(tabla);
            ast.setConsola("");

            for(let err of lista_errores){
                ast.Print("Error "+err.getTipoError()+ ". "+err.getDesc()+" linea: "+err.getFila()+" columna: "+(err.getCol
            }

            let recorrido1 = null;
            for(let i of ast.getInstrucciones()){

                if(i instanceof Metodo){
                    //console.log("guardo un metodo")
                    i.id = i.id.toLocaleLowerCase();
                    ast.agregarFunciones(i);
```

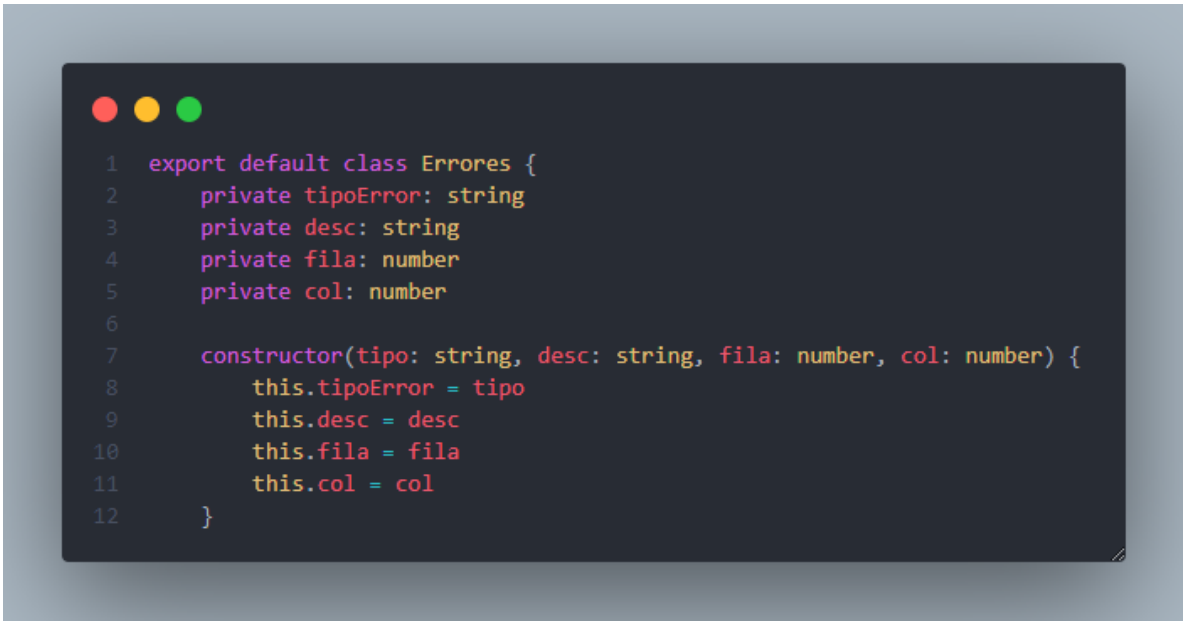
Uso de clases abstractas:

Para poder hacer un mejor uso de la aplicación, se hizo una clase abstracta llamada Instrucción, la cual, todas las siguientes clases heredarán de ella:

```
import Arbol from "../simbol/arbol";
import tablaSimbolos from "../simbol/tablaSimbolos";
import Tipo from "../simbol/tipo";

export abstract class Instruccion {
    public tipoDato: Tipo
    public linea: number
    public columna: number

    constructor(tipo: Tipo, linea: number, columna: number) {
        this.tipoDato = tipo
        this.linea = linea
        this.columna = columna
    }

    abstract interpretar(arbol: Arbol, tabla: tablaSimbolos): any
    abstract obtenerAST(anterior: string): string;

}
```

Manejo de errores:

La clase errores tiene la siguiente lógica:



```
export default class Errores {
    private tipoError: string
    private desc: string
    private fila: number
    private col: number

    constructor(tipo: string, desc: string, fila: number, col: number) {
        this.tipoError = tipo
        this.desc = desc
        this.fila = fila
        this.col = col
    }
```

Árbol:

La clase árbol es usada para poder ver las funciones. Es la encargada de poder ejecutar todo, desde las instrucciones, hasta la tabla de símbolos.

```
export default class Arbol {
    private instrucciones: Array<Instruccion>
    private consola: string
    private tablaGlobal: tablaSimbolo
    private funciones: Array<Instruccion>

    constructor(instrucciones: Array<Instruccion>) {
        this.instrucciones = instrucciones
        this.consola = ""
        this.tablaGlobal = new tablaSimbolo()
        this.funciones = new Array<Instruccion>
    }
```

Símbolo:

La clase símbolo será usada para poder guardar todos los símbolos:

```
export default class Simbolo {
    private tipo: Tipo
    private id: string
    private valor: any

    constructor(tipo: Tipo, id: string, valor?: any) {
        this.tipo = tipo
        this.id = id.toLocaleLowerCase()
        this.valor = valor
    }
```

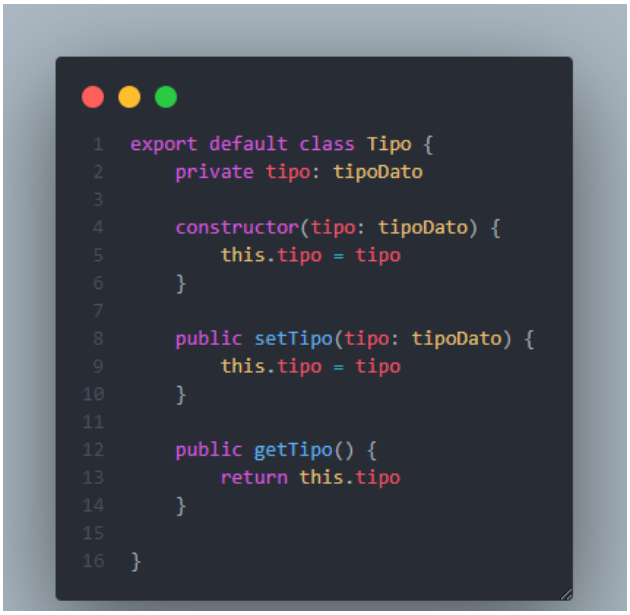
Tabla de símbolos:

En esta clase se guardarán todos los símbolos, y en este también mostrará las tablas anteriores, para poder ver variables en ámbitos anteriores. Tiene la siguiente estructura:

```
export default class tablaSimbolo {
    private tablaAnterior: tablaSimbolo | any
    private tablaActual: Map<string, Simbolo>
    private nombre: string

    constructor(anterior?: tablaSimbolo) {
        this.tablaAnterior = anterior
        this.tablaActual = new Map<string, Simbolo>()
        this.nombre = ""
    }

```

Tipo:

La clase Tipo, generará los diferentes tipos de datos en el lenguaje de programación, los cuales son, Entero, Decimal, Bool, Carácter, Cadena, Void



```
export default class Tipo {
    private tipo: tipoDato

    constructor(tipo: tipoDato) {
        this.tipo = tipo
    }

    public setTipo(tipo: tipoDato) {
        this.tipo = tipo
    }

    public getTipo() {
        return this.tipo
    }

}
```

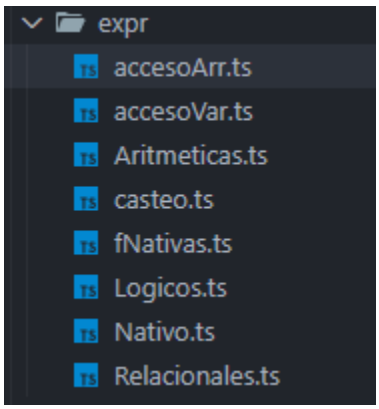
Uso de Patrón Singleton:

Se usó el patrón Singleton para poder generar IDS únicas al momento de generar el grafo.

```typescript
export default class ContadorSingleton {
    private static instance: ContadorSingleton;
    private contador: number;

    private constructor() {
        this.contador = 0;
    }

    public static getInstance(): ContadorSingleton {
        if (!ContadorSingleton.instance) {
            ContadorSingleton.instance = new ContadorSingleton();
        }

        return ContadorSingleton.instance;
    }

    public getContador(): number {
        this.contador++;
        return this.contador;
    }

}
```

Expresiones:

A grandes rasgos, esta carpeta contiene todas las expresiones que pueden ser creadas por el usuario. Aquí se contienen las expresiones aritméticas, lógicas y relacionales. El acceso a variables, los casteos y funciones nativas.



Instrucciones:

En esta carpeta están todas las instrucciones, tales como las asignaciones a variables, los ciclos, el if, switch, llamadas, ternario imprimir en pantalla, return, break, entre otros.

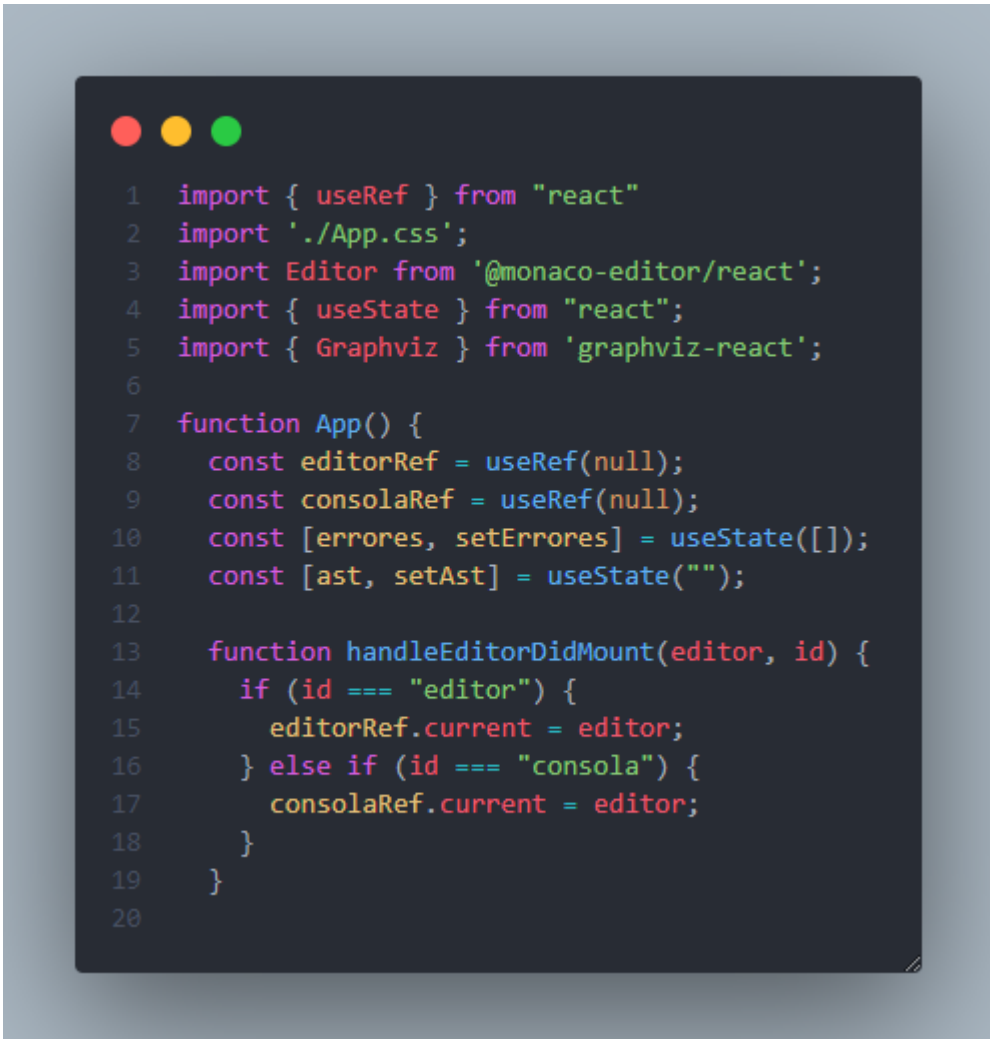
Cabe aclarar que, las instrucciones y las expresiones heredan de la clase instrucción. Esto hace que se deba implementar sus métodos abstractos. Un pequeño ejemplo de como una clase hereda de Instrucción:

```
export default class Nativo extends Instruccion {
    valor: any

    constructor(tipo: Tipo, valor: any, fila: number, columna: number) {
        super(tipo, fila, columna)
        this.valor = valor
    }

    interpretar(arbol: Arbol, tabla: tablaSimbolo) {
        return this.valor
    }

    obtenerAST(anterior: string): string {
        let contador = ContadorSingleton.getInstance();
        let nodoN = `n${contador.getContador()}`
        let nodoV = `n${contador.getContador()}`
        let result = `${nodoN}[label="Dato Nativo"];\n`
        result += `${nodoV}[label="${this.valor}"];\n`
        result += `${nodoN}->${nodoV};\n`
        result += `${anterior}->${nodoN};\n`
        return result
    }
}
```

Como logramos ver en la clase Nativo, esta tiene un constructor, el método abstracto interpretar, que interpretará la instrucción a nuestra conveniencia. Y el obtenerAST que será el responsable de generar el AST.

Cliente:

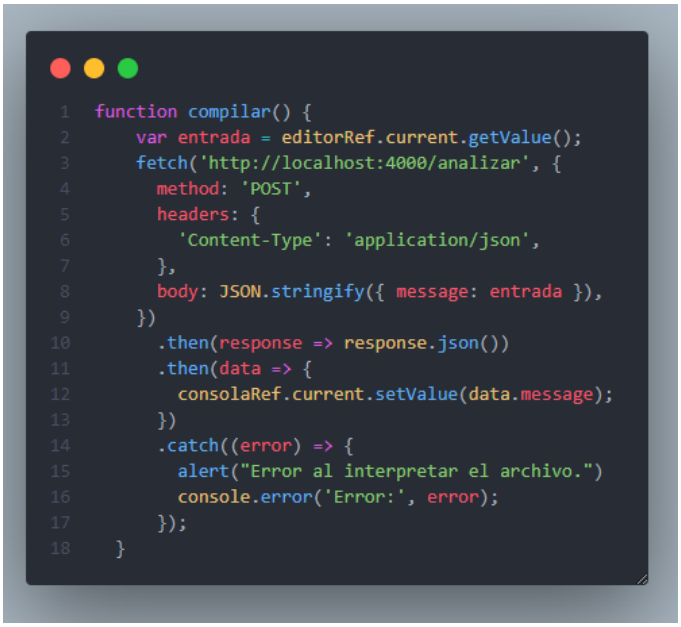
Como fue mencionado anteriormente, se usó React para la creación del cliente. Aquí se implementó una librería para poder usar graphviz, y se llaman los endpoints creados en el servidor.



```
import { useRef } from "react"
import './App.css';
import Editor from '@monaco-editor/react';
import { useState } from "react";
import { Graphviz } from 'graphviz-react';

function App() {
  const editorRef = useRef(null);
  const consolaRef = useRef(null);
  const [errores, setErrores] = useState([]);
  const [ast, setAst] = useState("");

  function handleEditorDidMount(editor, id) {
    if (id === "editor") {
      editorRef.current = editor;
    } else if (id === "consola") {
      consolaRef.current = editor;
    }
  }

```

Compilar:

Para poder llamar el endpoint de compilar, tenemos la siguiente función:

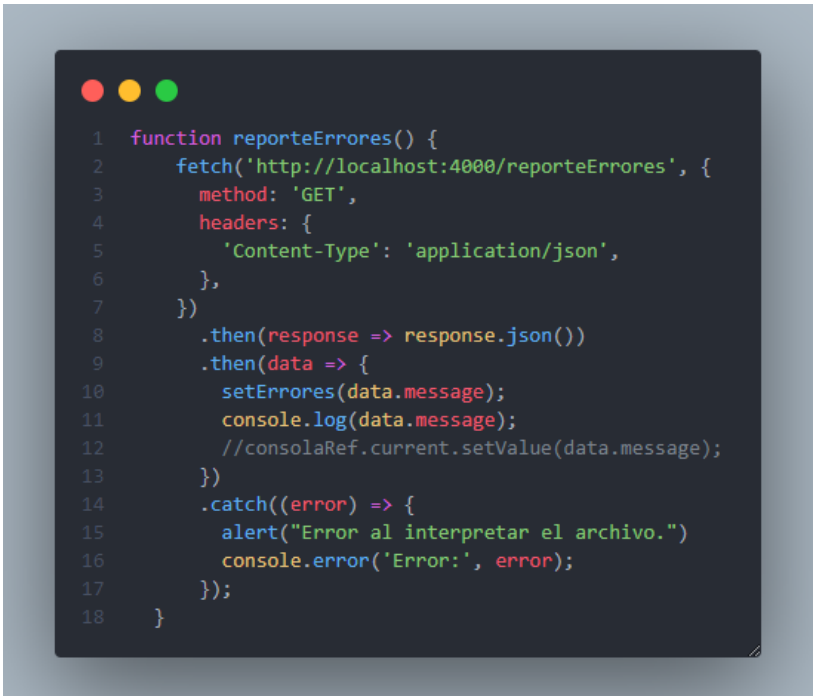


```
function compilar() {
    var entrada = editorRef.current.getValue();
    fetch('http://localhost:4000/analizar', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ message: entrada }),
    })
      .then(response => response.json())
      .then(data => {
        consolaRef.current.setValue(data.message);
      })
      .catch((error) => {
        alert("Error al interpretar el archivo.")
        console.error('Error:', error);
      });
  }
```

Esto interpretará el archivo y mostrará en consola lo que se desea.

Reporte de errores:

Para poder llamar el endpoint del reporte de errores es el siguiente.



```
function reporteErrores() {
    fetch('http://localhost:4000/reporteErrores', {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json',
      },
    })
      .then(response => response.json())
      .then(data => {
        setErrores(data.message);
        console.log(data.message);
        //consolaRef.current.setValue(data.message);
      })
      .catch((error) => {
        alert("Error al interpretar el archivo.")
        console.error('Error:', error);
      });
  }
```

Reporte AST:

También tendremos esta función que ejecute el endpoint de generar el reporte de AST.

```
function reporteAST(){
    fetch('http://localhost:4000/reporteAST', {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json',
      },
    })
      .then(response => response.json())
      .then(data => {
        setAst(data.message);
        console.log(data.message);
        //consolaRef.current.setValue(data.message);
      })
      .catch((error) => {
        alert("Error al interpretar el archivo.")
        console.error('Error:', error);
      });
  }
```

Package json:

Tenemos los siguientes imports en el package json, como vemos, tenemos Graphviz instalado.

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@monaco-editor/react": "^4.6.0",
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "graphviz-react": "^1.2.5",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
```