Universidad de San Carlos de Guatemala

Escuela de Ciencias y Sistemas

Laboratorio Organización de Lenguajes y Compiladores 2

José Leonel López Ajvix

Carné: 202201211

Manual técnico:

Objetivo:

El objetivo del Manual técnico es poder explicar de una manera entendible el código usado para el intérprete Oakland. El lenguaje de programación fue Javascript Vanilla, no se usó ningún tipo de servidor como NodeJS, y fue subido en Github Pages. Para el analizador fue usado PeggyJS.

Index.html:

En el index.html fue usado Bootstrap en CDN para poder manejar los estilos y los Modals más fácilmente. Aquí un ejemplo del encabezado en el HTML.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Oakland</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
    <style>
        body {
            background-color: #f0f0f0;
        }
        textarea {
            width: 100%;
            border: 1px solid #ced4da;
            padding: 10px;
            font-family: monospace;
            height: auto;
        }
    </style>
</head>
```

Gramática:

Para la gramática fue usado PeggyJS, para eso se conectó con la clase Nodos, para crear diferentes nodos y así manejar el intérprete.

```
programa = _ dcl:Declaracion* _ { return dcl }

Declaracion = decl: DeclStruct _ { return decl }
            /  dcl:VarDcl _ { return dcl }
            / dcl:DeclInstancia _ { return dcl }
            / stmt:Stmt _ { return stmt }
            / dcl:FuncDcl _ { return dcl }
            / arreglo:Arreglo _ { return arreglo }

VarDcl = tipo:TipoDato _ id:Identificador _ exp:("=" _ exp:Expresion _ {return exp})?";" { return crearNodo('tipoVariable', { tipo, id, exp }) }
       / "var" _ id:Identificador _ "=" _ exp:Expresion ";" { return crearNodo('declaracionVariable', { id, exp }) }
TipoDato = "int" / "float" / "string" / "boolean" / "char"
```

Index.js:

En el index.js fue llamada toda la lógica para compilar el proyecto, además para llenar los errores y la tabla de símbolos. Aquí se muestran los métodos más importantes, como para compilar el proyecto

```
openFileInput.addEventListener('change', (event) => {
    const files = event.target.files;
    if (files.length > 0) {
        const file = files[0]; // Tomar el primer archivo seleccionado
        const reader = new FileReader();

        reader.onload = function(e) {
            editor.value = e.target.result; // Cargar el contenido del archivo en el textarea
        };

        reader.readAsText(file); // Leer el archivo como texto
    }
});

const reportButton = document.getElementById('reportButton');
reportButton.addEventListener('click', () => {
    actualizarTablaErrores();
});

const variablesButton = document.getElementById('variablesButton');
variablesButton.addEventListener('click', () => {
    actualizarTablaSimbolos();
});


btn.addEventListener('click', () => {

    const codigoFuente = editor.value;
    errores = [];
    Entorno.listaVariables.length = 0;

    try{


        const sentencias = parse(codigoFuente);

        const interprete = new InterpreterVisitor();

        console.log({ sentencias });
        sentencias.forEach(sentencia => sentencia.accept(interprete));

        console.log("Salida:", interprete.salida);
        salida.textContent = interprete.salida;
    }catch(e){

        if(e instanceof SemanticError){
            addError(e.tipo, e.linea, e.columna, e.message);
            salida.textContent = `Error Semántico: ${e.message} en linea: ${e.linea} columna: ${e.columna}`;
        }else{
            const linea = e.location ? e.location.start.line: 'Desconocida';
            const columna = e.location ? e.location.start.column: 'Desconocida';
            salida.textContent = `Error léxico/sintáctico: ${e.message} en linea: ${linea} columna: ${columna}`;
            addError('Léxico/Sintáctico:', linea, columna, e.message);
            console.log(e);
        }

    }

});
```

Nodos.js:

La clase nodos.js fue usada como modelo para los diferentes nodos creados en PeggyJS, un ejemplo de los nodos en la operación Binaria es este:

```
export class OperacionUnaria extends Expresion {

    /**
     * @param {Object} options
     * @param {Expresion} options.exp Expresion de la operacion
  * @param {string} options.op Operador de la operacion
     */
    constructor({ exp, op }) {
        super();

        /**
         * Expresion de la operacion
         * @type {Expresion}
         */
        this.exp = exp;


        /**
         * Operador de la operacion
         * @type {string}
         */
        this.op = op;

    }
```

Visitor:

El Visitor será el responsable de manejar el patrón Visitor, el cual fue usado para el proyecto. Aquí un ejemplo del patrón visitor en operación Unaria:

```
/**
     * @param {OperacionUnaria} node
     * @returns {any}
     */
    visitOperacionUnaria(node) {
        throw new Error('Metodo visitOperacionUnaria no implementado');
    }
```

Interprete.js:

El interprete.js hereda de la clase Visitor, esto hará que deba heredar todos sus métodos, y este hará la lógica que manejará el interprete. Aquí un ejemplo de la declaración de variables:

```
/**
 * @type {BaseVisitor['visitDeclaracionVariable']}
 */
visitDeclaracionVariable(node) {
    const nombreVariable = node.id;

    if(InterpreterVisitor.palabrasReservadas.includes(nombreVariable)){
        let err = new SemanticError(node.location.start.line,node.location.start.column,`No se puede declarar una variable con el nombre de una palabra reservada`);
        errores.push(err);
    }else{
        if (!node.exp) {
            let err = new SemanticError(node.location.start.line,node.location.start.column,`La variable ${nombreVariable} no tiene un valor asignado`);
            errores.push(err);
        }

        // Evaluar la expresión para obtener el valor y el tipo
        const valorVariable = node.exp.accept(this);

        // Asignar la variable con el tipo y valor deducidos
        if(valorVariable.tipo == "null"){
            let err = new SemanticError(node.location.start.line,node.location.start.column,`No se le pudo asignar un tipo a la variable ${nombreVariable}`);
            errores.push(err);
        }else{
            this.entornoActual.setVariable(valorVariable.tipo, nombreVariable, valorVariable.valor,node.location.start.line,node.location.start.column);
        }
    }

}
```

Entorno.js:

La clase entorno será usada para manejar las variables y los entornos que se manejarán las variables, esto es debido a la propia naturaleza del lenguaje Oakland y la declaración de variables. Aquí se muestra un ejemplo de su constructor y de como se guarda una variable:

```
export class Entorno {

    static listaVariables = [];

    constructor(padre = undefined) {
        this.valores = {}
        this.padre = padre
    }

    /**
     * @param {string} nombre
     * @param {any} valor
     */
    setVariable(tipo, nombre, valor,linea,columna) {

        if(this.valores[nombre] != undefined) {
            let err = new SemanticError(linea,columna,`Variable ${nombre} ya definida`)
            errores.push(err);
        }

        this.valores[nombre] = {valor,tipo,linea,columna};
        Entorno.listaVariables.push({tipo,nombre,valor,linea,columna});
    }
```

Transfer.js:

Transfer.js tendrá varias clases, todas heredarán de la clase nativa Error en Javascript, esto es debido a que la clase error nos permitirá atraparlo en un try catch, que nos servirá para interrumpir los bucles o acciones en el intérprete, atrapando el break, continue o return y hacer acciones específicas dependiendo del error. También la clase SemanticError extiende de Error, esto se hace solamente por llevar un orden, esto guardará los errores semánticos en una lista, para poder ser mostrados previamente. La clase tiene esta estructura:

```
export class BreakException extends Error {
    constructor() {
        super('Break');
    }
}

export class ContinueException extends Error {
    constructor() {
        super('Continue');
    }
}

export class ReturnException extends Error {
    constructor(value) {
        super('Return');
        this.value = value;
    }
}

export class SemanticError extends Error {
    constructor(linea,columna, message) {
        super(message);
        this.tipo = "Semantico";
        this.linea = linea;
        this.columna = columna;
    }
}
```

Invocar.js:

La clase invocar servirá como una clase padre y modelo para los métodos y llamadas a funciones, además de los Structs manejados y las instancias, esto es debido a que los métodos deben llevar cierta aridad y se debe “invocar” o llamar al método. Es diferente a las variables debido a la naturaleza de los métodos y structs. La clase Invocar tiene lo siguiente:

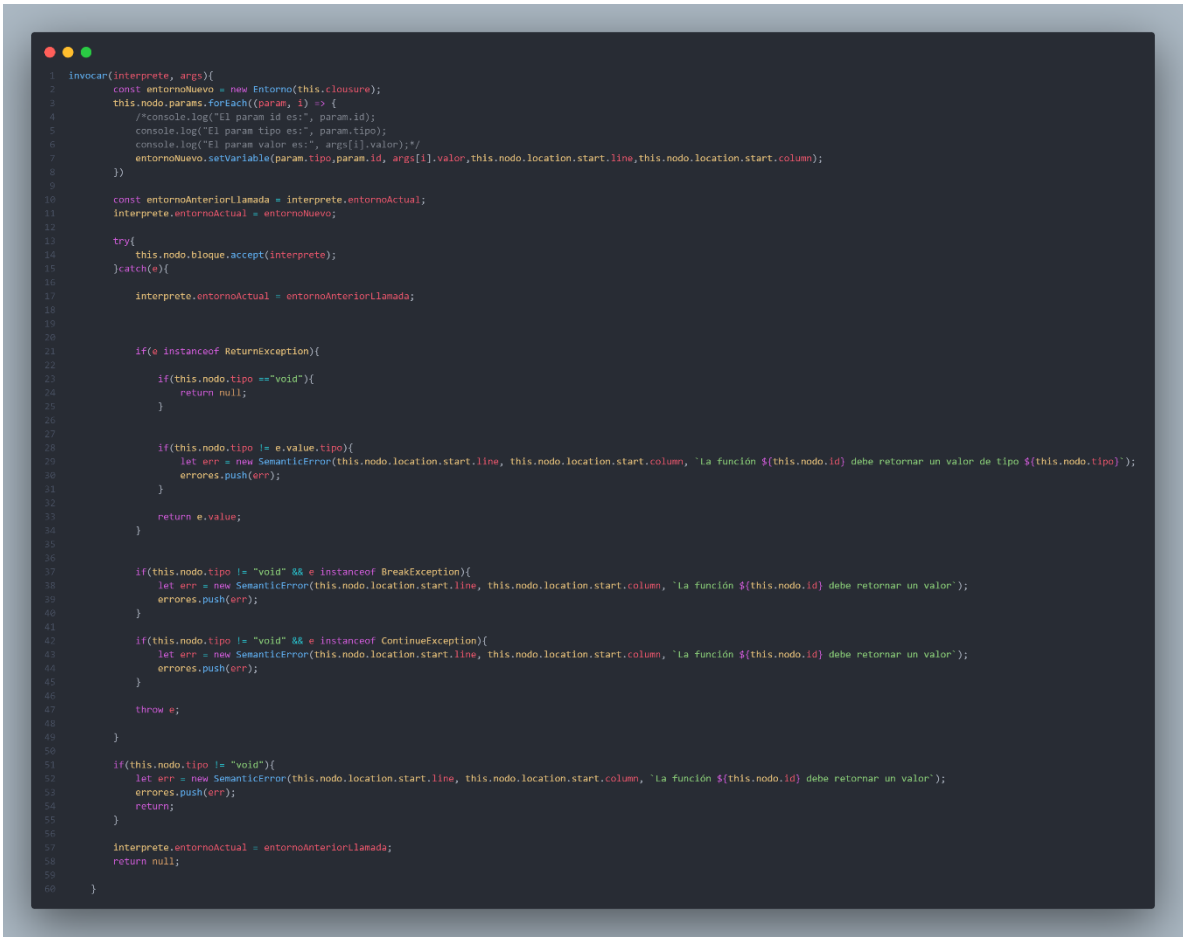
```
import { InterpreterVisitor } from "./interprete.js";

export class Invocar {
    aridad(){
        throw new Error("Objeto aún no implementado");
    }

    /**
     *
     * @param interprete {InterpreterVisitor}
     * @param args {any[]}
     */
    invocar(interprete, args){
        throw new Error("Objeto aún no implementado");
    }
}
```

Foreign.js:

La clase foreign extenderá de Invocar, esta clase es específica para poder manejar las llamadas de las funciones y las funciones. Al heredar de Invocar debe implementar aridad, que verá el número y concordancia del tipo de datos de los parámetros y la función invocar, que ejecutará la función, así mismo verá si lleva un retorno o no dependiendo si es función o método. La función invocar está puesta de la siguiente manera:



```
invocar(interprete, args){
        const entornoNuevo = new Entorno(this.clousure);
        this.nodo.params.forEach((param, i) => {
            /*console.log("El param id es:", param.id);
            console.log("El param tipo es:", param.tipo);
            console.log("El param valor es:", args[i].valor);*/
            entornoNuevo.setVariable(param.tipo,param.id, args[i].valor,this.nodo.location.start.line,this.nodo.location.start.column);
        })

        const entornoAnteriorLlamada = interprete.entornoActual;
        interprete.entornoActual = entornoNuevo;

        try{
            this.nodo.bloque.accept(interprete);
        }catch(e){

            interprete.entornoActual = entornoAnteriorLlamada;



            if(e instanceof ReturnException){

                if(this.nodo.tipo =="void"){
                    return null;
                }


                if(this.nodo.tipo != e.value.tipo){
                    let err = new SemanticError(this.nodo.location.start.line, this.nodo.location.start.column, `La función ${this.nodo.id} debe retornar un valor de tipo ${this.nodo.tipo}`);
                    errores.push(err);
                }

                return e.value;
            }


            if(this.nodo.tipo != "void" && e instanceof BreakException){
                let err = new SemanticError(this.nodo.location.start.line, this.nodo.location.start.column, `La función ${this.nodo.id} debe retornar un valor`);
                errores.push(err);
            }

            if(this.nodo.tipo != "void" && e instanceof ContinueException){
                let err = new SemanticError(this.nodo.location.start.line, this.nodo.location.start.column, `La función ${this.nodo.id} debe retornar un valor`);
                errores.push(err);
            }

            throw e;

        }

        if(this.nodo.tipo != "void"){
            let err = new SemanticError(this.nodo.location.start.line, this.nodo.location.start.column, `La función ${this.nodo.id} debe retornar un valor`);
            errores.push(err);
            return;
        }

        interprete.entornoActual = entornoAnteriorLlamada;
        return null;

    }
```

Structs:

La clase struct, como su nombre lo define, manejará los structs, desde la implementación hasta la instancia de los mismos. Estos structs además de su aridad e invocar llevan métodos get y set, estos métodos servirán para poder obtener y asignar valores a las instancias de los structs. Aquí se muestra la lógica de la clase:

```
import { Invocar } from "./invocar.js";
import { Instancia } from "./instancia.js";

export class Struct extends Invocar {

    constructor(nombre,properties){
        super();
        /**
         * @type {string}
         */
        this.nombre = nombre;

        /**
        * @type {Object.<string, Expresion>}
        */
        this.properties = properties;


    }


    aridad() {
        return Object.keys(this.properties).length;
    }

    /**
    * @type {Invocar['invocar']}
    */
    invocar(interprete, args) {
        const instanciaNueva = new Instancia(this);

        Object.entries(this.properties).forEach(([nombre, valor]) => {
            instanciaNueva.set(nombre, valor);
        });

        return instanciaNueva;

    }
}
```

Conclusión:

Se usó la herramienta PeggyJS para poder hacer el analizador léxico y sintáctico. Para el analizador semántico se usó varias clases, incluyendo el patrón visitor. Esta documentación de código servirá para poder mejorarlo y poder escalarlo en un futuro.