UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

ESCUELA DE SISTEMAS

ORGANIZACIÓN DE LENGUAJES COMPILADORES

ING. MARIO BAUTISTA


TECNICO

LUIS MARIANO MOREIRA GARCÍA

202010770

**Objetivo General**
Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

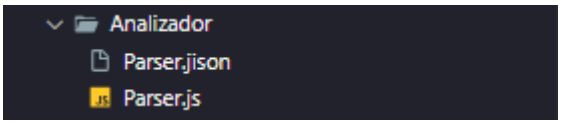
**Objetivos específicos**
● Reforzar los conocimientos de análisis léxico, sintáctico y semántico para la creación de un lenguaje de programación.
● Aplicar los conceptos de compiladores para implementar un proceso de interpretación de código de alto nivel.
● Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.
● Aplicar la teoría de compiladores para la creación de soluciones de software.
● Generar aplicaciones utilizando la arquitectura cliente-servidor.

Requerimientos Mínimos:

- Windows 10
- 1 GB RAM libre
- Tener node js instalado
- Tener Visual Studio Code instalado
- Conocimientos de programación
- Saber manejar JS, html y css

Comprensión de las clases Utilizadas

Analizador:

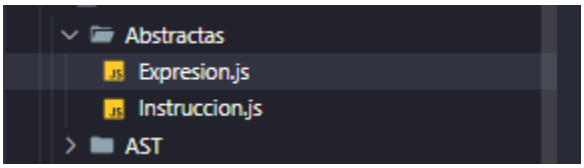



Aquí se encuentra para realizar la gramática, para ejecutar el comando de la gramática se usa:

npm run parser

Y se genera el archivo Parser.js

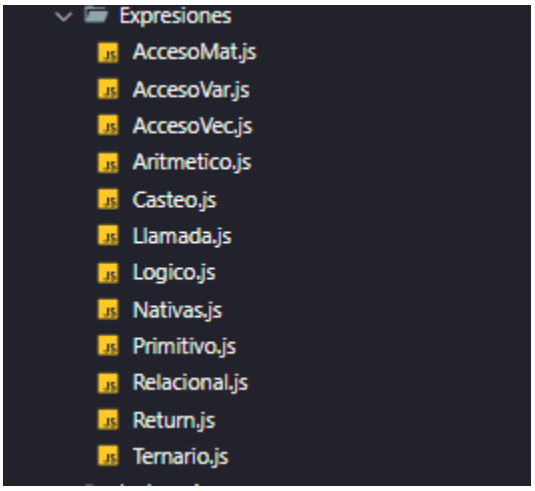
Clases Abstractas:




Expresión.js -> Sirve para manejar las impresiones, o mejor dicho los valores.

```
class Expresion {
    constructor(linea, columna, tipoExp) {
        this.linea = linea
        this.columna = columna
        this.tipoExp = tipoExp
    }

    execute(_) {}

    ast() {}
}

module.exports = { Expresion }
```

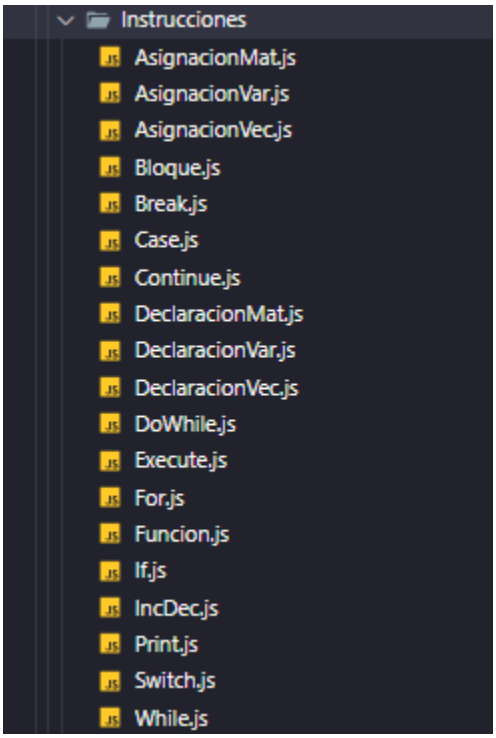
Tenemos la línea y columna en la que se encuentra la expresión y el tipo de expresión, más adelante se cubrirán:




Instrucción.js -> Sirve para manejar las instrucciones a realizar.

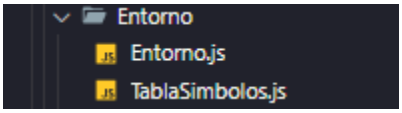
```
class Instruccion {
    constructor(linea, columna, tipoInst) {
        this.linea = linea
        this.columna = columna
        this.tipoInst = tipoInst
    }

    execute(_) {}

    ast() {}
}

module.exports = { Instruccion }
```

Al igual que en expresión aquí tenemos la línea y la columna que se encuentra y el tipo de instrucción, entre los cuales se encuentran:




Antes de cubrir las expresiones e instrucciones, cubriré lo que es el entorno:

Entorno:



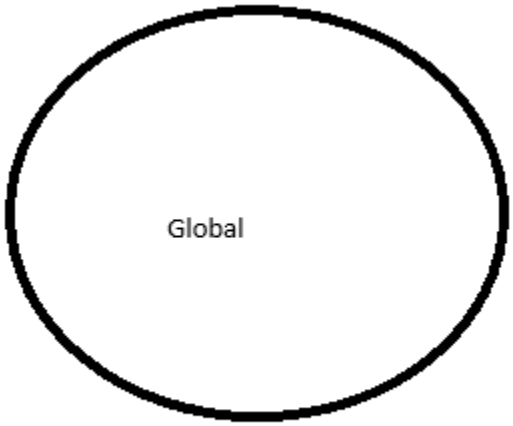

Entorno.js -> Es vital para poder ejecutar correctamente todo los bloques como son debidos

```
class Entorno {
    constructor(anterior, nombre) {
        this.anterior = anterior
        this.nombre = nombre
        this.variables = new Map()
        this.funciones = new Map()
    }
```

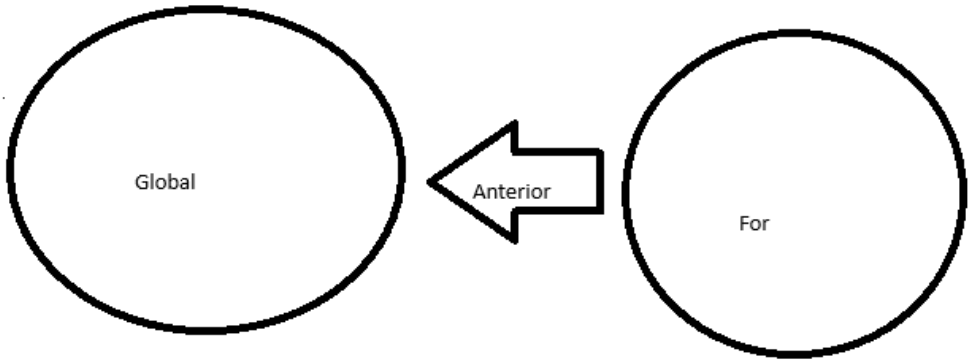
Aquí podremos ejecutar correctamente nuestras instrucciones y Expresiones, tiene una forma similar a una lista enlazada, pero esto nos ayuda con la instrucción break romper el ciclo y volver al entorno anterior.

Para entender mejor correctamente el entorno podemos verlo de la + siguiente forma:

Al iniciar el programa lo tendremos de esta forma:




Este es el núcleo del proyecto y básicamente garantiza que todas las variables o funciones que sean declaradas aquí se puedan trabajar mas adelante, veamos el caso en donde agregamos un for en el entorno global (Omitiremos el Execute para entendimiento práctico)




Como podemos apreciar Global no tiene idea del for, pero el for si conoce a su padre, lo cual nos ayuda a no confundir las variables de los entornos, más adelante se describirá como se obtienen los valores entre entornos.

Funciones:

guardarFuncion:

```
guardarFuncion = (nombre, funcion) => {
    if(!this.funciones.has(nombre.toLowerCase())) {
        this.funciones.set(nombre.toLowerCase(), funcion)
        const tipoFunc = this.obtenerTipoFunc(funcion.tipo)
        tablaSimbolos.push({nombre: nombre.toLowerCase(), nameEnv: this.nombre, tipoID: tipoFunc == 'void' ? 'Método' : 'Función', tipo:
        return
    }
    this.setError(`Redefinición de función existente "${nombre}".`, funcion.linea, funcion.columna)
}
```

Verificamos que no exista una función con el mismo nombre y se Registra dentro de nuestra tabla de símbolos.

obtenerFuncion:

```
obtenerFuncion = (nombre) => {
    let entorno = this
    while(entorno) {
        if(entorno.funciones.has(nombre.toLowerCase())) {
            return entorno.funciones.get(nombre.toLowerCase())
        }
        entorno = entorno.anterior
    }
    return null
}
```

Esta nos ayuda para obtener lo que este adentro de nuestra función.

guardarVariable:

```
guardarVariable = (nombre, valor, tipo, linea, columna) => {
    if(!this.variables.has(nombre.toLowerCase())) {
        this.variables.set(nombre.toLowerCase(), {nombre: nombre.toLowerCase(), valor: valor, tipo: tipo})
        tablaSimbolos.push({nombre: nombre.toLowerCase(), nameEnv: this.nombre, tipoID: 'Variable', tipo: this.obtenerT
        return
    }
    this.setError(`Redeclaración de variable existente "${nombre}".`, linea, columna)
}
```

Nos sirve para almacenar una variable siempre y cuando no exista dentro del entorno actual.

obtenerVariable:

```
obtenerVariable = (nombre) => {
    let entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            return entorno.variables.get(nombre.toLowerCase())
        }
        entorno = entorno.anterior
    }
    return null
}
```

Sirve para obtener el valor de nuestra variable.

reasignarValorVariable:

```
reasignarValorVariable = (nombre, valor, linea, columna) => {
    var entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            let simbolo = entorno.variables.get(nombre.toLowerCase())
            if(simbolo.tipo === valor.tipo || simbolo.tipo === Tipo.DOUBLE && valor.tipo === Tipo.INT) {
                simbolo.valor = valor.valor
                entorno.variables.set(nombre.toLowerCase(), simbolo)
                return true
            }
            this.setError(`Los tipos no coinciden en la asignación. Intenta asignar un "${this.obtenerTipo(valor.tipo)}" a un "$
            return false
        }
        entorno = entorno.anterior
    }
    this.setError(`Resignación de valor a variable inexistente  "${nombre}".`, linea, columna)
    return false
}
```

Sirve para reasignar el valor que esta dentro de nuestra variable, haciendo las validaciones correspondientes.

guardarVector:

```
guardarVector = (nombre, valor, tipo1, tipo2, linea, columna) => {
    if(!this.variables.has(nombre.toLowerCase())) {
        this.variables.set(nombre.toLowerCase(), {nombre: nombre.toLowerCase(), valor: valor, tipo: tipo1})
        tablaSimbolos.push({nombre: nombre.toLowerCase(), nameEnv: this.nombre, tipoID: 'Variable', tipo: `${this.ob
        return
    }
    this.setError(`Redeclaración de variable existente "${nombre}".`, linea, columna)
}
```

Funciona para guardar un vector dentro del entorno.

obtenerPosicionVector:

```
obtenerPosicionVector = (nombre, indiceI, linea, columna) => {
    var entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            var variable = entorno.variables.get(nombre.toLowerCase())
            if(variable.tipo === Tipo.VECTOR || variable.tipo === Tipo.MATRIZ) {
                variable = variable.valor
                if(indiceI < variable.length) {
                    return variable[indiceI]
                }
                this.setError(`Indice fuera de rango. Indice ${indiceI} en longitud ${variable.length}.`, linea, columna)
                return null
            }
            this.setError(`"${this.nombre.toLowerCase()}" no es un vector.`, linea, columna)
            return null
        }
        entorno = entorno.anterior
    }
    this.setError(`Acceso a variable inexistente "${this.nombre}".`, linea, columna)
    return null
}
```

Sirve para obtener un espacio en específico del vector, validando que se respete cada una de las posiciones del vector, se hace mención a vector por aquella “lista” que solo tiene una posición.

obtenerPocisionMatriz:

```
obtenerPosicionMatriz = (nombre, indiceI, indiceJ, linea, columna) => {
    var entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            var variable = entorno.variables.get(nombre.toLowerCase())
            if(variable.tip (local var) variable: any
                variable = variable.valor
                if(indiceI < variable.length) {
                    variable = variable[indiceI].valor
                    if(indiceJ < variable.length) {
                        return variable[indiceJ]
                    }
                    this.setError(`Indice fuera de rango. Indice ${indiceJ} en longitud ${variable.length}.`, linea, columna)
                    return null
                }
                this.setError(`Indice fuera de rango. Indice ${indiceI} en longitud ${variable.length}.`, linea, columna)
                return null
            }
            this.setError(`"${this.nombre}" no es un vector.`, linea, columna)
            return null
        }
        entorno = entorno.anterior
    }
    this.setError(`Acceso a variable inexistente "${this.nombre}".`, linea, columna)
    return null
}
```

Este cuenta con dos posiciones, esa es la principal diferencia con respecto al anteriormente descrito.

reasignarValorVector:

```
reasignarValorVector = (nombre, indiceI, valor, linea, columna) => {
    var entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            var simbolo = entorno.variables.get(nombre.toLowerCase())
            if(simbolo.tipo === Tipo.VECTOR) {
                var vector = simbolo.valor
                if(indiceI < vector.length) {
                    var vector = vector[indiceI]
                    if(vector.tipo === valor.tipo || vector.tipo === Tipo.DOUBLE && valor.tipo === Tipo.INT) {
                        simbolo.valor[indiceI] = valor
                        entorno.variables.set(nombre.toLowerCase(), simbolo)
                        return true
                    }
                    this.setError(`Los tipos no coinciden en la asignación. Intenta asignar un "${this.obtenerTipo(va
                    return false
                }
                this.setError(`Indice fuera de rango. Indice ${indiceI} en longitud ${vector.length}.`, linea, columr
                return false
            }
            this.setError(`Intenta acceder mediante índice a una variable "${this.obtenerTipo(simbolo.tipo)}".`, line
            return false
        }
        entorno = entorno.anterior
    }
    this.setError(`Resignación de valor a variable inexistente "${nombre}".`, linea, columna)
    return false
}
```

Primero se encuentra en donde queremos encontrar la posición, para para luego hacer las verificaciones y agregar el nuevo valor.

```
reasignarValorMatriz = (nombre, indiceI, indiceJ, valor, linea, columna) => {
    var entorno = this
    while(entorno) {
        if(entorno.variables.has(nombre.toLowerCase())) {
            var simbolo = entorno.variables.get(nombre.toLowerCase())
            if(simbolo.tipo === Tipo.MATRIZ) {
                var vector = simbolo.valor
                if(indiceI < vector.length) {
                    var vector = vector[indiceI].valor
                    if(indiceJ < vector.length) {
                        vector = vector[indiceJ]
                        if(vector.tipo === valor.tipo || vector.tipo === Tipo.DOUBLE && valor.tipo === Tipo.INT) {
                            simbolo.valor[indiceI].valor[indiceJ] = valor
                            entorno.variables.set(nombre.toLowerCase(), simbolo)
                            return true
                        }
                        this.setError(`Los tipos no coinciden en la asignación. Intenta asignar un "${this.obtenerTipo(valor.tipo)}"
                        return false
                    }
                    this.setError(`Indice fuera de rango. Indice ${indiceJ} en longitud ${vector.length}.`, linea, columna)
                    return false
                }
                this.setError(`Indice fuera de rango. Indice ${indiceI} en longitud ${vector.length}.`, linea, columna)
                return false
            }
            this.setError(`Intenta acceder mediante índice a una variable "${this.obtenerTipo(simbolo.tipo)}".`, linea, columna)
            return false
        }
        entorno = entorno.anterior
    }
    this.setError(`Resignación de valor a variable inexistente "${nombre}".`, linea, columna)
    return false
}
```

setPrint:

```
setPrint = (print) => {
    setConsola(print)
}
```

En esta función es donde se manda a imprimir, esta en entorno para hacer mas sencillo todo.

Match y setError:

```
match = (err, linea, columna) => {
    for(const s of errores) {
        if(`${s}` == `${{tipo: 'SEMANTICO', descripcion: err, linea: linea, columna: columna + 1}}`) {
            return true
        }
    }
    return false
}

setError = (errorD, linea, columna) => {
    if(!this.match(errorD, linea, columna)) {
        errores.push({tipo: 'SEMANTICO', descripcion: errorD,  linea: linea, columna: columna + 1})
    }
}
```

Aquí es donde se captan los errores.

**getGlobal**:

```
getGlobal = () => {
    let env = this
    while(env.previous) {
        env = env.previous
    }
    return env
}
```

Para obtener el global, lo cual es un método recursivo que regresa hasta la raíz.

**obtenerTipo:**

```
obtenerTipo = (tipo) => {
    if(tipo === Tipo.INT) {
        return 'int'
    }
    if(tipo === Tipo.DOUBLE) {
        return 'double'
    }
    if(tipo === Tipo.BOOL) {
        return 'bool'
    }
    if(tipo === Tipo.CHAR) {
        return 'char'
    }
    if(tipo === Tipo.STRING) {
        return 'std::string'
    }
    if(tipo === Tipo.VECTOR) {
        return 'Vector'
    }
    if(tipo === Tipo.MATRIZ) {
        return 'Matrix'
    }
    return 'NULL'
}
```

Ayuda a la obtención del tipo, es mas que todo para la verificación de que no haya errores.

**obtenerTipoVector:**

```
obtenerTipoVector = (tipo) => {
    if(tipo == Tipo.VECTOR) {
        return '[]'
    }
    return '[][]'
}
```

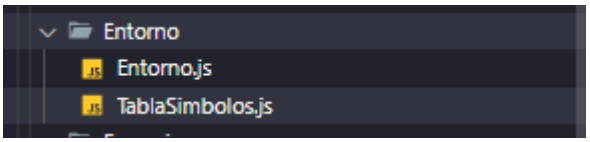
Para obtener el tipo de lista ya sea vector o matriz.

**obtenerTipoFunc:**

```
obtenerTipoFunc = (tipo) => {
    if(tipo === Tipo.INT) {
        return 'int'
    }
    if(tipo === Tipo.DOUBLE) {
        return 'double'
    }
    if(tipo === Tipo.BOOL) {
        return 'bool'
    }
    if(tipo === Tipo.CHAR) {
        return 'char'
    }
    if(tipo === Tipo.STRING) {
        return 'std::string'
    }
    return 'void'
}
}
```

Sirve para verificar el tipo de función que se va a obtener, ayuda básicamente a evitar errores potenciales.

## TABLA DE SIMBOLOS




Con la tabla de símbolo podremos verificar más fácilmente el tipo de Dato, el entorno al que pertenece, se utiliza mas que todo para facilitar los reportes.

**Push:**

```
push = (simbolo) => {
    if(this.validate(simbolo)) {
        this.simbolos.push(simbolo)
    }
}
```

Se verifica que el símbolo no haya sido ingresado, si esto no es así entonces se agrega a la lista.

**Validate:**

```
validate = (simbolo) => {
    for(const i of this.simbolos) {
        if(this.hash(i) == this.hash(simbolo)) {
            return false
        }
    }
    return true
}
```

Sirve para identificar que no se intente crear una variable dos veces y en el mismo entorno.

**Hash:**

```
hash = (simbolo) => {
    return `${simbolo.id}_${simbolo.tipo}_${simbolo.nameEnv}_${simbolo.linea}_${simbolo.columna}_${simbolo.tipoID}_${simbolo.tipo}`
}
```

Se recorre para determinar si el símbolo fue creado o no.

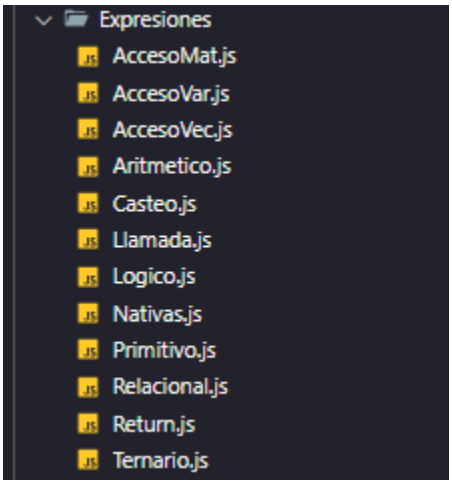
**Splice:**

```
splice = () => {
    this.simbolos.splice(0, this.simbolos.length)
}
```

Para limpiar todo.

Expresiones:

Todas las expresiones retornan un tipo de valor, esto es clave para ir Obteniendo los datos que se encuentran dentro de las expresiones que queremos evaluar.




**AcessoMat.js**: Funciona para acceder al valor de la matriz.

```
class AccesoMat extends Expresion {
    constructor(linea, columna, nombre, indiceI, indiceJ) {
        super(linea, columna, TipoExp.ACCESOMAT)
        this.nombre = nombre
        this.indiceI = indiceI
        this.indiceJ = indiceJ
    }

    execute = (entorno) => {
        const indiceI = this.indiceI.execute(entorno)
        const indiceJ = this.indiceJ.execute(entorno)
        const valor = entorno.obtenerPosicionMatriz(this.nombre, indiceI.valor, indiceJ.valor, this.linea, this.columna)
        if(valor) {
            return valor
        }
        return {valor: 'NULL', tipo: Tipo.NULL}
    }

    ast = () => {
        const nodo = new Nodo(this.nombre)
        nodo.insertarHijo(this.indiceI.ast())
        nodo.insertarHijo(this.indiceJ.ast())
        return nodo
    }
}
```

Se obtiene el índice I, J, para luego verificar el entorno y obtener el valor que se encuentra en la matriz. La función ast funciona para formar el árbol.

## AccesoVar

```
const { Expresion } = require("../Abstractas/Expresion")
const { Tipo } = require("../Utilities/Tipo")
const { TipoExp } = require("../Utilities/TipoExp")
const { Nodo } = require('../AST/Nodo')

class AccesoVar extends Expresion {
    constructor(linea, columna, nombre) {
        super(linea, columna, TipoExp.ACCESOVAR)
        this.nombre = nombre
    }

    execute = (entorno) => {
        const valor = entorno.obtenerVariable(this.nombre)
        if(valor) {
            return {valor: valor.valor, tipo: valor.tipo}
        }
        entorno.setError(`Acceso a variable inexistente "${this.nombre}".`, this.linea, this.columna)
        return {valor: 'NULL', tipo: Tipo.NULL}
    }

    ast = () => {
        return new Nodo(this.nombre)
    }
}

module.exports = { AccesoVar }
```

Al igual que el anterior retorna el valor de una variable en específico, como se hizo mención anteriormente no utilice la tabla de símbolos para sacar los valores, mas bien es con los entornos que voy sacando la información necesaria.

## AccesoVector:

```
const { Expresion } = require("../Abstractas/Expresion")
const { Tipo } = require("../Utilities/Tipo")
const { TipoExp } = require("../Utilities/TipoExp")
const { Nodo } = require('../AST/Nodo')

class AccesoVec extends Expresion {
    constructor(linea, columna, nombre, indiceI) {
        super(linea, columna, TipoExp.ACCESOVEC)
        this.nombre = nombre
        this.indiceI = indiceI
    }

    execute = (entorno) => {
        const indiceI = this.indiceI.execute(entorno)
        const valor = entorno.obtenerPosicionVector(this.nombre, indiceI.valor, this.linea, this.columna)
        if(valor) {
            return valor
        }
        return {valor: 'NULL', tipo: Tipo.NULL}
    }

    ast = () => {
        const nodo = new Nodo(this.nombre)
        nodo.insertarHijo(this.indiceI.ast())
        return nodo
    }
}

module.exports = { AccesoVec }
```

Funciona igual que la matriz, solo que aquí es para un vector.

## Aritmetico:

Para operaciones aritméticas, estas verifican y devuelven:

```
execute = (entorno) => {
    switch(this.signo) {
        case '+':
            return this.sum(entorno)
        case '-':
            if(this.exp1) {
                return this.res(entorno)
            }
            return this.neg(entorno)
        case '*':
            return this.mul(entorno)
        case '/':
            return this.div(entorno)
        case '%':
            return this.mod(entorno)
        case '^':
            return this.pow(entorno)
        default:
            return {valor: 'NULL', tipo: Tipo.NULL}
    }
}
```

## Casteo:

```
class Casteo extends Expresion {
    constructor(linea, columna, destino, valor) {
        super(linea, columna, TipoExp.CASTEO)
        this.destino = destino
        this.valor = valor
    }
```

Se analizan todos los casos posibles para los casteos que son posibles de ejecutar, cuando estos no son posibles se produce un error.

```
execute(entorno) {
    let valor = this.valor.execute(entorno)
    if(this.destino === Tipo.INT) {
        if(valor.tipo === Tipo.DOUBLE) {
            return {valor: parseInt(valor.valor), tipo: this.destino}
        }
        if(valor.tipo === Tipo.CHAR) {
            return {valor: valor.valor.charCodeAt(0), tipo: this.destino}
        }
        entorno.setError(`No hay casteo de "${valor.tipo}" a "${this.destino}"`, this.valor.linea, this.valor.columna)
        return {valor: 'NULL', tipo: Tipo.NULL}
    }
```

## Llamada

```
class Llamada extends Expresion {
    constructor(linea, columna, nombre, argumentos) {
        super(linea, columna, TipoExp.LLAMADAFUNC)
        this.nombre = nombre
        this.argumentos = argumentos
    }
```

Se utiliza para realizar la función:

```
execute = (entorno) => {
    const func = entorno.obtenerFuncion(this.nombre)
    if(func) {
        const entornoFunc = new Entorno(entorno.getGlobal(), `Funcion ${this.nombre.toLowerCase()}`)
        if(func.parametros.length == this.argumentos.length) {
            for(let i = 0; i < func.parametros.length; i ++) {
                const valor = this.argumentos[i].execute(entorno)
                const param = func.parametros
                if(valor.tipo == param[i][0] || valor.tipo === Tipo.DOUBLE && param[i][0] === Tipo.INT) {
                    entornoFunc.guardarVariable(param[i][1], valor.valor, valor.tipo, param[i][2], param[i][3])
                } else {
                    entorno.setError(`El Parámetro "${param.valor}" no es del tipo "${this.obtenerTipo(param.tipo)}".`, this.linea,
                    return
                }
            }
            const return_ = func.bloque.execute(entornoFunc)
            if(return_) {
                if(return_.valor === TipoExp.RETURN) {
                    return
                }
                return return_
            }
        } else {
            entorno.setError(`La Función "${this.nombre}" no tiene la cantidad correcta de parámetros.`, this.linea, this.columna)
        }
    } else {
        entorno.setError(`La Función "${this.nombre}" no existe.`, this.linea, this.columna)
    }
}
```

Primero se evalúan los parámetros, luego se guardan las variables en el entorno para luego ejecutar los bloques de la instrucción.

```
ast = () => {
    const nodo = new Nodo('LLAMADA')
    const llamada = new Nodo(this.nombre)
    for(const argumento of this.argumentos) {
        llamada.insertarHijo(argumento.ast())
    }
    nodo.insertarHijo(llamada)
    return nodo
}
```

Por ultimo se hace la llamada recursiva de la función.

## Logico

```
class Logico extends Expresion {
    constructor(linea, columna, exp1, signo, exp2) {
        super(linea, columna, TipoExp.LOGICO)
        this.exp1 = exp1
        this.signo = signo
        this.exp2 = exp2
    }

    execute = (entorno) => {
        switch(this.signo) {
            case '&&':
                return this.and(entorno)
            case '||':
                return this.or(entorno)
            case '!':
                return this.not(entorno)
            default:
                return {value: 'NULL', type: Type.NULL}
        }
    }
```

Al igual que el aritmético, aquí se hacen las expresiones lógicas.

## Nativas:

```
class Nativas extends Expresion {
    constructor(line, column, func, valor) {
        super(line, column, TipoExp.FUNCNATIVA)
        this.func = func
        this.valor = valor
    }

    execute = (entorno) => {
        var valor = this.valor.execute(entorno)
        switch(this.func.toLowerCase()) {
            case 'tolower':
                return {valor: valor.valor.toString().toLowerCase(), tipo: Tipo.STRING}
            case 'toupper':
                return {valor: valor.valor.toString().toUpperCase(), tipo: Tipo.STRING}
            case 'length':
                return {valor: valor.valor.length, tipo: Tipo.INT}
            case 'round':
                return {valor: Math.round(valor.valor), tipo: Tipo.INT}
            case 'typeof':
                return {valor: this.obtenerTypeOf(valor), tipo: Tipo.STRING}
            case 'std::tostring':
                return {valor: valor.valor.toString(), tipo: Tipo.STRING}
            case 'c_str':
                return {valor: this.obtenerCharArray(valor.valor), tipo: Tipo.VECTOR}
            default:
                return {valor: 'NULL', tipo: Tipo.NULL}
        }
    }
```

Se realizan las verificaciones correspondientes y se realizan las funciones nativas del lenguaje.

**Primitivos:**

```
execute = (_) => {
    switch (this.tipo) {
        case Tipo.INT:
            return { valor: parseInt(this.valor), tipo: this.tipo }
        case Tipo.DOUBLE:
            return { valor: parseFloat(this.valor), tipo: this.tipo }
        case Tipo.BOOL:
            return { valor: this.valor.toString().toLowerCase() === 'true', tipo: this.tipo }
        case Tipo.CHAR:
            this.valor = this.valor.replace(/\\n/g, '\n')
            this.valor = this.valor.replace(/\\t/g, '\t')
            this.valor = this.valor.replace(/\\"/g, '\"')
            this.valor = this.valor.replace(/\\'/g, '\'')
            this.valor = this.valor.replace(/\\\\/g, '\\')
            return { valor: this.valor, tipo: this.tipo }
        default:
            this.valor = this.valor.replace(/\\n/g, '\n')
            this.valor = this.valor.replace(/\\t/g, '\t')
            this.valor = this.valor.replace(/\\"/g, '\"')
            this.valor = this.valor.replace(/\\'/g, '\'')
            this.valor = this.valor.replace(/\\\\/g, '\\')
            return { valor: this.valor, tipo: this.tipo }
    }
}
```

Se modifican las cadenas para poder trabajarlas correctamente y se agregan los otros tipos de datos.

**Relacional:**

```
execute = (entorno) => {
    switch(this.signo) {
        case '==':
            return this.igual(entorno)
        case '!=':
            return this.diferente(entorno)
        case '>=':
            return this.mayorigual(entorno)
        case '<=':
            return this.menorigual(entorno)
        case '>':
            return this.mayor(entorno)
        case '<':
            return this.menor(entorno)
        default:
            return {valor: 'NULL',type: Type.NULL}
    }
}
```

Se verifican los tipos y se agregan los correspondientes.

**Return:**

```
class Return extends Expresion {
    constructor(linea, columna, expresion) {
        super(linea, columna, TipoExp.RETURN)
        this.expresion = expresion
    }

    execute = (entorno) => {
        if(this.expresion) {
            var valor = this.expresion.execute(entorno)
            return valor
        }
        return {valor: this.tipoExp, tipo: Tipo.NULL}
    }

    ast = () => {
        const nodo = new Nodo('RETURN')
        if(this.expresion) {
            nodo.insertarHijo(this.expresion.ast())
        }
        return nodo
    }
}

module.exports = { Return }
```

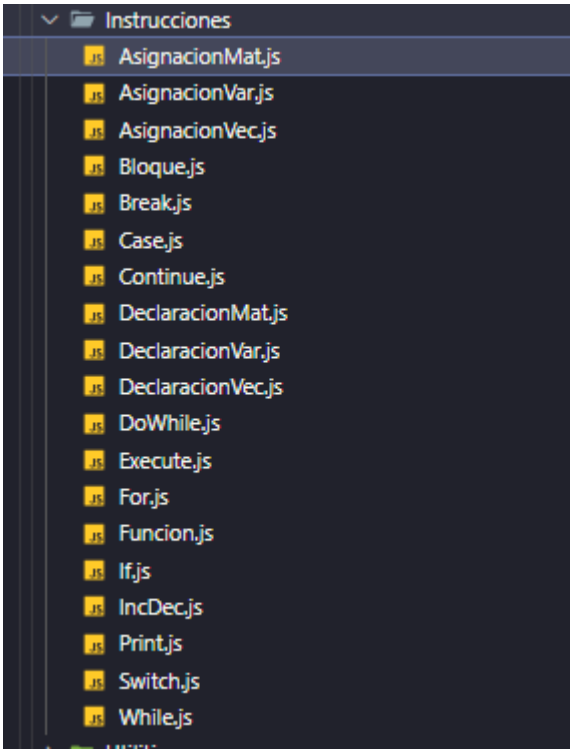
Es necesario para las funciones es aquí donde se obtiene la expresión.

## Ternario:

```
class Ternario extends Expresion {
    constructor(line, column, condicion, verdadero, falso) {
        super(line, column, Tipo.NULL, TipoExp.TERNARIO)
        this.condicion = condicion
        this.verdadero = verdadero
        this.falso = falso
    }

    execute = (entorno) => {
        const condicion = this.condicion.execute(entorno)
        if(condicion.tipo != Tipo.BOOL) {
            entorno.setError("El tipo de dato de la condición no es aceptable.", condicion.linea, condicion.columna)
            return {valor: 'NULL', tipo: Tipo.NULL}
        }
        if(condicion.valor) {
            const verdadero = this.verdadero.execute(entorno)
            return verdadero
        }
        const falso = this.falso.execute(entorno)
        return falso
    }

    ast = () => {
        const nodo = new Nodo('TERNARIO')
        const condicion = new Nodo('CONDICION')
        condicion.insertarHijo(this.condicion.ast())
        nodo.insertarHijo(condicion)
        const verdadero = new Nodo('VERDADERO')
        verdadero.insertarHijo(this.verdadero.ast())
        nodo.insertarHijo(verdadero)
        const falso = new Nodo('FALSO')
        falso.insertarHijo(this.falso.ast())
        nodo.insertarHijo(falso)
        return nodo
    }
```

Se analizan las dos condiciones y al igual que las funciones se realizan las ejecuciones de los bloques.

## Instrucción:




## AsignacionMat:

```
class AsignacionMat extends Instruccion {
    constructor(linea, columna, nombre, indiceI, indiceJ, valor) {
        super(linea, columna, TipoInst.ASIGMAT)
        this.nombre = nombre
        this.indiceI = indiceI
        this.indiceJ = indiceJ
        this.valor = valor
    }

    execute = (entorno) => {
        const valor = this.valor.execute(entorno)
        const indiceI = this.indiceI.execute(entorno)
        const indiceJ = this.indiceJ.execute(entorno)
        entorno.reasignarValorMatriz(this.nombre, indiceI.valor, indiceJ.valor, valor, this.linea, this.columna)
    }
```

Aquí es donde se puede ver el fruto del trabajo duro, se puede apreciar los execute de las expresiones. Se hace desde el entorno.

## AsignacionVar:

```
class AsignacionVar extends Instruccion {
    constructor(linea, columna, nombre, valor) {
        super(linea, columna, TipoInst.ASIGVAR)
        this.nombre = nombre
        this.valor = valor
    }

    execute = (entorno) => {
        const valor = this.valor.execute(entorno)
        entorno.reasignarValorVariable(this.nombre, valor, this.linea, this.columna)
    }

    ast = () => {
        const nodo = new Nodo('ASIGNACION')
        const igual = new Nodo('=')
        igual.insertarHijo(new Nodo(this.nombre))
        igual.insertarHijo(this.valor.ast())
        nodo.insertarHijo(igual)
        return nodo
    }
}
```

Se ejecutan las expresiones y se asignan las variables.

## AsignacionVec

```
class AsignacionVec extends Instruccion {
    constructor(linea, columna, nombre, indiceI, valor) {
        super(linea, columna, TipoInst.ASIGVAR)
        this.nombre = nombre
        this.indiceI = indiceI
        this.valor = valor
    }

    execute = (entorno) => {
        const valor = this.valor.execute(entorno)
        const indiceI = this.indiceI.execute(entorno)
        entorno.reasignarValorVector(this.nombre, indiceI.valor, valor, this.linea, this.columna)
    }

    ast = () => {
        const nodo = new Nodo('ASIGNACION')
        const igual = new Nodo('=')
        const identificador = new Nodo(this.nombre)
        identificador.insertarHijo(this.indiceI.ast())
        igual.insertarHijo(identificador)
        igual.insertarHijo(this.valor.ast())
        nodo.insertarHijo(igual)
        return nodo
    }
}
```

Lo mismo que la matriz solo que en vector.

**Bloque:**

```
class Bloque extends Instruccion {
    constructor(linea, columna, instrucciones) {
        super(linea, columna, TipoInst.BLOQUE)
        this.instrucciones = instrucciones
    }

    execute = (entorno) => {
        const entornoBloque = new Entorno(entorno, entorno.nombre)
        for(const instruccion of this.instrucciones) {
            try {
                const return_ = instruccion.execute(entornoBloque)
                if(return_) {
                    return return_
                }
            } catch(error) {}
        }
    }
```

Una de las clases mas importantes de todo el proyecto, por ejemplo cada función tiene un bloque, los bloques por así decirlo son un conjunto de expresiones y aquí es donde se ejecutan.

**Break:**

```
class Break extends Instruccion {
    constructor(linea, columna) {
        super(linea, columna, TipoInst.BREAK)
    }

    execute = (_) => {
        return {valor: this.tipoInst, tipo: Tipo.NULL}
    }

    ast = () => {
        return new Nodo('BREAK')
    }
}
```

Rompe cualquier ciclo que se este llevando a cabo y regresa al bloque anterior.

**Case:**

```
class Case extends Instruccion {
    constructor(linea, columna, caso, bloque) {
        super(linea, columna, TipoInst.CASE)
        this.caso = caso
        this.bloque = bloque
    }

    setCaso = (casoEvaluado) => {
        this.casoEvaluado = casoEvaluado
    }

    execute = (entorno) => {
        const entornoCase = new Entorno(entorno, entorno.nombre)
        var caso = this.caso.execute(entornoCase)
        if(caso.valor === this.casoEvaluado.valor) {
            var bloque = this.bloque.execute(entornoCase)
            if(bloque) {
                return bloque
            }
        }
    }
```

Al igual que todas las instrucciones se crea un nuevo entorno y en este caso se ejecuta el bloque del caso.

**Continue:**

```
class Continue extends Instruccion {
    constructor(linea, columna) {
        super(linea, columna, TipoInst.CONTINUE)
    }

    execute = (_) => {
        return {valor: this.tipoInst, tipo: Tipo.NULL}
    }

    ast = () => {
        return new Nodo('CONTINUE')
    }
}
```

Funciona similar que el break, solo que en lugar de romper el ciclo que se este llevando a cabo termina la ejecución actual para pasar a la siguiente iteración.

## DeclaracionMat

```
execute = (entorno) => {
    if(this.valores) {
        var hayError = false
        for(var i = 0; i < this.valores.length; i ++) {
            if(this.valores[i] instanceof Nativas) {
                if(this.valores[i].func === 'c_str') {
                    if(this.tipo1 === Tipo.CHAR) {
                        this.valores[i] = this.valores[i].execute(entorno)
                    } else {
                        entorno.setError(`Tipos incompatibles en vector.`, this.valores[i].linea, this.valores[i].columna)
                        hayError = true
                    }
                } else {
                    entorno.setError(`Valor no aceptado para el vector.`, this.valores[i].linea, this.valores[i].columna)
                    hayError = true
                }
            } else {
                for(var j = 0; j < this.valores[i].length; j ++) {
                    const valor = this.valores[i][j].execute(entorno)
                    if(this.tipo1 == valor.tipo) {
                        this.valores[i][j] = valor
                    } else {
                        entorno.setError(`Tipos incompatibles en vector.`, this.valores[i][j].linea, this.valores[i][j].columna)
                        hayError = true
                    }
                }
                this.valores[i] = {valor: this.valores[i], tipo: Tipo.VECTOR}
            }
        }
        if(!hayError) {
            entorno.guardarVector(this.nombre, this.valores, Tipo.MATRIZ, this.tipo1, this.linea, this.columna)
        }
    } else {
```

Se verifica si es una lista o no lo que recibe para luego ir determinando si hay un error en el tipo, luego de esto si no hay ningún error se termina de recorrer y guardar la información en el entorno correspondiente.

```
    } else {
        if(this.tipo1 == this.tipo2) {
            const longitudI = this.longitudI.execute(entorno)
            const longitudJ = this.longitudJ.execute(entorno)
            entorno.guardarVector(this.nombre, this.obtenerMatriz(longitudI.valor, longitudJ.valor), Tipo.MATRIZ, this.tipo1, thi
        } else {
            entorno.setError(`Tipos incompatibles en matriz.`, this.linea, this.columna)
        }
    }
}
```

En caso que no sea así pues solo se almacena el valor correspondiente.

## DeclaracionVar

```
execute = (entorno) => {
    var valor = null
    if(this.valor) {
        valor = this.valor.execute(entorno)
        if(this.tipo !== valor.tipo || this.tipo === Tipo.DOUBLE && valor.tipo === Tipo.INT) {
            entorno.setError(`Los tipos no coinciden en la asignación. Intenta asignar un "${this
            return
        }
    } else {
        switch(this.tipo) {
            case Tipo.INT:
                valor = {valor: 0, tipo: this.tipo}
                break
            case Tipo.DOUBLE:
                valor = {valor: 0.0, tipo: this.tipo}
                break
            case Tipo.BOOL:
                valor = {valor: true, tipo: this.tipo}
                break
            case Tipo.CHAR:
                valor = {valor: '0', tipo: this.tipo}
                break
            case Tipo.STRING:
                valor = {valor: "", tipo: this.tipo}
                break
        }
    }
    for(const id of this.identificadores) {
        entorno.guardarVariable(id, valor.valor, this.tipo, this.linea, this.columna)
    }
```

Se ejecuta la expresión y si no hay ningún valor asignado se inicializan los datos.

## DeclaracionVec

```
class DeclaracionVec extends Instruccion {
    constructor(linea, columna, nombre, tipo1, tipo2, longitudI, valores) {
        super(linea, columna, TipoInst.DECVEC)
        this.nombre = nombre
        this.tipo1 = tipo1
        this.tipo2 = tipo2
        this.longitudI = longitudI
        this.valores = valores
    }
```

Es casi que lo mismo que la matriz solo que con el vector.

## DoWhile

```
class DoWhile extends Instruccion {
    constructor(line, column, condicion, bloque) {
        super(line, column, TipoInst.DOWHILE)
        this.condicion = condicion
        this.bloque = bloque
    }

    execute = (entorno) => {
        const entornoWhile = new Entorno(entorno, entorno.nombre)
        var condicion = null
        do {
            var bloque = this.bloque.execute(entornoWhile)
            if(bloque) {
                if(bloque.valor === TipoInst.CONTINUE) {
                    condicion = this.condicion.execute(entornoWhile)
                    continue
                }
                else if(bloque.valor === TipoInst.BREAK) {
                    break
                }
                return bloque
            }
            condicion = this.condicion.execute(entornoWhile)
        } while(condicion.valor)
    }
```

Ejecuta las instrucciones siempre y cuando no se salga con un break o que se cumpla la condición regresa el bloque.

**Execute**

```
class Execute extends Instruccion {
    constructor(linea, columna, funcion) {
        super(linea, columna, TipoInst.EXECUTE)
        this.funcion = funcion
    }

    execute = (entorno) => {
        this.funcion.execute(entorno)
    }

    ast = () => {
        const nodo = new Nodo('EXECUTE')
        nodo.insertarHijo(this.funcion.ast())
        return nodo
    }
}

module.exports = { Execute }
```

Se realiza para ejecutar las funciones con su entorno correspondiente.

**For**

```
class For extends Instruccion {
    constructor(line, column, args, bloque) {
        super(line, column, TipoInst.FOR)
        this.args = args
        this.bloque = bloque
    }

    execute = (entorno) => {
        const entornoFor = new Entorno(entorno, entorno.nombre)
        this.args[0].execute(entornoFor)
        var condicion = this.args[1].execute(entornoFor)
        while(condicion.valor) {
            var bloque = this.bloque.execute(entornoFor)
            if(bloque) {
                if(bloque.valor === TipoInst.CONTINUE) {
                    this.args[2].execute(entornoFor)
                    condicion = this.args[1].execute(entornoFor)
                    continue
                }
                else if(bloque.valor === TipoInst.BREAK) {
                    break
                }
                return bloque
            }
            this.args[2].execute(entornoFor)
            condicion = this.args[1].execute(entornoFor)
        }
    }
```

Si, este for no funciona con un for, lo hace con un while, se hizo así para reutilizar de alguna manera la lógica del while. Pero básicamente se verifica que las condiciones de break o que se cumplan las iteraciones se cumplan, también toma en cuenta el break.

## Función

```
class Funcion extends Instruccion {
    constructor(linea, columna, nombre, parametros, bloque, tipo) {
        super(linea, columna, TipoInst.DECFUNC)
        this.nombre = nombre
        this.parametros = parametros
        this.bloque = bloque
        this.tipo = tipo
    }

    execute = (entorno) => {
        entorno.guardarFuncion(this.nombre, this)
    }

    ast = () => {
```

Aquí es donde se van guardando los bloques para luego ejecutarlos, también almacena el tipo.

## If

```
class If extends Instruccion {
    constructor(linea, columna, condicion, bloque, _else_) {
        super(linea, columna, TipoInst.IF)
        this.condicion = condicion
        this.bloque = bloque
        this._else_ = _else_
    }

    execute = (entorno) => {
        var condicion = this.condicion.execute(entorno)
        if(condicion.valor) {
            var bloque = this.bloque.execute(entorno)
            if(bloque) {
                return bloque
            }
            return
        }
        if(this._else_) {
            var _else_ = this._else_.execute(entorno)
            if(_else_) {
                return _else_
            }
        }
    }
```

Este se forma de manera recursiva en la gramática y se va ejecutando siempre y cuando cumpla con la condición correspondiente.

## IncDec

```
class IncDec extends Instruccion {
    constructor(line, column, nombre, signo) {
        super(line, column, signo == '++' ? TipoInst.INC : TipoInst.DEC)
        this.nombre = nombre
        this.signo = signo
    }

    execute = (entorno) => {
        var valor = entorno.obtenerVariable(this.nombre)
        if(!valor) {
            entorno.setError(`Acceso a variable inexistente "${this.nombre}".`, this.linea, this.columna)
            return
        }
        if(valor.tipo !== Tipo.INT && valor.tipo !== Tipo.DOUBLE) {
            entorno.setError(`Solo se puede usar el incremento o decremento para tipos "INT" o "DOUBLE".`, this.linea, this.columna)
            return
        }
        switch(this.signo) {
            case '++':
                valor.valor += 1
                entorno.reasignarValorVariable(this.nombre, {valor: valor.valor, tipo: Tipo.INT})
                valor = entorno.obtenerVariable(this.nombre)
                break
            default:
                valor.valor -= 1
                entorno.reasignarValorVariable(this.nombre, {valor: valor.valor, tipo: Tipo.INT})
                valor = entorno.obtenerVariable(this.nombre)
                break
        }
    }
```

Sirve para sumar o restar una variable con uno mas o uno menos.

## Print

```
class Print extends Instruccion {
    constructor(linea, columna, expresion, endl) {
        super(linea, columna, TipoInst.PRINT)
        this.expresion = expresion
        this.endl = endl
    }

    execute = (entorno) => {
        const valor = this.expresion.execute(entorno)
        entorno.setPrint(`${valor.valor}`)
        if(this.endl) {
            entorno.setPrint('\n')
        }
    }

    ast = () => {
        const nodo = new Nodo('PRINT')
        nodo.insertarHijo(this.expresion.ast())
        if(this.endl) {
            nodo.insertarHijo(new Nodo('endl'))
        }
        return nodo
    }
}
```

Determina si es con salto de línea o no, imprime.

**Switch:**

```
class Switch extends Instruccion {
    constructor(linea, columna, argumento, casos, default_) {
        super(linea, columna, TipoInst.SWITCH)
        this.argumento = argumento
        this.casos = casos
        this.default_ = default_
    }

    execute = (entorno) => {
        const entornoSwitch = new Entorno(entorno, entorno.nombre)
        if(this.casos) {
            const argumento = this.argumento.execute(entorno)
            for(const caso of this.casos) {
                caso.setCaso(argumento)
                const caso_exe = caso.execute(entornoSwitch)
                if(caso_exe) {
                    if(caso_exe.valor === TipoExp.RETURN) {
                        return
                    }
                    else if(caso_exe.valor === TipoInst.BREAK) {
                        return
                    }
                    return caso_exe
                }
            }
        }
        if(this.default_) {
            const default_ = this.default_.execute(entornoSwitch)
            if(default_) {
                if(default_.valor === TipoExp.RETURN) {
                    return
                }
                else if(default_.valor === TipoInst.BREAK) {
                    return
                }
                return default_
            }
        }
    }
```

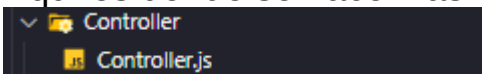
Se usan los casos para ejecutar básicamente y se verifican tanto el return como el break.

**While:**

```
class While extends Instruccion {
    constructor(line, column, condicion, bloque) {
        super(line, column, TipoInst.WHILE)
        this.condicion = condicion
        this.bloque = bloque
    }

    execute = (entorno) => {
        const entornoWhile = new Entorno(entorno, entorno.nombre)
        var condicion = this.condicion.execute(entornoWhile)
        while(condicion.valor) {
            var bloque = this.bloque.execute(entornoWhile)
            if(bloque) {
                if(bloque.valor === TipoInst.CONTINUE) {
                    condicion = this.condicion.execute(entornoWhile)
                    continue
                }
                else if(bloque.valor === TipoInst.BREAK) {
                    break
                }
                return bloque
            }
            condicion = this.condicion.execute(entornoWhile)
        }
    }
```

Se ejecuta igual que el for solo que este si es un while, se analizan las condiciones y se analizan los continue y break.

Aquí es donde se hacen las instrucciones.




```
    test = (req, res) => {
        const { ruta } = req.body;


        let resultado = analizador.parse(ruta.toString())
        console.log(resultado)
        let interpretacion = this.interpretar(resultado)
        //console.log(interpretacion);
        res.json({message: "Funcion analizar", salida: interpretacion})

    }
```

La petición se debe hacer a:

localhost:4000/test

Json:

{
Ruta: “Texto que se recibe desde el frontend”
}

```
interpretar = (instrucciones) => {
    try {
        tablaSimbolos.splice()
        reiniciarSalidas()
        const entornoGlobal = new Entorno(null, 'Global')
        var execute = null
        var ast = new Nodo('INSTRUCCIONES')
        for(const instruccion of instrucciones) {
            try {
                if(instruccion instanceof Execute) {
                    execute = instruccion
                } else {
                    instruccion.execute(entornoGlobal)
                }
                ast.insertarHijo(instruccion.ast())
            } catch(error) {}
        }
        if(execute) {
            execute.execute(entornoGlobal)
        }

        this.escribirAST(ast.obtenerGrafo())
        return {
            consola: getConsole(),
            ast: ast.obtenerGrafo(),
            tablasimbolos: tablaSimbolos,
            errores: errores,
        }
    } catch(error) {
        return {
            consola: error,
```

Se limpia todo, luego se inicializa el árbol, con lo que se obtiene de la gramática tenemos una lista de instrucciones o de “funciones” con esto se busca el execute que es el main y se realiza todo, de ultimo se forma el árbol de manera recursiva.