Universidad San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Laboratorio de Organización y Compiladores 1, Sección B

# CARATULA

## Proyecto de lenguaje de programación, Typesty.

## Manual Técnico.

PERSONA QUE PRESENTA EL MANUAL:

| | |
|---|---|
| Nombre: | Angel Oswaldo Arteaga García |
| Carné: | 201901816 |

DATOS DEL SOLICITANTE:

| | |
|---|---|
| Catedrático: | Ing. Manuel Castillo |
| Auxiliar: | Erick Lemus |
| Auxiliar: | René Corona |

# Manual Técnico

- Archivo jison encargado del análisis sintáctico y léxico:

Este archivo es el encargado de reconocer cada token que nuestro lenguaje analiza, además de reconocer el orden que hay detrás de cada token a la hora de estar leyendo la cadena de entrada.

Además de solo reconocer el cada símbolo y token de la cadena, también se encarga de realizar ciertas funciones que generan el AST que luego explicaremos:

```
EXPRESION: EXPRESION suma EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.SUMA,this._$.first_line,this._$.first_column+1);}
         | EXPRESION menos EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.RESTA,this._$.first_line,this._$.first_column+1);}
         | EXPRESION multi EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MULT,this._$.first_line,this._$.first_column+1);}
         | EXPRESION div EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.DIV,this._$.first_line,this._$.first_column+1);}
         | EXPRESION exponente EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.EXP,this._$.first_line,this._$.first_column+1);}
         | EXPRESION modulo EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MOD,this._$.first_line,this._$.first_column+1);}
         | menos EXPRESION %prec umenos {$$= INSTRUCCION.nuevaOperacionBinaria($2,$2, TIPO_OPERACION.MEN,this._$.first_line,this._$.first_column+1);}
         | parA EXPRESION parC {$$=$2}
         | EXPRESION igualigual EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.IGUALIGUAL,this._$.first_line,this._$.first_column+1);}
         | EXPRESION diferente EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.DIFERENTE,this._$.first_line,this._$.first_column+1);}
         | EXPRESION menor EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MENOR,this._$.first_line,this._$.first_column+1);}
         | EXPRESION menorigual EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MENORIGUAL,this._$.first_line,this._$.first_column+1);}
         | EXPRESION mayor EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MAYOR,this._$.first_line,this._$.first_column+1);}
         | EXPRESION mayorigual EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.MAYORIGUAL,this._$.first_line,this._$.first_column+1);}
         | EXPRESION or EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.OR,this._$.first_line,this._$.first_column+1);}
         | EXPRESION and EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.AND,this._$.first_line,this._$.first_column+1);}
         | not EXPRESION {$$= INSTRUCCION.nuevaOperacionBinaria($2,$2, TIPO_OPERACION.NOT,this._$.first_line,this._$.first_column+1);}
         | parA TIPO parC EXPRESION %prec umenos {$$= INSTRUCCION.nuevaOperacionBinaria($2,$4, TIPO_OPERACION.CAST,this._$.first_line,this._$.first_column+1);}
         | EXPRESION sumasuma {$$= INSTRUCCION.nuevaOperacionBinaria($1,$1, TIPO_OPERACION.MASMAS,this._$.first_line,this._$.first_column+1);}
         | EXPRESION menosmenos {$$= INSTRUCCION.nuevaOperacionBinaria($1,$1, TIPO_OPERACION.MENOSMENOS,this._$.first_line,this._$.first_column+1);}
         | identificador corA EXPRESION corC // para vectores
         | identificador corA corA EXPRESION corC corC // para listas
         | identificador parA  parC {$$= INSTRUCCION.nuevaOperacionBinaria($1,null, TIPO_OPERACION.LLAMADA,this._$.first_line,this._$.first_column+1);} //LLAMADA
         | identificador parA LISTAVALORES parC {$$= INSTRUCCION.nuevaOperacionBinaria($1,$3, TIPO_OPERACION.LLAMADA,this._$.first_line,this._$.first_column+1);} //LLAMADA
         | NUMBER {$$ = INSTRUCCION.nuevoValor(Number($1), TIPO_VALOR.DOUBLE, this._$.first_line,this._$.first_column+1)}
         | entero {$$ = INSTRUCCION.nuevoValor(Number($1), TIPO_VALOR.DECIMAL, this._$.first_line,this._$.first_column+1)}
         | true {$$ = INSTRUCCION.nuevoValor(($1), TIPO_VALOR.BANDERA, this._$.first_line,this._$.first_column+1)}
         | false {$$ = INSTRUCCION.nuevoValor($1, TIPO_VALOR.BANDERA, this._$.first_line,this._$.first_column+1)}
         | cadena {$$ = INSTRUCCION.nuevoValor($1, TIPO_VALOR.CADENA, this._$.first_line,this._$.first_column+1)}
         | identificador {$$ = INSTRUCCION.nuevoValor($1, TIPO_VALOR.IDENTIFICADOR, this._$.first_line,this._$.first_column+1)}
         | caracter {$$ = INSTRUCCION.nuevoValor($1, TIPO_VALOR.CARACTER, this._$.first_line,this._$.first_column+1)}
;

PRINT: print parA EXPRESION parC ptcoma {$$ = new INSTRUCCION.nuevoCout($3, this._$.first_line,this._$.first_column+1)}
```

- Archivo Analizadores.js encargado de dar servicio:

Este archivo solo se encarga de poder recibir la cadena por medio de una ruta dentro de un servidor, y así mismo se encarga de mandar a analizar esa cadena y así poder devolver un mensaje que seria la consola de la entrada.

```
module.exports = (parser, app) => {
    app.post('/analizar', (req, res) => {
        //Reseteamos las listas
        while (ListaSimbolos.length > 0) {
            ListaSimbolos.pop();
        }
        while (ListaErrores.length > 0) {
            ListaErrores.pop();
        }
        //ya leemos
        var prueba = req.body.prueba
        var ast = parser.parse(prueba)

        var ast = parser.parse(prueba)
        const AmbitoGlobal = new Ambito(null, "Global")
        //var cadena = Bloque(ast, AmbitoGlobal)
        var cadena = Global(ast, AmbitoGlobal)
        var grafica = new Graficador(ast)
        var dot = grafica.graficar()
        var resultado = {
            arbol: ast,
            consola: cadena, //cadena
            tablaSimbolos: ListaSimbolos,
            tablaErrores: ListaErrores
        }
        //GENERAMOS EL ARCHIVO DOT
        fs.writeFile("./controller/ReporteAST/AST.dot", dot, function (error) {
            if (error) {
                console.log(error);
            }
        })
        //RESPUESTA
        res.send(resultado)
    })
```

- Archivo Ámbito,js:

Este archivo es el que posee una clase Ámbito que se encarga de poder almacenar los métodos, las funciones y las variables que el lenguaje vaya leyendo, además de eso tiene métodos que nos sirven para poder verificar si ya existe una variable, método o función y a su vez para poder agregar cualquiera de estos.

```
3   class Ambito{
4       constructor(_anterior, _entorno){
5           this.anterior = _anterior;
6           this.entorno = _entorno;
7           this.tablaSimbolos = new Map();
8           this.tablaMetodos = new Map();
9           this.tablaFunciones = new Map();
10      }
11
12      addSimbolo(_s, _simbolo){
13          this.tablaSimbolos.set(_s.toLowerCase(), _simbolo)
14          var cadenaEntorno = ""
15          var i = 0
16          for(let e=this; e!=null; e=e.anterior){
17              if(i == 0){
18                  cadenaEntorno = e.entorno
19              } else {
20                  cadenaEntorno = e.entorno + "=>" + cadenaEntorno
21              }
22              i++
23          }
24          //INGRESAMOS A LA TABLA DE SIMBOLOS
25          var existe = false
26          for(let i = 0; i < ListaSimbolos.length; i++){
27              if(ListaSimbolos[i].Entorno == cadenaEntorno && ListaSimbolos[i].Identificador == _simbolo.id){
28                  existe = true;
29                  break;
30              }
31          }
32          var sim = {
33              Identificador: _simbolo.id,
34              TipoVar: "Variable",
35              Tipo: _simbolo.tipo,
36              Entorno: cadenaEntorno,
37              Linea: _simbolo.linea,
38              Columna: _simbolo.columna
39          }
40          if(!existe){
41              ListaSimbolos.push(sim)
42          }
43      }
```

- Archivo Función.js, Método.js, Símbolo.js:

Estos archivos son muy simples, solo poseen sus respectivas clases y sus constructores recibiendo los parámetros necesarios para cada uno.

```
class Funcion{
    constructor(_id, _tipo, _lista_parametros, _instrucciones, _linea, _columna){
        this.id = _id;
        this.tipo = _tipo;
        this.lista_parametros = _lista_parametros;
        this.instrucciones = _instrucciones;
        this.linea = _linea;
        this.columna = _columna;
    }
}

module.exports = Funcion
```

- Archivo ListaErrores.js y ListaSimbolos.js:

Estos archivos solo exportan un arreglo donde se estarán almacenando la tabla de errores y de símbolos:

```
var ListaErrores = [];

module.exports = ListaErrores;
```

- Archivos TipoDato.js, TipoInstruccion.js, TipoOperacion.js y TipoValor.js:

Estos archivos son todos similares, solo declaran una pequeña estructura del tipo de variable y su valor.

```
const TIPO_VALOR={
    DECIMAL:        'VAL_DECIMAL',
    CADENA:         'VAL_CADENA',
    BANDERA:        'VAL_BANDERA',
    CARACTER:       'VAL_CARACTER',
    DOUBLE:         'VAL_DOUBLE',
    IDENTIFICADOR: 'VAL_IDENTIFICADOR'
}

module.exports = TIPO_VALOR
```

- Archivo Instrucción.js:

Este archivo es super importante dentro del análisis de nuestro lenguaje ya que este es el encargado de realizar el árbol del lenguaje para luego poderlo ejecutar dependiendo del tipo de instrucción que venga, el analizador sintáctico usa todas las funciones para poder obtener los datos y así irlos seteando.

```
    nuevaAsignacion: function(_id, _expresion, _linea, _columna){
        return {
            tipo: TIPO_INSTRUCCION.ASIGNACION,
            id: _id,
            expresion: _expresion,
            linea: _linea,
            columna: _columna
        }
    },
    nuevoWhile: function(_expresion, _instrucciones, _linea, _columna){
        return {
            tipo: TIPO_INSTRUCCION.WHILE,
            expresion: _expresion,
            instrucciones: _instrucciones,
            linea: _linea,
            columna: _columna
        }
    },
    nuevoDoWhile: function(_expresion, _instrucciones, _linea, _columna){
        return {
            tipo: TIPO_INSTRUCCION.DOWHILE,
            expresion: _expresion,
            instrucciones: _instrucciones,
            linea: _linea,
            columna: _columna
        }
    },
    nuevoFor: function(_declaracion, _condicion, _actualizacion, _instrucciones,  _linea, _columna){
        return {
            tipo: TIPO_INSTRUCCION.FOR,
            declaracion: _declaracion,
            condicion: _condicion,
            actualizacion: _actualizacion,
            instrucciones: _instrucciones,
            linea: _linea,
            columna: _columna
        }
    },
```

- Archivo Global.js:

Este es el encargado de iniciar todo el análisis del árbol generado por el archivo de arriba, este realiza 3 pasadas al árbol, la primera es para ver si viene exactamente solo un exec dentro del archivo, la segunda pasada es para la declaración de las variables, funciones y métodos y la tercera pasada sirve para ejecutar como tal el árbol que generamos anteriormente.

```
function Global(_instrucciones, _ambito) {
    var cadena = ""
    //PRIMERA PASADA, QUE VENGA UN EXEC
    var contadorExec = 0
    for (let i = 0; i < _instrucciones.length; i++) {
        if (_instrucciones[i].tipo === TIPO_INSTRUCCION.EXEC) {
            contadorExec++;
        }
    }
    if (contadorExec == 0) {
        var err = {
            TipoError: "Semántico",
            Descripcion: `No se ha encontrado un Exec() para ser ejecutado`,
            Linea: 0,
            Columna: 0
        }
        ListaErrores.push(err)
        return `Error: No se ha encontrado un Exec() para ser ejecutado\n`
    }
    if (contadorExec > 1) {
        var err = {
            TipoError: "Semántico",
            Descripcion: `Se han encontrado mas de un Exec() para ejecutar`,
            Linea: 0,
            Columna: 0
        }
        ListaErrores.push(err)
        return `Error: Se han encontrado mas de un Exec() para ejecutar\n`
    }
    //SEGUNDA PASADA, DECLARAR VARIABLES, METODOS Y ASIGNACIONES
```

- Archivo Bloque.js:

Este archivo es el encargado de leer todas las instrucciones que se fueron creando dentro de la primera fase de análisis y dependiendo de cual sea el tipo de instrucción, así será su ejecución, esta función devuelve un objeto con un valor en caso de llamar instrucciones desde una operación y devuelve un mensaje para concatenar a la consola.

```
26      _instrucciones.forEach(instruccion => {
27          if(existeBreak || existeContinue || existeReturn){
28              return {
29                  existeBreak: existeBreak,
30                  existeContinue: existeContinue,
31                  existeReturn: existeReturn,
32                  valor: valor,
33                  cadena: cadena
34              }
35          }
36          if(instruccion.tipo === TIPO_INSTRUCCION.COUT){
37              cadena+=Cout(instruccion, _ambito)+'\n'
38          }
39          else if(instruccion.tipo === TIPO_INSTRUCCION.DECLARACION){
40              var mensaje = Declaracion(instruccion, _ambito)
41              if(mensaje!=null){
42                  cadena+=mensaje
43              }
44          }
45          else if(instruccion.tipo === TIPO_INSTRUCCION.ASIGNACION){
46              var mensaje = Asignacion(instruccion, _ambito)
47              if(mensaje!=null){
48                  cadena+=mensaje
49              }
50          }
51          else if(instruccion.tipo === TIPO_INSTRUCCION.WHILE){
52              var exec = CicloWhile(instruccion, _ambito)
53              var mensaje = exec.cadena
54              valor = exec.valor
55              existeBreak = false
56              existeContinue = false
57              existeReturn = false
58              if(mensaje!=null){
59                  cadena+=mensaje
60              }
61          }
```

❖ Archivo Asignacion.js:

Este archivo se encarga de brindar una función que recibe una expresión de tipo asignación, verifica si ya existe ese símbolo y si existe, le actualiza el nuevo valor a ese símbolo.

```
function Asignacion(_instruccion, _ambito){
    const id = _instruccion.id;
    const existe = _ambito.existeSimbolo(id)
    if(existe){
        var valor = Operacion(_instruccion.expresion, _ambito)

        var simbolo = _ambito.getSimbolo(id)
        var tipos = {
            tipoSimbolo: simbolo.tipo,
            tipoNuevoValor: valor.tipo
        }
        if(tipos.tipoSimbolo===tipos.tipoNuevoValor){
            simbolo.valor = valor.valor
            _ambito.actualizar(id,simbolo)
            return valor.mensaje
        }
        var err = {
            TipoError: "Semántico",
            Descripcion: "No es posible asignar un valor de tipo "+tipos.tipoNuevoValor+" a la variable '"+ id +"' que es de tipo "+tipos.tipoSimbolo,
            Linea: _instruccion.linea,
            Columna: _instruccion.columna
        }
        ListaErrores.push(err)
        return "Error: No es posible asignar un valor de tipo "+tipos.tipoNuevoValor+" a la variable \n'"+ id +"' que es de tipo "+tipos.tipoSimbolo+"... Linea: "+_instruccion.linea+"
    }
    var err = {
        TipoError: "Semántico",
        Descripcion: `La variable '${String(id)}' no existe`,
        Linea: _instruccion.linea,
        Columna: _instruccion.columna
    }
    ListaErrores.push(err)
    return `Error: la variable '${String(id)}' no existe... Linea: ${_instruccion.linea} Columna: ${_instruccion.columna}`+"\n"
}
```

❖ Archivo Cout.js:

Este es el encargado se recibir una expresión, llama a procesar la cadena y la devulve.

```
const procesarCadena = require("../Operacion/procesarCadena")

function Cout(_instruccion, _ambito){
    const cadena = procesarCadena(_instruccion.expresion, _ambito).valor
    return cadena
}

module.exports = Cout
```

❖ Archivo DecFuncion.js, DecMetodo.js, Declaración.js y DecParametro:

Estos 3 archivos son lo mismo, todos verifican dentro del ámbito entrante si ya existe un símbolo con el mismo nombre y si no, procede a guardar el símbolo dentro del ámbito realizando previamente las verificaciones de tipo y valor de la expresión.

```
function DecFuncion(_instruccion, _ambito){
    //console.log(_instruccion.lista_parametros)
    const nuevaFuncion = new Funcion(_instruccion.nombre, _instruccion.tipoFuncion, _instruccion.lista_parametros, _instruccion.instrucciones, _instruccion.linea, _instruccion.columna)
    //Si el nombre ya existe como simbolo
    if(_ambito.existeSimbolo(nuevaFuncion.id)!=false){
        var err = {
            TipoError: "Semántico",
            Descripcion: `No se puede declarar un metodo con el mismo nombre de una variable "${nuevaFuncion.id}"`,
            Linea: nuevaFuncion.linea,
            Columna: nuevaFuncion.columna
        }
        ListaErrores.push(err)
        return `Error: No se puede declarar una funcion con el mismo nombre de una variable "${nuevaFuncion.id}" Linea: ${nuevaFuncion.linea} Columna: ${nuevaFuncion.columna}\n`
    }
    //Verificamos si el metodo ya existe
    else if(_ambito.existeMetodo(nuevaFuncion.id)!=false){
        var err = {
            TipoError: "Semántico",
            Descripcion: `Ya existe un metodo con el nombre: "${nuevaFuncion.id}"`,
            Linea: nuevaFuncion.linea,
            Columna: nuevaFuncion.columna
        }
        ListaErrores.push(err)
        return `Error: Ya existe un metodo con el nombre: "${nuevaFuncion.id}" Linea: ${nuevaFuncion.linea} Columna: ${nuevaFuncion.columna}\n`
    }
    else if(_ambito.existeFuncion(nuevaFuncion.id)!=false){
        var err = {
            TipoError: "Semántico",
            Descripcion: `La Funcion "${nuevaFuncion.id}" ya existe`,
            Linea: nuevaFuncion.linea,
            Columna: nuevaFuncion.columna
        }
        ListaErrores.push(err)
        return `Error: La Funcion "${nuevaFuncion.id}" ya existe, Linea: ${nuevaFuncion.linea} Columna: ${nuevaFuncion.columna}\n`
    }
    //Si no hay error guardamos
    _ambito.addFuncion(nuevaFuncion.id, nuevaFuncion)
    return null
}

module.exports = DecFuncion
```

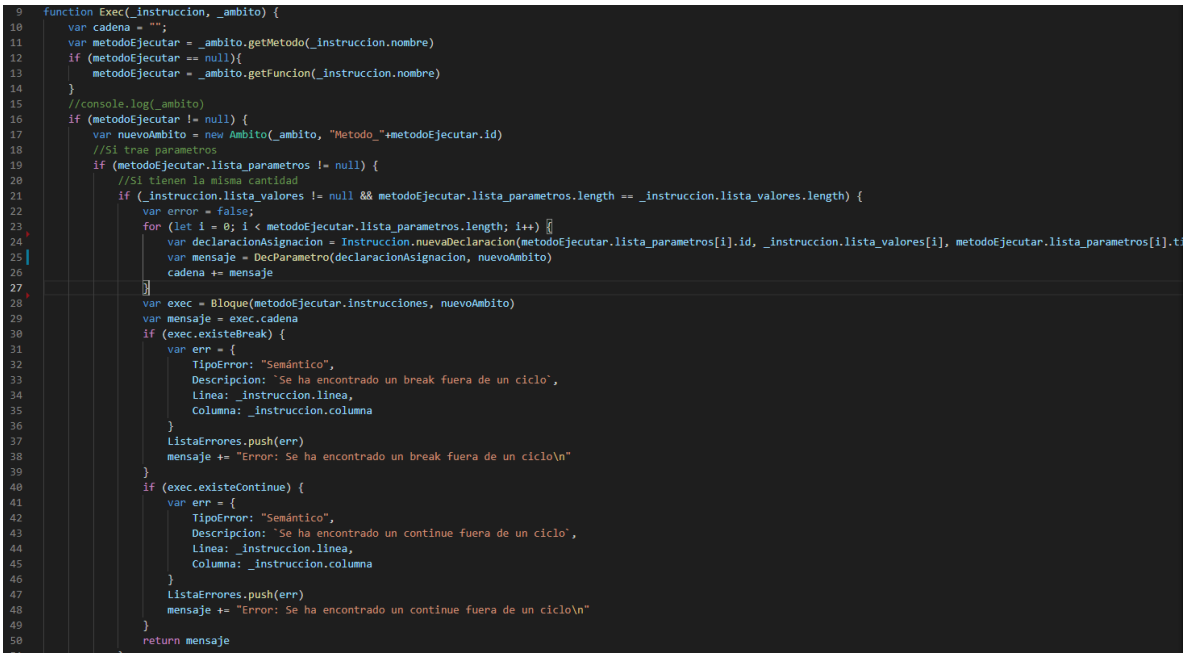
- ❖ Archivo DoWhile.js y While.js:

Este archivo contiene una función que devuelve una cadena, primero se verifica si la condición del while es correcta y si es así, empieza a iterar el bloque de instrucciones que posee y actualizando el operador.

```
function CicloDoWhile(_instruccion, _ambito) {
    var mensaje = ""
    var valor = {
        valor: null,
        tipo: null,
        mensaje: "",
        linea: _instruccion.linea,
        columna: _instruccion.columna
    }
    var operacion = Operacion(_instruccion.expresion, _ambito)
    mensaje += operacion.mensaje
    if (operacion.tipo === TIPO_DATO.BANDERA) {
        do {
            var nuevoAmbito = new Ambito(_ambito, "DoWhile")
            const Bloque = require('./Bloque')
            var exec = Bloque(_instruccion.instrucciones, nuevoAmbito)
            mensaje += exec.cadena
            valor = exec.valor
            if (exec.existeBreak) {
                return {
                    cadena: mensaje,
                    valor: valor
                }
            }
            //actualizamos
            operacion = Operacion(_instruccion.expresion, _ambito)
            mensaje += operacion.mensaje
        } while (operacion.valor);
        return {
            cadena: mensaje,
            valor: valor
        }
    }
```

- Archivo Exec.js:

Este archivo recibe una instrucción y revisa si la función o parámetro que se está llamando se encuentra dentro del ámbito y si se encuentra, procede a ejecutar el cuerpo de ese método o función con ayuda del Bloque.



```
function Exec(_instruccion, _ambito) {
    var cadena = "";
    var metodoEjecutar = _ambito.getMetodo(_instruccion.nombre)
    if (metodoEjecutar == null){
        metodoEjecutar = _ambito.getFuncion(_instruccion.nombre)
    }
    //console.log(_ambito)
    if (metodoEjecutar != null) {
        var nuevoAmbito = new Ambito(_ambito, "Metodo_"+metodoEjecutar.id)
        //Si trae parametros
        if (metodoEjecutar.lista_parametros != null) {
            //Si tienen la misma cantidad
            if (_instruccion.lista_valores != null && metodoEjecutar.lista_parametros.length == _instruccion.lista_valores.length) {
                var error = false;
                for (let i = 0; i < metodoEjecutar.lista_parametros.length; i++) {
                    var declaracionAsignacion = Instruccion.nuevaDeclaracion(metodoEjecutar.lista_parametros[i].id, _instruccion.lista_valores[i], metodoEjecutar.lista_parametros[i].ti
                    var mensaje = DecParametro(declaracionAsignacion, nuevoAmbito)
                    cadena += mensaje
                }
                var exec = Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
                var mensaje = exec.cadena
                if (exec.existeBreak) {
                    var err = {
                        TipoError: "Semántico",
                        Descripcion: `Se ha encontrado un break fuera de un ciclo`,
                        Linea: _instruccion.linea,
                        Columna: _instruccion.columna
                    }
                    ListaErrores.push(err)
                    mensaje += "Error: Se ha encontrado un break fuera de un ciclo\n"
                }
                if (exec.existeContinue) {
                    var err = {
                        TipoError: "Semántico",
                        Descripcion: `Se ha encontrado un continue fuera de un ciclo`,
                        Linea: _instruccion.linea,
                        Columna: _instruccion.columna
                    }
                    ListaErrores.push(err)
                    mensaje += "Error: Se ha encontrado un continue fuera de un ciclo\n"
                }
                return mensaje
```

- Archivo For.js:

Este archivo contiene una función que es muy parecida a las del while, con la diferencia que ahora la instrucción trae una declaración y una asignación/actualización ya que para el for se tiene que tomar en cuenta estos parámetros, se procede a iterar actualizando al final de cada iteración y verificando otra vez su condición.

```
    var nuevoAmbito = new Ambito(_ambito, "For")
    if (_instruccion.declaracion.tipo === TIPO_INSTRUCCION.DECLARACION){
        error = Declaracion(_instruccion.declaracion, nuevoAmbito)
    }
    else if (_instruccion.declaracion.tipo === TIPO_INSTRUCCION.ASIGNACION){
        //console.log(_instruccion.declaracion);
        error = Asignacion(_instruccion.declaracion, nuevoAmbito)
    } else {
        return {
            cadena: error,
            valor: valor
        }
    }
    //console.log(_instruccion.actualizacion)
    //console.log(nuevoAmbito)
    var condicion = Operacion(_instruccion.condicion, nuevoAmbito)
    mensaje += condicion.mensaje
    //console.log(condicion)
    if(condicion.tipo === TIPO_DATO.BANDERA){
        while(condicion.valor){
            const Bloque = require('./Bloque')
            //mensaje+=Bloque(_instruccion.instrucciones, nuevoAmbito2)
            var exec = Bloque(_instruccion.instrucciones, nuevoAmbito)
            mensaje += exec.cadena
            valor = exec.valor
            if (exec.existeBreak || exec.existeReturn){
                return {
                    cadena: mensaje,
                    valor: valor
                }
            }
            //actualizamos
            actualizacion = Asignacion(_instruccion.actualizacion, nuevoAmbito)
            condicion = Operacion(_instruccion.condicion, nuevoAmbito)
            mensaje += condicion.mensaje
        }
        return {
            cadena: mensaje,
            valor: valor
        }
```

- Archivo If.js y IfElse.js:

Estos archivos contienen una función que recibe una instrucción, operan la condición de esa instrucción y si el valor es verdadero ejecuta su bloque y si no, ejecuta el bloque del else.

```
    var operacion = Operacion(_instruccion.expresion, _ambito)
    mensaje += operacion.mensaje
    if (operacion.tipo === TIPO_DATO.BANDERA){
        if(operacion.valor){
            var nuevoAmbito = new Ambito(_ambito, "If")
            const Bloque = require("./Bloque")
            //mensaje += Bloque(_instruccion.instrucciones, nuevoAmbito)
            var exec = Bloque(_instruccion.instrucciones, nuevoAmbito)
            mensaje += exec.cadena
            existeBreak = exec.existeBreak
            existeContinue = exec.existeContinue
            existeReturn = exec.existeReturn
            valor = exec.valor
        }
        return{
            existeBreak: existeBreak,
            existeContinue: existeContinue,
            existeReturn: existeReturn,
            valor: valor,
            cadena: mensaje
        }
    }
    var err = {
        TipoError: "Semántico",
        Descripcion: `No es una expresión válida para el If`,
        Linea: _instruccion.linea,
        Columna: _instruccion.columna
    }
    ListaErrores.push(err)
```

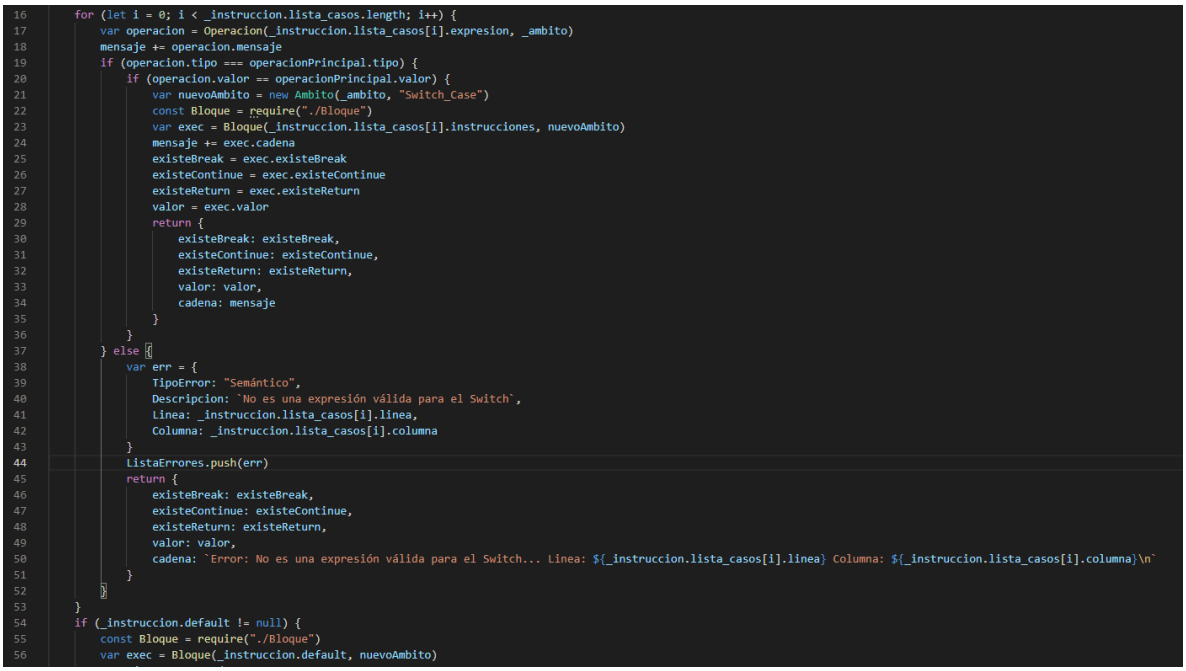
- Archivo IfElseIf.js:

Este archivo a diferencia de los 2 anteriores, posee una lista de else if, que en medio de el if y el else se ejecuta e itera para verificar su condición y si en alguna iteración se cumple la condición, procede a ejecutar su respectivo bloque y omite los demás.

```
15      if (operacion.tipo === TIPO_DATO.BANDERA) {
16          if (operacion.valor) {
17              var nuevoAmbito = new Ambito(_ambito, "If")
18              const Bloque = require("./Bloque")
19              //mensaje += Bloque(_instruccion.instruccionesIf, nuevoAmbito)
20              var exec = Bloque(_instruccion.instruccionesIf, nuevoAmbito)
21              mensaje += exec.cadena
22              existeBreak = exec.existeBreak
23              existeContinue = exec.existeContinue
24              existeReturn = exec.existeReturn
25              valor = exec.valor
26              return {
27                  existeBreak: existeBreak,
28                  existeContinue: existeContinue,
29                  existeReturn: existeReturn,
30                  valor: valor,
31                  cadena: mensaje
32              }
33          }
34          //console.log(_instruccion.lista_elseif);
35          for (let i = 0; i < _instruccion.lista_elseif.length; i++) {
36              var operacion = Operacion(_instruccion.lista_elseif[i].expresion, _ambito)
37              mensaje += operacion.mensaje
38              if (operacion.tipo === TIPO_DATO.BANDERA) {
39                  if (operacion.valor) {
40                      var nuevoAmbito = new Ambito(_ambito, "IfElse")
41                      const Bloque = require("./Bloque")
42                      var exec = Bloque(_instruccion.lista_elseif[i].instruccionesElseIf, nuevoAmbito)
43                      mensaje += exec.cadena
44                      existeBreak = exec.existeBreak
45                      existeContinue = exec.existeContinue
46                      existeReturn = exec.existeReturn
47                      valor = exec.valor
48                      return {
49                          existeBreak: existeBreak,
50                          existeContinue: existeContinue,
51                          existeReturn: existeReturn,
52                          valor: valor,
53                          cadena: mensaje
54                      }
55                  }
```

- Archivo Switch.js:

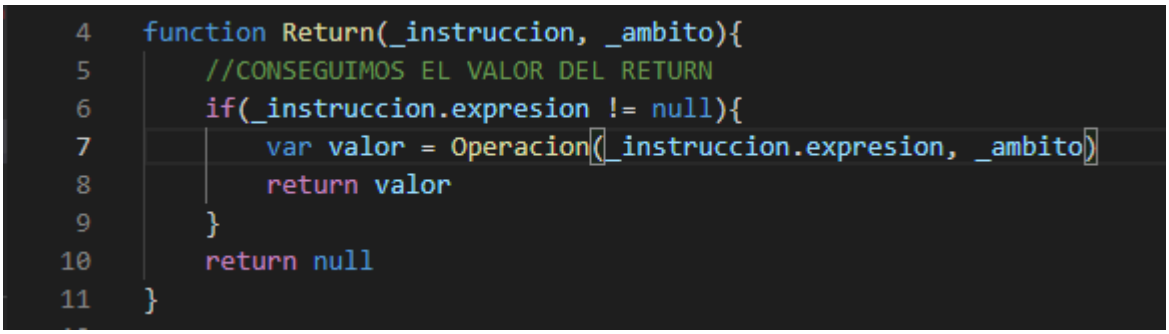
Este archivo es muy similar al del ElseIfElse.js ya que utilizan la misma lógica, pero sin el primer if, ya que directamente itera dentro de la lista de casos entrantes e iguala la expresión a su determinante del switch y si se cumple, se ejecuta el bloque de dicho caso, y si ningún bloque fue ejecutado, procede a verificar la condición del default en caso de que exista y si es verdadera, ejecuta ese bloque.



```
for (let i = 0; i < _instruccion.lista_casos.length; i++) {
    var operacion = Operacion(_instruccion.lista_casos[i].expresion, _ambito)
    mensaje += operacion.mensaje
    if (operacion.tipo === operacionPrincipal.tipo) {
        if (operacion.valor == operacionPrincipal.valor) {
            var nuevoAmbito = new Ambito(_ambito, "Switch_Case")
            const Bloque = require("./Bloque")
            var exec = Bloque(_instruccion.lista_casos[i].instrucciones, nuevoAmbito)
            mensaje += exec.cadena
            existeBreak = exec.existeBreak
            existeContinue = exec.existeContinue
            existeReturn = exec.existeReturn
            valor = exec.valor
            return {
                existeBreak: existeBreak,
                existeContinue: existeContinue,
                existeReturn: existeReturn,
                valor: valor,
                cadena: mensaje
            }
        }
    } else {
        var err = {
            TipoError: "Semántico",
            Descripcion: `No es una expresión válida para el Switch`,
            Linea: _instruccion.lista_casos[i].linea,
            Columna: _instruccion.lista_casos[i].columna
        }
        ListaErrores.push(err)
        return {
            existeBreak: existeBreak,
            existeContinue: existeContinue,
            existeReturn: existeReturn,
            valor: valor,
            cadena: `Error: No es una expresión válida para el Switch... Linea: ${_instruccion.lista_casos[i].linea} Columna: ${_instruccion.lista_casos[i].columna}\n`
        }
    }
}
if (_instruccion.default != null) {
    const Bloque = require("./Bloque")
    var exec = Bloque(_instruccion.default, nuevoAmbito)
```

- Archivo Return.js:

Este archivo es bien sencillo, solo ejecuta la operación dentro de la instrucción, y luego la devuelve, siempre u cuando sea diferente de nulo.



```
function Return(_instruccion, _ambito){
    //CONSEGUIMOS EL VALOR DEL RETURN
    if(_instruccion.expresion != null){
        var valor = Operacion(_instruccion.expresion, _ambito)
        return valor
    }
    return null
}
```

❖ Archivo Operación.js:

Este archivo solo verifica el tipo de la operación y dependiendo de ese, ejecuta ciertas funciones.

```
function Operacion(_expresion, _ambito){
    if(_expresion.tipo === TIPO_VALOR.DECIMAL || _expresion.tipo === TIPO_VALOR.BANDERA ||
       _expresion.tipo === TIPO_VALOR.CADENA || _expresion.tipo === TIPO_VALOR.IDENTIFICADOR ||
       _expresion.tipo === TIPO_VALOR.CARACTER || _expresion.tipo === TIPO_VALOR.DOUBLE){
       return ValorExpresion(_expresion, _ambito)
    }
    else if(_expresion.tipo === TIPO_OPERACION.SUMA || _expresion.tipo === TIPO_OPERACION.RESTA ||
       _expresion.tipo === TIPO_OPERACION.MULT || _expresion.tipo === TIPO_OPERACION.DIV ||
       _expresion.tipo === TIPO_OPERACION.EXP || _expresion.tipo === TIPO_OPERACION.MOD ||
       _expresion.tipo === TIPO_OPERACION.MEN || _expresion.tipo === TIPO_OPERACION.MASMAS ||
       _expresion.tipo === TIPO_OPERACION.MENOSMENOS){
       return Aritmetica(_expresion, _ambito)
    }
    else if(_expresion.tipo === TIPO_OPERACION.IGUALIGUAL || _expresion.tipo === TIPO_OPERACION.DIFERENTE ||
       _expresion.tipo === TIPO_OPERACION.MENOR || _expresion.tipo === TIPO_OPERACION.MAYOR ||
       _expresion.tipo === TIPO_OPERACION.MENORIGUAL || _expresion.tipo === TIPO_OPERACION.MAYORIGUAL){
       return Relacional(_expresion, _ambito)
    }
    else if(_expresion.tipo === TIPO_OPERACION.OR || _expresion.tipo === TIPO_OPERACION.AND ||
       _expresion.tipo === TIPO_OPERACION.NOT){
       return Logica(_expresion, _ambito)
    }
    else if(_expresion.tipo === TIPO_OPERACION.CAST){
       return Castear(_expresion, _ambito)
    }
    else if(_expresion.tipo === TIPO_OPERACION.LLAMADA){
       return OpLlamada(_expresion, _ambito)
    }
}

module.exports = Operacion
```

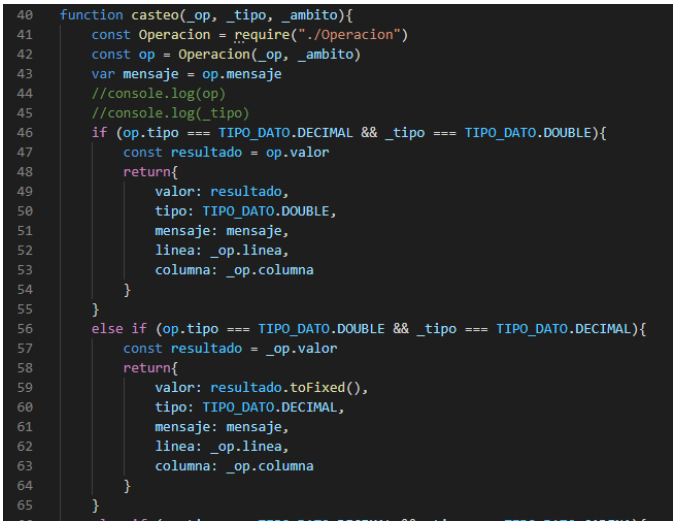
❖ Archivo Aritmética.js:

Este verifica el tipo de la operación que esta recibiendo y dependiendo de ese tipo, manda a llamar a su respectiva función, dentro de cada función se operan su operador izquierdo y derecho primero y luego se procede a operar los valores de las operaciones.

```
240  function div(_opIzq, _opDer, _ambito){
241      const opIzq = Aritmetica(_opIzq,_ambito)
242      const opDer = Aritmetica(_opDer,_ambito)
243      const tipoRes = TipoResultado(opIzq.tipo, opDer.tipo)
244      var mensaje = ""
245      mensaje += opIzq.mensaje
246      mensaje += opDer.mensaje
247      if(tipoRes!=null){
248          if(tipoRes!=null){
249              if(tipoRes === TIPO_DATO.DECIMAL || tipoRes === TIPO_DATO.DOUBLE){
250                  if (opIzq.tipo == TIPO_DATO.CARACTER){
251                      const resultado = (opIzq.valor.toString()).charCodeAt() / Number(opDer.valor);
252                      return{
253                          valor: resultado,
254                          tipo: TIPO_DATO.DOUBLE,
255                          mensaje: mensaje,
256                          linea: _opIzq.linea,
257                          columna: _opIzq.columna
258                      }
259                  } else if (opDer.tipo == TIPO_DATO.CARACTER) {
260                      const resultado = Number(opIzq.valor) / (opDer.valor.toString()).charCodeAt();
261                      return{
262                          valor: resultado,
263                          tipo: TIPO_DATO.DOUBLE,
264                          mensaje: mensaje,
265                          linea: _opIzq.linea,
266                          columna: _opIzq.columna
267                      }
268                  } else {
269                      const resultado = Number(opIzq.valor) / Number(opDer.valor);
270                      return{
271                          valor: resultado,
272                          tipo: TIPO_DATO.DOUBLE,
273                          mensaje: mensaje,
274                          linea: _opIzq.linea,
275                          columna: _opIzq.columna
276                      }
277                  }
278              }
279          }
280      }
```

❖ Archivo Casteo.js:

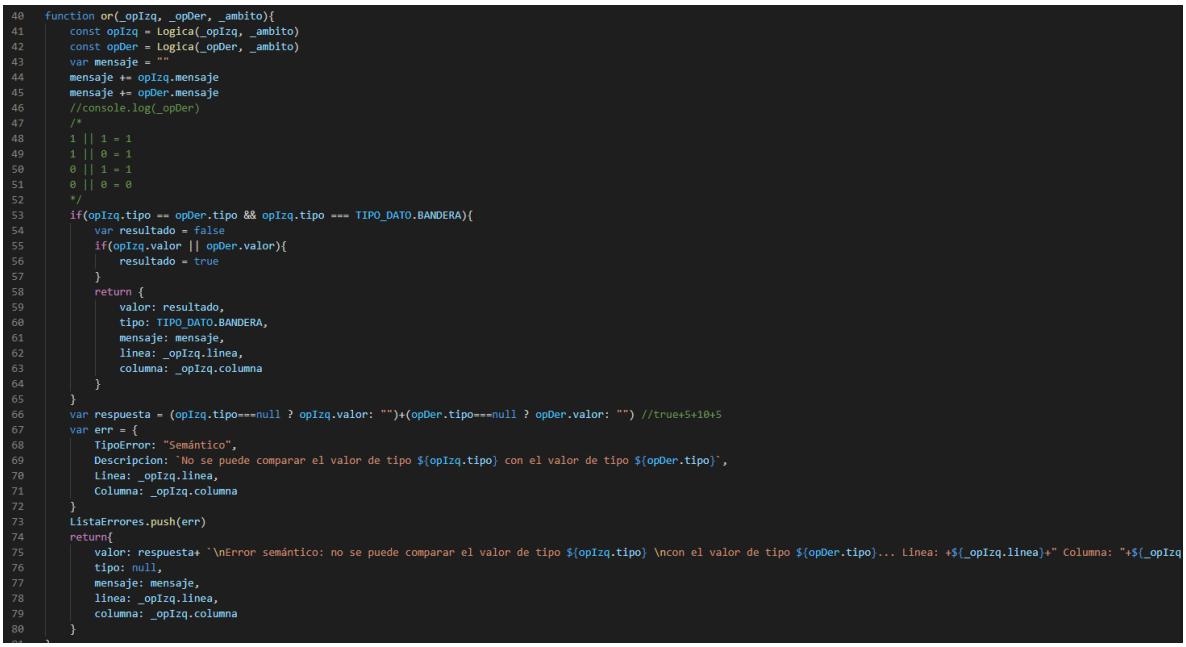
Este archivo recibe una operación de casteo, verifica el tipo de la expresión que recibe, y el tipo del casteo, luego de eso procede a castear el valor de la operación y la devuelve, en caso de no coincidir en ninguna combinación, se retorna un error.



```
function casteo(_op, _tipo, _ambito){
    const Operacion = require("./Operacion")
    const op = Operacion(_op, _ambito)
    var mensaje = op.mensaje
    //console.log(op)
    //console.log(_tipo)
    if (op.tipo === TIPO_DATO.DECIMAL && _tipo === TIPO_DATO.DOUBLE){
        const resultado = op.valor
        return{
            valor: resultado,
            tipo: TIPO_DATO.DOUBLE,
            mensaje: mensaje,
            linea: _op.linea,
            columna: _op.columna
        }
    }
    else if (op.tipo === TIPO_DATO.DOUBLE && _tipo === TIPO_DATO.DECIMAL){
        const resultado = _op.valor
        return{
            valor: resultado.toFixed(),
            tipo: TIPO_DATO.DECIMAL,
            mensaje: mensaje,
            linea: _op.linea,
            columna: _op.columna
        }
    }
```

❖ Archivo Lógica.js:

Este archivo es muy similar al de aritmética pero con la diferencia de que verifica que ambos operadores deben de ser de tipo booleano y luego dependiendo de la operación, retornara el nuevo valor booleano.



```
function or(_opIzq, _opDer, _ambito){
    const opIzq = Logica(_opIzq, _ambito)
    const opDer = Logica(_opDer, _ambito)
    var mensaje = ""
    mensaje += opIzq.mensaje
    mensaje += opDer.mensaje
    //console.log(_opDer)
    /*
    1 || 1 = 1
    1 || 0 = 1
    0 || 1 = 1
    0 || 0 = 0
    */
    if(opIzq.tipo == opDer.tipo && opIzq.tipo === TIPO_DATO.BANDERA){
        var resultado = false
        if(opIzq.valor || opDer.valor){
            resultado = true
        }
        return {
            valor: resultado,
            tipo: TIPO_DATO.BANDERA,
            mensaje: mensaje,
            linea: _opIzq.linea,
            columna: _opIzq.columna
        }
    }
    var respuesta = (opIzq.tipo===null ? opIzq.valor: "")+(opDer.tipo===null ? opDer.valor: "") //true+5+10+5
    var err = {
        TipoError: "Semántico",
        Descripcion: `No se puede comparar el valor de tipo ${opIzq.tipo} con el valor de tipo ${opDer.tipo}`,
        Linea: _opIzq.linea,
        Columna: _opIzq.columna
    }
    ListaErrores.push(err)
    return{
        valor: respuesta+ `\nError semántico: no se puede comparar el valor de tipo ${opIzq.tipo} \ncon el valor de tipo ${opDer.tipo}... Linea: +${_opIzq.linea}+" Columna: "+${_opIzq
        tipo: null,
        mensaje: mensaje,
        linea: _opIzq.linea,
        columna: _opIzq.columna
    }
```

- Archivo OpLlamada.js:

Este archivo es muy parecido al de exec.js pero con la diferencia de que ahora retornara el valor que retorne el bloque a la hora de ejecutarlo, además de eso, si no se llegase a encontrar la función dentro del ámbito, procede a revisar si el nombre de la función es alguna de las funciones nativas del lenguaje, y si se da el caso ejecuta cada función dependiendo de esta, de lo contrario retornara un valor y un error.

```
    if (metodoEjecutar != null) {
        var nuevoAmbito = new Ambito(_ambito, "Funcion_" + metodoEjecutar.id)
        //Si trae parametros
        if (metodoEjecutar.lista_parametros != null) {
            //Si tienen la misma cantidad
            if (lista_valores != null && metodoEjecutar.lista_parametros.length == lista_valores.length) {
                var error = false;
                for (let i = 0; i < metodoEjecutar.lista_parametros.length; i++) {
                    var declaracionAsignacion = Instruccion.nuevaDeclaracion(metodoEjecutar.lista_parametros[i].id, lista_valores[i], metodoEjecutar.lista_parametros[i].tipo_dato,
                    const DecParametro = require("../Instruccion/DecParametro")
                    var mensaje = DecParametro(declaracionAsignacion, nuevoAmbito)
                    cadena += mensaje
                }
                //console.log(_ambito);
                //return Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
                const Bloque = require("../Instruccion/Bloque")
                var exec = Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
                valor = exec.valor
                cadena += exec.cadena
                valor.mensaje += cadena
                return valor
            }
        }
        const Bloque = require("../Instruccion/Bloque")
        var exec = Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
        valor = exec.valor
        cadena += exec.cadena
        valor.mensaje += cadena
        return valor
    }
```

```
    else if((_nombre.toString()).toLowerCase()=="tolower"){
        const Operacion = require("./Operacion")
        var op = Operacion(lista_valores[0], _ambito)
        if(op.tipo === TIPO_DATO.CADENA){
            op.valor = op.valor.toLowerCase()
            return op
        }
        var err = {
            TipoError: "Semántico",
            Descripcion: 'No se puede realizar la funcion ToLower',
            Linea: _linea,
            Columna: _columna
        }
        ListaErrores.push(err)
        return{
            tipo: null,
            valor: 'Error semántico: No se puede realizar la funcion ToLower, Linea: '+_linea+" Columna: "+_columna + "\n",
            linea: _linea,
            mensaje: 'Error semántico: No se puede realizar la funcion ToLower, Linea: '+_linea+" Columna: "+_columna +"\n",
            columna: _columna
        }
    }
    else if((_nombre.toString()).toLowerCase()=="toupper"){
        const Operacion = require("./Operacion")
        var op = Operacion(lista_valores[0], _ambito)
        if(op.tipo === TIPO_DATO.CADENA){
            op.valor = op.valor.toUpperCase()
            return op
        }
        var err = {
            TipoError: "Semántico",
            Descripcion: 'No se puede realizar la funcion ToUpper',
            Linea: _linea,
            Columna: _columna
        }
        ListaErrores.push(err)
        return{
            tipo: null,
            valor: 'Error semántico: No se puede realizar la funcion ToUpper, Linea: '+_linea+" Columna: "+_columna + "\n",
            linea: _linea,
            mensaje: 'Error semántico: No se puede realizar la funcion ToUpper, Linea: '+_linea+" Columna: "+_columna +"\n",
            columna: _columna
        }
    }
```

- Archivo ProcesarCadena.js:

Este archivo procede a devolver la operación de la expresión que recibe.

```
const Operacion = require("./Operacion")

function procesarCadena(_expresion, _ambito){
    return Operacion(_expresion, _ambito)
}

module.exports = procesarCadena
```

- Archivo Relacional.js:

Este archivo es igual muy similar al de Aritmética.js o al de Lógica.js solo que ahora compara el valor del operador izquierdo y el valor del operador derecho respecto a las definiciones de las operaciones y obviamente validando el tipo del valor en caso de ser necesario.

```
function igualigual(_opIzq, _opDer, _ambito){
    const opIzq = Relacional(_opIzq, _ambito)
    const opDer = Relacional(_opDer, _ambito)
    var mensaje = ""
    mensaje += opIzq.mensaje
    mensaje += opDer.mensaje
    if(opIzq.tipo == opDer.tipo){ //1==1 true==false ...
        var resultado = false
        if(opIzq.valor == opDer.valor){
            resultado = true
        }
        return {
            valor: resultado,
            tipo: TIPO_DATO.BANDERA,
            mensaje: mensaje,
            linea: _opIzq.linea,
            columna: _opIzq.columna
        }
    }
    var respuesta = (opIzq.tipo===null ? opIzq.valor: "")+(opDer.tipo===null ? opDer.valor: "") //true+5+10+5
    var err = {
        TipoError: "Semántico",
        Descripcion: `No se puede comparar el valor de tipo ${opIzq.tipo} con el valor de tipo ${opDer.tipo}`,
        Linea: _opIzq.linea,
        Columna: _opIzq.columna
    }
    ListaErrores.push(err)
    return{
        valor: respuesta+ `\nError semántico: no se puede comparar el valor de tipo ${opIzq.tipo} \ncon el valor de tipo ${opDer.tipo}... Linea: +${_opIzq.linea}+" Columna: "+${_opIzq.
        tipo: null,
        mensaje: mensaje,
        linea: _opIzq.linea,
        columna: _opIzq.columna
    }
}
```

- Archivo TipoResultado.js:

Este archivo solo verifica el tipo del operador izquierdo y el derecho y dependiendo de eso retorna un nuevo tipo.

```
function TipoResultado(_tipo1, _tipo2){
    if(_tipo1 === TIPO_DATO.DECIMAL && _tipo2 === TIPO_DATO.DECIMAL){ return TIPO_DATO.DECIMAL}
    else if ((_tipo1 === TIPO_DATO.CADENA || _tipo2 === TIPO_DATO.CADENA) && _tipo1!==null && _tipo2!==null) return TIPO_DATO.CADENA
    else if (_tipo1 === TIPO_DATO.BANDERA && _tipo2 === TIPO_DATO.BANDERA) return TIPO_DATO.BANDERA
    else if (_tipo1 === TIPO_DATO.CARACTER && _tipo2 === TIPO_DATO.CARACTER) return TIPO_DATO.CADENA
    else if (_tipo1 === TIPO_DATO.DOUBLE && _tipo2 === TIPO_DATO.DOUBLE) return TIPO_DATO.DOUBLE
    else if ((_tipo1 === TIPO_DATO.DOUBLE && _tipo2 === TIPO_DATO.DECIMAL) ||(_tipo1 === TIPO_DATO.DECIMAL && _tipo2 === TIPO_DATO.DOUBLE)) return TIPO_DATO.DOUBLE
    else if ((_tipo1 === TIPO_DATO.DECIMAL && _tipo2 === TIPO_DATO.BANDERA) ||(_tipo1 === TIPO_DATO.BANDERA && _tipo2 === TIPO_DATO.DECIMAL)) return TIPO_DATO.DECIMAL
    else if ((_tipo1 === TIPO_DATO.DECIMAL && _tipo2 === TIPO_DATO.CARACTER) ||(_tipo1 === TIPO_DATO.CARACTER && _tipo2 === TIPO_DATO.DECIMAL)) return TIPO_DATO.DECIMAL
    else if ((_tipo1 === TIPO_DATO.DOUBLE && _tipo2 === TIPO_DATO.CARACTER) ||(_tipo1 === TIPO_DATO.CARACTER && _tipo2 === TIPO_DATO.DOUBLE)) return TIPO_DATO.DOUBLE
    else if ((_tipo1 === TIPO_DATO.CARACTER && _tipo2 === TIPO_DATO.CADENA) ||(_tipo1 === TIPO_DATO.CADENA && _tipo2 === TIPO_DATO.CARACTER)) return TIPO_DATO.CADENA
    return null
}
```

- Archivo ValorExpresion.js:

Este archivo solo verifica el tipo del valor que entra y dependiendo de eso solo retorna un objeto del operador.

```
function ValorExpresion(_expresion, _ambito){
    if(_expresion.tipo === TIPO_VALOR.DECIMAL){
        return {
            valor: Number(_expresion.valor),
            tipo: TIPO_DATO.DECIMAL,
            mensaje: "",
            linea: _expresion.linea,
            columna: _expresion.columna
        }
    }
    else if(_expresion.tipo === TIPO_VALOR.DOUBLE){
        return {
            valor: Number(_expresion.valor),
            tipo: TIPO_DATO.DOUBLE,
            mensaje: "",
            linea: _expresion.linea,
            columna: _expresion.columna
        }
    }
    else if(_expresion.tipo === TIPO_VALOR.BANDERA){
        return {
            valor: _expresion.valor.toLowerCase()==='true' ? true: false,
            tipo: TIPO_DATO.BANDERA,
            mensaje: "",
            linea: _expresion.linea,
            columna: _expresion.columna
        }
    }
    else if(_expresion.tipo === TIPO_VALOR.CADENA){
        return {
            valor: _expresion.valor.substring(1, _expresion.valor.length-1),
            tipo: TIPO_DATO.CADENA,
            mensaje: "",
            linea: _expresion.linea,
            columna: _expresion.columna
        }
    }
    else if(_expresion.tipo === TIPO_VALOR.CARACTER){
        return {
            valor: _expresion.valor.substring(1, _expresion.valor.length-1),
            tipo: TIPO_DATO.CARACTER,
            mensaje: "",
            linea: _expresion.linea,
            columna: _expresion.columna
        }
```

Archivo Graficador.js:

Este archivo recorre el árbol de la misma manera que el Bloque.js lo hace, además también ejecuta ciertas funciones dependiendo del tipo de instrucción que sea la expresión, solo que ahora la diferencia esta en que usará todas las propiedades de los objetos para concatenar cadenas del lenguaje Dot para así poder graficar el árbol.

```
            else if (instruccion.tipo === TIPO_INSTRUCCION.IF_ELSE_IF) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="IF_ELSE_IF"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
                this.graficarIfElseIf(instruccion, nombreHijo)
            }
            else if (instruccion.tipo === TIPO_INSTRUCCION.SWITCH) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="SWITCH"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
                this.graficarSwitch(instruccion, nombreHijo)
            }
            else if (instruccion.tipo === TIPO_INSTRUCCION.LLAMADA) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="LLAMADA"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
                this.graficarExec(instruccion, nombreHijo)
            }
            else if (instruccion.tipo === TIPO_INSTRUCCION.BREAK) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="BREAK"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
            }
            else if (instruccion.tipo === TIPO_INSTRUCCION.CONTINUE) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="CONTINUE"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
            }
            else if (instruccion.tipo === TIPO_INSTRUCCION.RETURN) {
                var nombreHijo = "Nodo" + this.contador
                this.contador++
                this.grafo += nombreHijo + '[label="RETURN"];\n'
                this.grafo += _nombrePadre + "->" + nombreHijo + ';\n'
                this.graficarPrint(instruccion, nombreHijo)
            }
        })
    }
```

```
    graficarExec(_instruccion, _padre) {
        var nombreMet = `Nodo${this.contador}`
        this.grafo += nombreMet + `[label="METODO/FUNCION \\n ${_instruccion.nombre}"];\n`
        this.grafo += _padre + "->" + nombreMet + ";\n"
        this.contador++
        if (_instruccion.lista_valores != null) {
            var nombreLista = `Nodo${this.contador}`
            this.grafo += nombreLista + `[label="LISTA_VALORES"];\n`
            this.grafo += _padre + "->" + nombreLista + ";\n"
            this.contador++
            for (let i = 0; i < _instruccion.lista_valores.length; i++) {
                this.graficarOperacion(_instruccion.lista_valores[i], nombreLista)
            }
        }
    }

    graficarDeclaracion(_instruccion, _padre) {
        var tipoVar = `Nodo${this.contador}`
        this.grafo += tipoVar + `[label="TIPO \\n ${_instruccion.tipo_dato}"];\n`
        this.grafo += _padre + "->" + tipoVar + ";\n"
        this.contador++
        var nombreVar = `Nodo${this.contador}`
        this.grafo += nombreVar + `[label="Identificador \\n ${_instruccion.id}"];\n`
        this.grafo += _padre + "->" + nombreVar + ";\n"
        this.contador++
        if (_instruccion.valor != null) {
            this.graficarOperacion(_instruccion.valor, _padre)
        }
    }
```