CompScript

Manual Técnico

CompScript

Lenguaje Utilizado: JavaScript / TypeScript
Requisitos: Angular CLI, Node.jjS
Sistema Operativo: Windows 7 en adelante

Resumen

CompScript es una analizador que interpreta una serie de instrucciones para un lenguaje de programación simple que utiliza las sentencias más basicas usadas en cualquier otro lenguaje. La aplicación fuciona en el servidor con ayuda de Node.js.

Explicación del funcionamiento

Como se menciono antes la aplicación funciona del lado del servidor. El front envia una peticion con el texto que venga del editor, el servidor lo recibe para procesarlo. El proceso que ejecuta el srvidor se divide en tres: primero pasa por el analizador de encargado de separar en instrucciones el codigo que venga de la entrada. Por motivos de requisitos, se diseño otro analizador que analiza la entrada pero que se encarga de generar un arbol para que pueda ser graficado y mostrado.

Con ambas salidas, las instrucciones entran en un proceso donde son analizadas y ejecutadas una por una, devolviendo una cadena con los mensajes a mostrar en la consola. Para la parte del ast, se maneja otro proceso donde se lee cada nodo de manera recursiva y se va añadiendo a una cadena las declaraciones y conexiones de cada nodo.

```
app.post('/analizar',(req,res)=>{
    let parser = require('./gramatica');
    let parse = require('./arbol');
    var entrada = req.body.entrada;
    var arbol = parser.parse(entrada)
    var a = parse.parse(entrada)
    //Extraer Valores
    var errores = arbol.lerrores
    var simbolo = arbol.lsimbolos;
    var metodos = arbol.lmetodos
    var ast = a.arbol
    var graphviz = Codigo(ast)
    console.log("----------------------ArbOL--------------------------")
    console.log(graphviz)
    console.log("-----------------------------------------------------")
    //Ejecucuon de Instrucciones
    const global = new Entorno(null, "GLOBAL")
    console.log("INICIANDO ANALISIS")
    let recorrido = Iniciar(arbol.instrucciones , global, errores, simbolo, metodos)
    console.log("ANALISIS FINALIZADO")
    //Salida
    var respuesta={
        message:"Resultado correcto",
        ast: "Hola",
        ast: graphviz,        //Retorna el arbol. Debe de graficarse.
        salida: recorrido,  //Retorna el texto de la consola
        errores: errores,    //Retorna la lista de errores
        simbolos: simbolo,   //Retorna la tabla de simbolos a mostrar
        metodos: metodos    //Retorna los metodos de la ejecucion
    }
    res.send(respuesta)
})
```

Gramatica.jison

Como se menciono antes, el archivo de gramatica se encarga de ir leyendo el archivo y replicar el formato de instrucción que maneja un lenguaje de programacion. Auxiliado de la clase instrucciones, confoem va encontrando coincidencias va creando objetos de tipo instrucción que van subiendo conforme avanza el analizador. Algo a mencionar es que las instrucciones sirven como contenedor y es frecuente ver que cosas como la declaracion de una funcion es una instrucción que contiene un listado de instrucciones dentro; o una suma que nos oslo contiene una instrucción indicando lo que se va a hacer, sino tambien, contiene instrucciones dentro que permiten manejar alguna operación anidada o la busqueda del valor de un identificador.

```
//Declarar Metodos-------------------------------------------------------------------------
DMETODO: identificador parA parC llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dmetodo($1, null, $5, this._$.first_line, this._$.first_column+1)
        }
        |identificador parA parC dospuntos void llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dmetodo($1, null, $7, this._$.first_line, this._$.first_column+1)
        }
        | identificador parA PARAMETROS parC llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dmetodo($1, $3, $6, this._$.first_line, this._$.first_column+1)
        }
        |identificador parA PARAMETROS parC dospuntos void llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dmetodo($1, $3, $8, this._$.first_line, this._$.first_column+1)
        }
;

//Declarar Funciones-----------------------------------------------------------------------
DFUNCION: identificador parA parC dospuntos TIPO llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dfuncion($1, null, $5, $7, this._$.first_line, this._$.first_column+1)
        }
        |identificador parA PARAMETROS parC dospuntos TIPO llavA INSTRUCCIONES llavC
        {
                $$ = INSTRUCCION.dfuncion($1, $3, $6, $8, this._$.first_line, this._$.first_column+1)
        }
;
```

arbol.jison

Aquí es donde se genera el arbol como estructura. Se tenia pensado en un inicio manejar las instrucciones y el arbol en uno solo pero resulto ser mucho mas complejo. A pesar de tener implementado el uso de las uinstrucciones, no es muy recomendado intentar extraer de ahí la informacion ya que hay ciertos bugs por depurar aun y se hacia muy engorroso ir revisando que no existieran errores. Los nodos vienen de la clase noodo y simplemente se van creando instancias y añadiendo a cada uno los apuntadores a a los hijos respectivos.

```
286
287 DVARIABLES: TIPO LISTAID asignar EXPRESION puntocoma
288         {···
302         }
303         | TIPO LISTAID puntocoma
304         {
305                 //Obtencion de valores
306                 entrada1 = $1
307                 entrada2 = $2
308                 //Creacion de la salida
309                 instruccion = INSTRUCCION.declaracionv(entrada1.instruccion, entrada2.instruccion, null, th
310                 nodo = new Nodo("DEC VARIABLES", "DEC VARIABLES")
311                 nodo.add(entrada1.nodo, entrada2.nodo ,new Nodo("OPERADOR", $3))
312                 salida = {
313                         nodo: nodo,
314                         instruccion: instruccion
315                 }
316                 $$ = salida
317         }
318 ;
319
320 LISTAID:  LISTAID coma identificador
321         {···
334         }
335
336         | identificador
337         {
338                 //Creacion de la salida
339                 instruccion = [$1]
340                 nodo = new Nodo("LISTA ID", "LISTA ID")
341                 nodo.add(new Nodo("ID", $1))
342                 salida = {
```

En la carpeta recursos se encuentran los archivos que sirven para manejar cada uno de los apartados anteriormente mencionados, separados por su respectiva tarea.

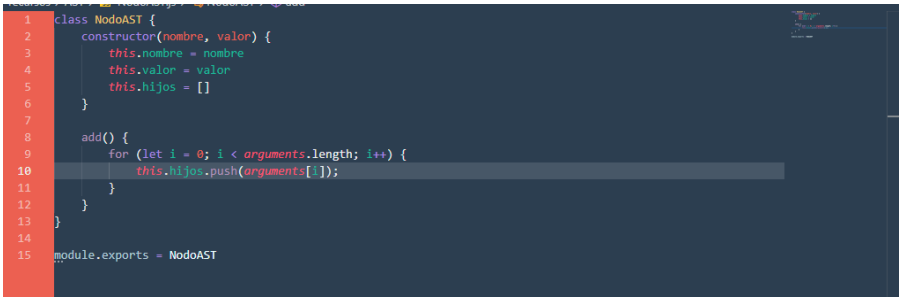
Graphviz.js

Recibe el arbol generado por el analizador y se encarga de convertirlo a xintaxis de tipo dot. Para ello, maneja recursivamente el recorrido, recibiendo un nodo y añadiendo a la cadena su informacion. Luego se dedica a revsar la lista de hijos que cornforman ese nodo y manda a llamar al metodo otra vez con su lista de hijos hasta que el programa detecte que esa lista esta vacia y empiece a hacer returns.

```
1  //ANALIZADOR DEL AST PARA GENERAR CODIGO DE GRAPHVIZ
2  function Graphviz(raiz){
3      let contador = 0
4      var salida = "digraph G{" + "\n"
5      salida += "Raiz[label=\"" + raiz.valor + "\"];\n"
6      conectar("Raiz", raiz)
7      salida += "}";
8      return salida;
9
10
11
12
13     function conectar(nombrePadre, padre){
14         if(padre === undefined || padre === null || padre.hijos.length === 0) {
15             return
16         }else{
17             //console.log(padre.hijos)
18             padre.hijos.forEach(
19                 temp=> {
20                     if(temp == undefined){
21
22                     }else{
23                         let hijo = "N" + contador
24                         salida += hijo +"[label=\"" + temp.valor + "\"];\n";
25                         salida += nombrePadre + "->" + hijo +";\n";
26                         contador++
27                         conectar(hijo, temp);
28                     }
29                 }
30             )
31         }
32     }
```

NodoAST.js

Es una clase que crea los nodos AST del arbol. No maneja algun metodo o funcion especial.



```
class NodoAST {
    constructor(nombre, valor) {
        this.nombre = nombre
        this.valor = valor
        this.hijos = []
    }

    add() {
        for (let i = 0; i < arguments.length; i++) {
            this.hijos.push(arguments[i]);
        }
    }
}

module.exports = NodoAST
```

Entorno.js

La clase entorno, como su nombre indica, sirve para manejar el contenido y recursos necesarios para la ejecucion de instrucciones. El nombre se usa para auxiliar el reporte de simbolos, anterior es el padre y la conexión que permite entre entornos(cada que se crea uno nuevo se conecta al entorno donde fue invocado), la tabla de simbolos almacena las variables y la de metodos los declarados aunque solo en el global tiene utilidad. El retorno fue creado para el manejo de funciones. Si un entorno tiene retorno, significa que al mandar un return, tiene que concordar con el tipo. Esto es para evitar hacer returns en metodos. Ambas tablas que manejan los simbolos y los metodos son tablas hash, eso para que se facilite la busqueda de la informacion.

Entre los metodos, se encuentras los metodos para agregar, los de busqueda con retorno y los que no tienen retorno. Los metodos para agregar solo agregan el objeto y usan el nombre de la variable como el hash. Los de busqueda se dividen en dos: los locales y los globales. Para declaraciones de nuevos entornos debe de poderse añadir variables sin importar si existen en niveles superiores, por eso existe el metodo buscarSimbolo que solo revisa el entorno local. Para casos donde se necesiten hacer llamadas de valores pueden venir util el metodo buscarSimboloGlobal que revisa todos los entornos conectados en busca de alguna variable que culpa con la coincidencia. El metodo actualizar tambien busca en todos los entornos alguna coincidencia.

```
class Entorno {
    constructor(padre, nombre) {
        this.nombre = nombre
        this.anterior = padre
        this.tablaSimbolos = new Map()
        this.tablaMetodos = new Map()
        this.retorno = ""

    }

    //Métodos para añadir----------------------------------------------------------------------------
    addSimbolo(nombre, simbolo) {
        this.tablaSimbolos.set(nombre.toLowerCase(), simbolo)
    }

    addMetodo(nombre, metodo) {
        this.tablaMetodos.set(nombre.toLowerCase(), metodo)
    }

    //Métodos de busqueda----------------------------------------------------------------------------

    //Busca el simvbolo en el entorno actual. Si no lo encuentra, busca en el padre.
    //Esto se repite hasta que llega a null
    getSimbolo(nombre) {
        for (let entorno = this; entorno != null; entorno = entorno.anterior) {
            var resultado = entorno.tablaSimbolos.get(nombre.toLowerCase())
            if (resultado != null) {
                return resultado
            }
        }
        return null
    }
```

```
    actualizar(nombre, simbolo) {
        for (let entorno = this; entorno != null; entorno = entorno.anterior) {
            var encontrado = entorno.tablaSimbolos.get(nombre.toLowerCase())
            if (encontrado) {
                entorno.tablaSimbolos.set(nombre.toLowerCase(), simbolo)
                return true
            }
        }
        return false
    }

    //Buscar ----------------------------------------------------------------------------------------
    //Determina si el simbolo existe en todos los niveles
    buscarSimboloGlobal(nombre) {
        for (let entorno = this; entorno != null; entorno = entorno.anterior) {
            var resultado = entorno.tablaSimbolos.get(nombre.toLowerCase())
            if (resultado != null) {
                return true
            }
        }
        return false
    }

    buscarSimbolo(nombre) {
        var resultado = this.tablaSimbolos.get(nombre.toLowerCase())
        if (resultado != null) {
            return true
        }
        return false
    }
```

ListaMetodo.js

Como su nombre indica, dentro se van guardando los metodos. No maneja nada especial mas que servir en el reporte.

```
//Imports
var RMetodo = require("./RMetodo");

//Constructor
class ListaMetodo {
    constructor() {
        this.lista = new Array();
    }

    add(nombre, tipo, linea, columna){
        var nuevo = new RMetodo(nombre, tipo, linea, columna);
        this.lista.push(nuevo);
    }


}

module.exports = ListaMetodo;
```

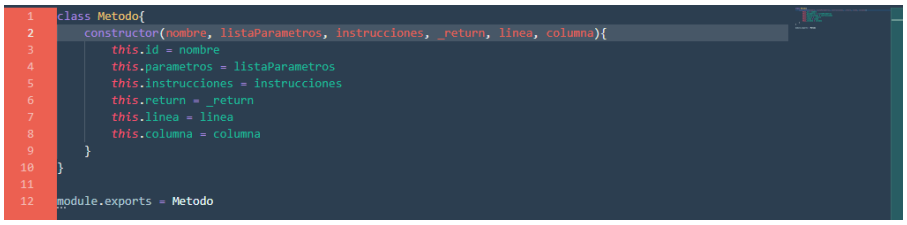
ListaSImbolos.js

Sirve en el reporte de simboloes. Se va moviendo en casi toda la ejecucion para que se pueda modificar o crear valores.

```
//Imports
var RSimbolo = require("./RSimbolo");

//Constructor
class ListaSimbolo {
    constructor() {
        this.lista = new Array();
     }

    add(nombre, contenido, tipo, entorno, linea, columna){
        var nuevo = new RSimbolo(nombre, contenido, tipo, entorno, linea, columna);
        this.lista.push(nuevo);
    }

    update(nombre, entorno, nuevo){
        let indice = this.lista.findIndex((obj => obj.id == nombre && obj.entorno == entorno))
        this.lista[indice].valor = nuevo
    }


}

module.exports = ListaSimbolo;
```

Metodo.js

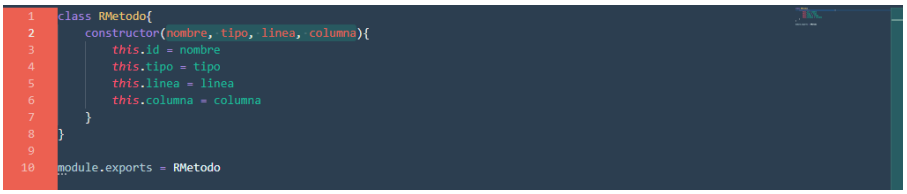
Clase que sirve para declarar objetos de clase metodo.



```
class Metodo{
    constructor(nombre, listaParametros, instrucciones, _return, linea, columna){
        this.id = nombre
        this.parametros = listaParametros
        this.instrucciones = instrucciones
        this.return = _return
        this.linea = linea
        this.columna = columna
    }
}

module.exports = Metodo
```

RMetodo.js

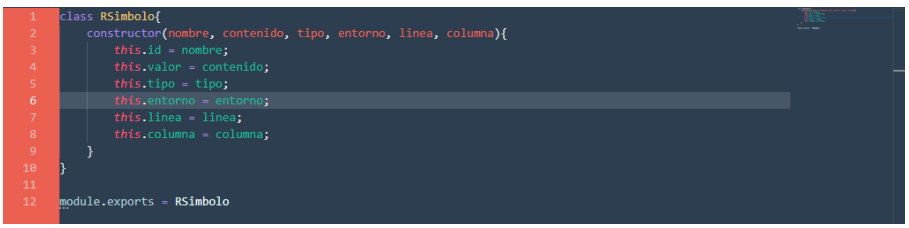
Declara objetos de tipo metodo para el reporte de metodos.



```
class RMetodo{
    constructor(nombre, tipo, linea, columna){
        this.id = nombre
        this.tipo = tipo
        this.linea = linea
        this.columna = columna
    }
}

module.exports = RMetodo
```

RSimbolo.js

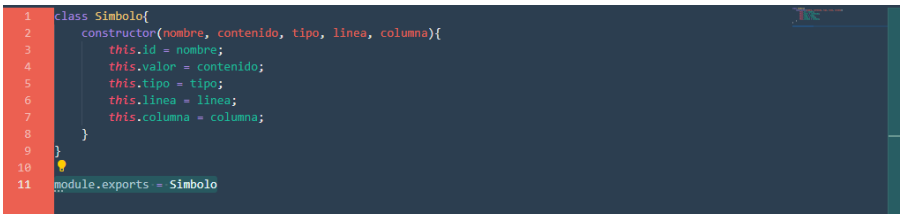
Declara objetos tipo simbolo para el reporte de simbolos.



```
class RSimbolo{
    constructor(nombre, contenido, tipo, entorno, linea, columna){
        this.id = nombre;
        this.valor = contenido;
        this.tipo = tipo;
        this.entorno = entorno;
        this.linea = linea;
        this.columna = columna;
    }
}

module.exports = RSimbolo
```

Simbolo:

Declara objetos de tipo simbolo.



```
class Simbolo{
    constructor(nombre, contenido, tipo, linea, columna){
        this.id = nombre;
        this.valor = contenido;
        this.tipo = tipo;
        this.linea = linea;
        this.columna = columna;
    }
}

module.exports = Simbolo
```

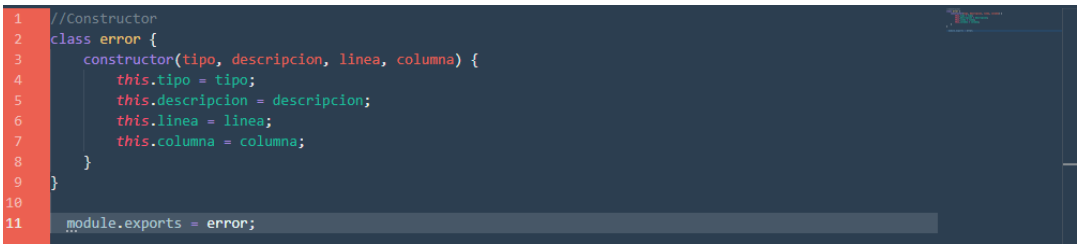
Enums

Dentro de la carpeta enums estan los enums para los fistintos de instrucciones que se manejan en el lenguaje. Se hace uso de estos para que en la escritura sea facil saber a que se esta refiriendo y en que contexto. Son solo un objeto con Strings e identificadores.

```
//Valores que se pueden asignar
const TIPO_VALOR = {
    INT:            'VAL_INT',
    DOUBLE:         'VAL_DOUBLE',
    STRING:         'VAL_STRING',
    BOOLEAN:        'VAL_BOOLEAN',
    IDENTIFICADOR:  'VAL_IDENTIFICADOR',
    VECTORU:        'VAL_VECTORU',
    CHAR:           'VAL_CHAR'
}

module.exports = TIPO_VALOR
```

Error.js

Sirve para declarar objetos de tipo error.



```
//Constructor
class error {
    constructor(tipo, descripcion, linea, columna) {
        this.tipo = tipo;
        this.descripcion = descripcion;
        this.linea = linea;
        this.columna = columna;
    }
}

module.exports = error;
```

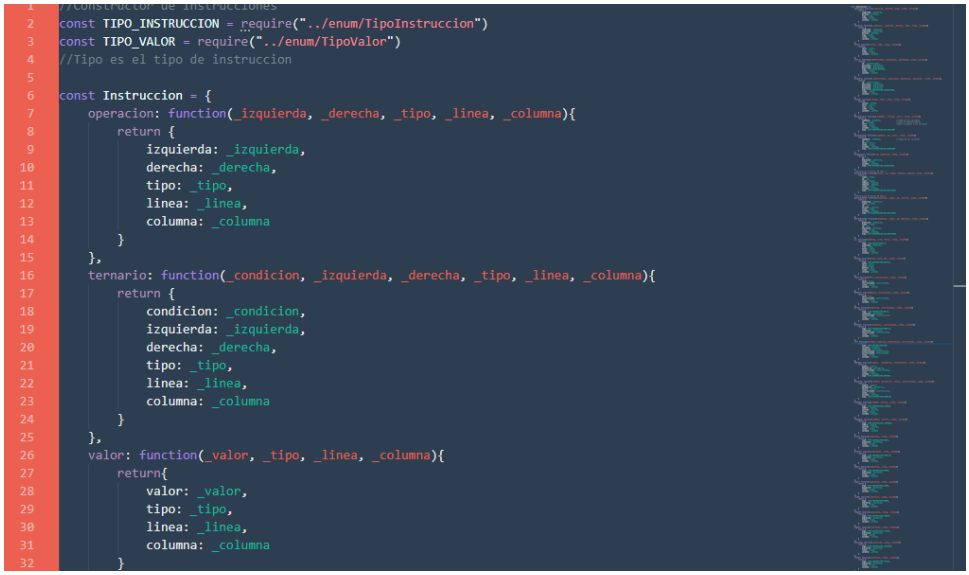
ListaErrores.js

Almacena los errores que se van a reportar.

```
//Imports
var error = require("./Error");

//Constructor
class ListaErrores {
    constructor() {
        this.lista = new Array();
     }

    add(tipo, descripcion, linea, columna){
        var nuevo = new error(tipo, descripcion, linea, columna);
        this.lista.push(nuevo);
    }
}

module.exports = ListaErrores;
```

Instrucción.js

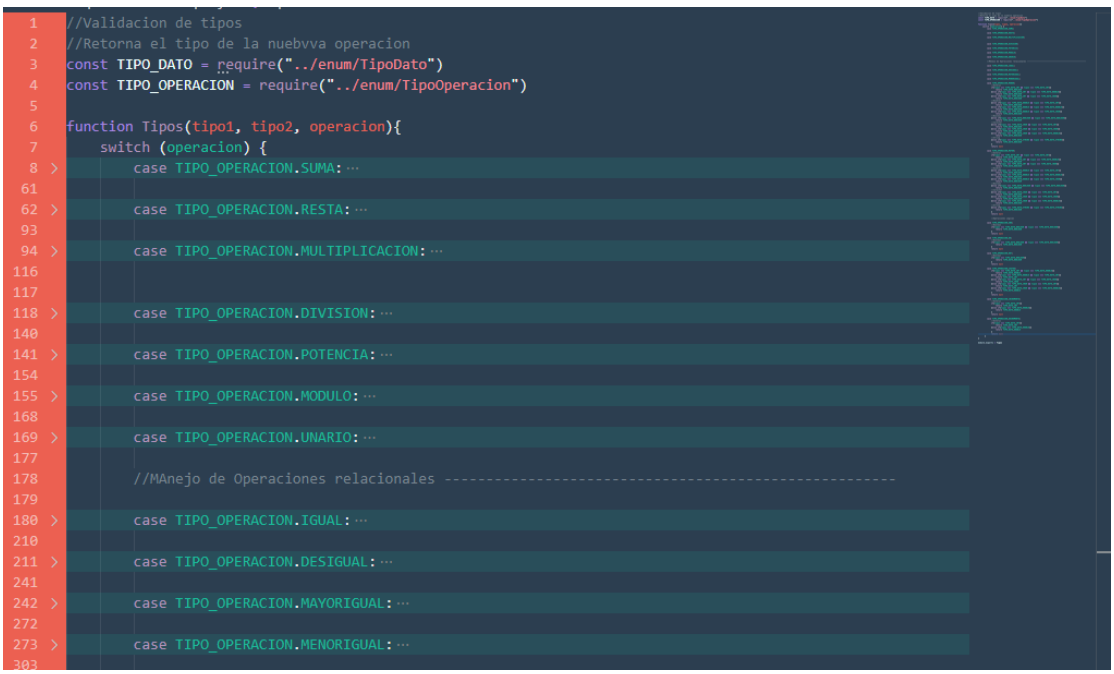
Genera los objetos de tipo instrucción del lenguaje. Esta clase es el pilar de todo lo que se maneja posteriorment ya que cada una de las otras clases consisten en extaraer la informacion de las instrucciones e ir procesando cada uno de sus valores.



```
//Constructor de instrucciones
const TIPO_INSTRUCCION = require("../enum/TipoInstruccion")
const TIPO_VALOR = require("../enum/TipoValor")
//Tipo es el tipo de instruccion

const Instruccion = {
    operacion: function(_izquierda, _derecha, _tipo, _linea, _columna){
        return {
            izquierda: _izquierda,
            derecha: _derecha,
            tipo: _tipo,
            linea: _linea,
            columna: _columna
        }
    },
    ternario: function(_condicion, _izquierda, _derecha, _tipo, _linea, _columna){
        return {
            condicion: _condicion,
            izquierda: _izquierda,
            derecha: _derecha,
            tipo: _tipo,
            linea: _linea,
            columna: _columna
        }
    },
    valor: function(_valor, _tipo, _linea, _columna){
        return{
            valor: _valor,
            tipo: _tipo,
            linea: _linea,
            columna: _columna
        }
```

Tipos.js

Es un switch que maneja los tipos de las entradas de cada instrucción. Es invocado para verificar que los tipos cumplen con las especificaciones del lenguaje. Es llamado unicamente por el metodo de operaciones aritmeticas cuando tiene que verificar tipos.

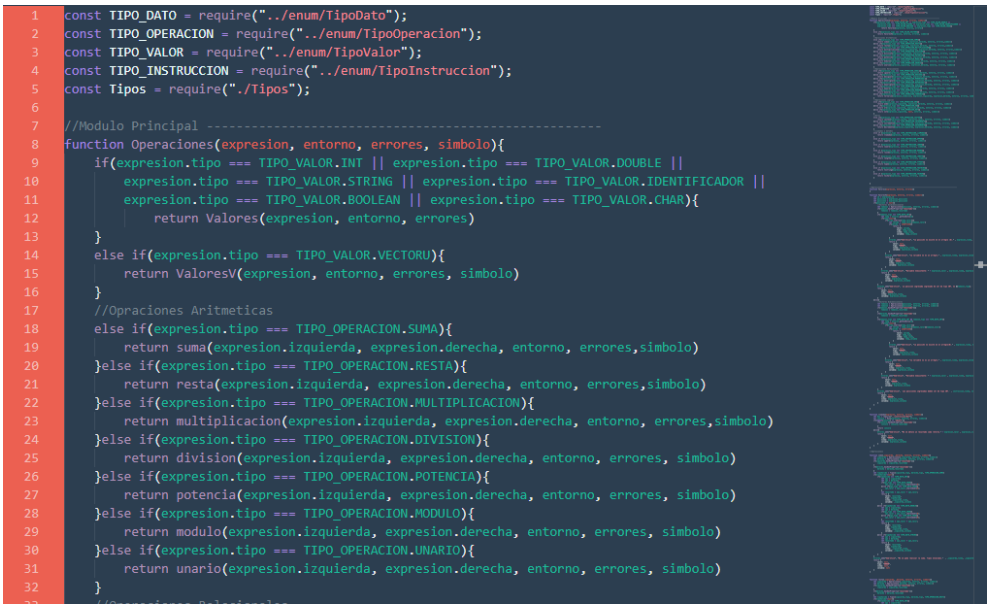


```
//Validacion de tipos
//Retorna el tipo de la nuebvva operacion
const TIPO_DATO = require("../enum/TipoDato")
const TIPO_OPERACION = require("../enum/TipoOperacion")

function Tipos(tipo1, tipo2, operacion){
    switch (operacion) {
        case TIPO_OPERACION.SUMA: ···

        case TIPO_OPERACION.RESTA: ···

        case TIPO_OPERACION.MULTIPLICACION: ···


        case TIPO_OPERACION.DIVISION: ···

        case TIPO_OPERACION.POTENCIA: ···

        case TIPO_OPERACION.MODULO: ···

        case TIPO_OPERACION.UNARIO: ···

        //MAnejo de Operaciones relacionales ----------------------------------------------------

        case TIPO_OPERACION.IGUAL: ···

        case TIPO_OPERACION.DESIGUAL: ···

        case TIPO_OPERACION.MAYORIGUAL: ···

        case TIPO_OPERACION.MENORIGUAL: ···
```

NOTA: Los retornos de tipo objeto se toman como valores que se van a asignar o usar en otros lados. Las cadenas son sinonimos de errores y los Array significa que se invoco un break o un continue. Esto es importante porque dependiendo del contexto puede ser un error.

Operaciones.js

Aquí se ejecutan y validan todas las expresiones que retornen algun valor. Hay un switch que separa cada instrucción por su tipo y lo redirige con su metodo respectivo para su ejecucion. En un nivel general, las instrucciones se separan en los valores a operar (los cuales son instrucciones) y se viuelve a llamar al metodo instrucciones de nuevo para que las ejecute y retorne el valor con el que se tiene que operar. Esto funciona por recursividad. Al final retornan un objeto con la informacion relevante del valor obtenido como el tipo o donde se originó. Como detalle, aquí se manejan ciertas funciones como los length, round o upper.



```
const TIPO_DATO = require("../enum/TipoDato");
const TIPO_OPERACION = require("../enum/TipoOperacion");
const TIPO_VALOR = require("../enum/TipoValor");
const TIPO_INSTRUCCION = require("../enum/TipoInstruccion");
const Tipos = require("./Tipos");

//Modulo Principal ------------------------------------------------------
function Operaciones(expresion, entorno, errores, simbolo){
    if(expresion.tipo === TIPO_VALOR.INT || expresion.tipo === TIPO_VALOR.DOUBLE ||
        expresion.tipo === TIPO_VALOR.STRING || expresion.tipo === TIPO_VALOR.IDENTIFICADOR ||
        expresion.tipo === TIPO_VALOR.BOOLEAN || expresion.tipo === TIPO_VALOR.CHAR){
            return Valores(expresion, entorno, errores)
    }
    else if(expresion.tipo === TIPO_VALOR.VECTORU){
        return ValoresV(expresion, entorno, errores, simbolo)
    }
    //Opraciones Aritmeticas
    else if(expresion.tipo === TIPO_OPERACION.SUMA){
        return suma(expresion.izquierda, expresion.derecha, entorno, errores,simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.RESTA){
        return resta(expresion.izquierda, expresion.derecha, entorno, errores,simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.MULTIPLICACION){
        return multiplicacion(expresion.izquierda, expresion.derecha, entorno, errores,simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.DIVISION){
        return division(expresion.izquierda, expresion.derecha, entorno, errores, simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.POTENCIA){
        return potencia(expresion.izquierda, expresion.derecha, entorno, errores, simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.MODULO){
        return modulo(expresion.izquierda, expresion.derecha, entorno, errores, simbolo)
    }else if(expresion.tipo === TIPO_OPERACION.UNARIO){
        return unario(expresion.izquierda, expresion.derecha, entorno, errores, simbolo)
    }
```

```
94  //Operaciones-----------------------------------------------------------------------------
95  function Valores(expresion, entorno, errores){···
154 }
155
156 function ValoresV(expresion, entorno, errores, simbolo){···
266 }
267
268
269 function llamada(expresion, entorno, errores, simbolo){
270     let Run = require("../instruccion/Run");
271     var consola = Run(expresion, entorno, errores, simbolo)
272     if(typeof(consola) == 'object'){
273         if(consola.hasOwnProperty('resultado')){
274             consola = consola.resultado
275         }
276         return consola
277     }else{
278         errores.add("Semántico", "No se obtuvo un resultado como retorno." + expresion.valor , expresion.li
279         return {
280             valor: null,
281             tipo: "ERROR",
282             linea: expresion.linea,
283             columna: expresion.columna
284         }
285     }
286 }
287
288 //Operacioness
289
```

Iniciar.js

Este metodo es el que se llama para ejecutar las instrucciones. Como primero debe de declararse metodos y declaraciones. Este metodo primero se encarga de buscar cada instrucción que sea una declaracion o asignacion y la ejecuta. Cuando finalice, va a ejecutar el metodo run. Ejecuta cada una de las sentencias de esta clase que vengan. Si no las hay, ahí finaliza el programa. Se retorna la variable salida, la cual va almacenando cada error que venga en la invcacion de los metodos de ejecucion.

```
13  function Iniciar(instrucciones, entorno, errores, simbolo, metodo){
14      var salida = ""
15
16      //Declaracion, asignacion y creacion de metodos y variables
17      for(let i = 0; i< instrucciones.length; i++){
18          if (instrucciones[i].tipo === TIPO_INSTRUCCION.DECLARACIONV){
19              var consola = DeclararVariable(instrucciones[i], entorno, errores, simbolo, "GLOBAL")
20              if(consola != null){
21                  salida += consola + "\n"
22              }
23          }
24          else if (instrucciones[i].tipo === TIPO_INSTRUCCION.DECLARACIONA1 || instrucciones[i].tipo === TIPO_
25              var consola = DeclararArreglos(instrucciones[i], entorno, errores, simbolo, "GLOBAL")
26              if(consola != null){
27                  salida += consola +'\n'
28              }
29          }
30          else if (instrucciones[i].tipo === TIPO_INSTRUCCION.DMETODO){
31              var consola = DeclararMetodo(instrucciones[i], entorno, errores, metodo)
32              if(consola != null){
33                  salida += consola +'\n'
34              }
35          }
36
37          else if (instrucciones[i].tipo === TIPO_INSTRUCCION.DFUNCION){
38              var consola = DeclararFuncion(instrucciones[i], entorno, errores, metodo)
39              if(consola != null){
40                  salida += salida += consola +'\n'
41              }
42          }
43      }
44
```

Run.js

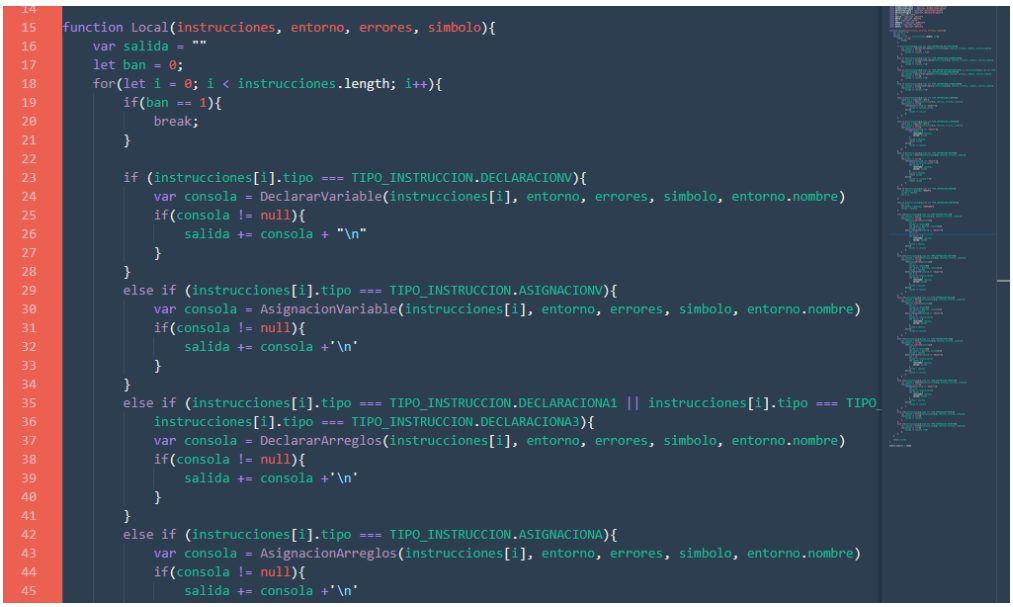
El metodo run es el equivalente a la llamada de un metodo. Los metodos al declararse se separan los parametros, las instrucciones y el return. El metodo run lo busca en la tabla de metodos y lo guarda. Como resumen, se encarga de crear un nuevo entorno, declara y asigna estos valores a la tabla del entorno local. Verifica si existe un return en el metodo. Si no existe, simplemente invoca el metodo Local y ejecuta las instrucciones. Si lo tiene, entonces revisa que venga un return en las instrucciones a ejecutar. Luego ejecuta las instrucciones con Local y verifica que el return que este haya traido sea de tipo objeto. Si viene otra osa como un Array, es error.

```
function Run(instruccion, entorno, errores, simbolo){
    var salida = ""
    var buscar = entorno.getMetodo(instruccion.nombre)
    if(buscar != null){
        var entornoLocal = new Entorno(entorno, instruccion.nombre)
        entornoLocal.setRetorno(buscar.return)
        var existeReturn = false
        if(buscar.return == null){
            existeReturn = true
        }else{
            for(let i = 0; i < buscar.instrucciones.length; i++){
                if (buscar.instrucciones[i].tipo === TIPO_INSTRUCCION.RETURN){
                    existeReturn = true
                }
            }
        }

        if(existeReturn){
            if(buscar.parametros != null){
                if(instruccion.valores != null && buscar.parametros.length == instruccion.valores.length){
                    var error = false;
                    //Declarar Parametros
                    for(let i = 0 ; i < buscar.parametros.length ; i++){
                        var declarar = Instruccion.declaracionp(buscar.parametros[i].tipodato, buscar.parame
                        var consola = DeclararParametro(declarar, entornoLocal, errores, simbolo, entornoLoc
                        if(consola != null){
                            error = true;
                            salida += consola +'\n'
                        }
                    }
                    if(error){
                        return salida
```

```
                }
            }
            else {
                let ejecutar = Local(buscar.instrucciones, entornoLocal, errores, simbolo)
                if(ejecutar != null){
                    if(Array.isArray(ejecutar)){
                        errores.add("Semántico", `Las sentencias continue o break solo se pueden usar en cic
                        return `Las sentencias continue o break solo se pueden usar en ciclos.`
                    }else if(typeof(ejecutar) == 'object'){
                        return{
                            resultado: ejecutar.resultado,
                            salida: ejecutar.salida
                        }
                    }else{
                        salida += ejecutar
                        return salida
                    }
                }
                return salida
            }
        }else{
            errores.add("Semántico", `La función '${instruccion.nombre}' carece de un return.` , instruccior
            return `La función '${instruccion.nombre}' carece de un return.`
        }
    }
    errores.add("Semántico", `El metodo '${instruccion.nombre}' no existe.` , instruccion.linea, instruccior
    return `El metodo '${instruccion.nombre}' no existe.`
}

module.exports = Run
```

Local.js

El metodo local es como una variacuion del metodo iniciar. Va recorriendo la lista de instrucciones que le fue pasada y conforme va detectando su tipo las va enviando a su respectivo metodo para que sean ejecutadas. Al igual, se revisa que va recibiendo y en una variable salida se van añadiendo cosas como errores o las cosas que se van a mandar en el print.



```
function Local(instrucciones, entorno, errores, simbolo){
    var salida = ""
    let ban = 0;
    for(let i = 0; i < instrucciones.length; i++){
        if(ban == 1){
            break;
        }

        if (instrucciones[i].tipo === TIPO_INSTRUCCION.DECLARACIONV){
            var consola = DeclararVariable(instrucciones[i], entorno, errores, simbolo, entorno.nombre)
            if(consola != null){
                salida += consola + "\n"
            }
        }
        else if (instrucciones[i].tipo === TIPO_INSTRUCCION.ASIGNACIONV){
            var consola = AsignacionVariable(instrucciones[i], entorno, errores, simbolo, entorno.nombre)
            if(consola != null){
                salida += consola +'\n'
            }
        }
        else if (instrucciones[i].tipo === TIPO_INSTRUCCION.DECLARACIONA1 || instrucciones[i].tipo === TIPO_
            instrucciones[i].tipo === TIPO_INSTRUCCION.DECLARACIONA3){
            var consola = DeclararArreglos(instrucciones[i], entorno, errores, simbolo, entorno.nombre)
            if(consola != null){
                salida += consola +'\n'
            }
        }
        else if (instrucciones[i].tipo === TIPO_INSTRUCCION.ASIGNACIONA){
            var consola = AsignacionArreglos(instrucciones[i], entorno, errores, simbolo, entorno.nombre)
            if(consola != null){
                salida += consola +'\n'
```

Return.js

Gracias al retorno que maneja el entorno, es posible hacer validaciones de si debe hacer o no un retorno. Este metodo hace eso evaluando la expresion que se quiera retornar si es que viene y lo compara con el tipo que arrastra el entorno. Si resulta que existe retorno y no hay uno definido es error, lo mismo cuando no se devuelve nada y el mtodo explicitamente lo solicita. Lo que va a retornar es un objeto con la informacion relevante del resultado.

```
function Return(instruccion, entorno, errores, simbolo){

    if(instruccion.valor == null){
        if(entorno.retorno == ""){
            return null
        }else{
            errores.add("Semántico", `Retorno Inexistente. Se esperaba un objeto de tipo ${entorno.retorno}`
            return `Retorno Inexistente. Se esperaba un objeto de tipo ${entorno.retorno}`
        }
    }else{
        if(entorno.retorno == ""){
            errores.add("Semántico", `Los metodos no retornan valores.` , instruccion.linea, instruccion.col
            return `Los metodos no retornan valores.`
        }else{
            let expresion = Operaciones(instruccion.valor,entorno, errores, simbolo)
            console.log(expresion)
            if(expresion.hasOwnProperty('resultado')){
                expresion = expresion.resultado
            }
            if(expresion.tipo == entorno.retorno){
                return {
                    valor: expresion.valor,
                    tipo: expresion.tipo,
                    linea: expresion.linea,
                    columna: expresion.columna
                }
            }else{
                errores.add("Semántico", `Retorno de tipo ${expresion.tipo}. Se esperaba un objeto de tipo $
                return `Retorno de tipo ${expresion.tipo}. Se esperaba un objeto de tipo ${entorno.retorno}`
            }
        }
    }
```

AsignacionArreglo.js

Este metodo se encarga de asignar valores a un arreglos. A pesar de separr en bidimensional o unidimensional el proceso es el mismo. Primero verifica que exista el arreglo dentro d la tabla de simbolos, calcula el valor a asignar y verifica tipos. Si todo entra bien, calcula las posiciones donde se va a asignar el arreglos. Si no son enteros, es error. Pasa entonces a verificar que la variable que se manda a llamar es arreglos, si no, otra vez se manda con los errores. Una vez hecho, verifica que la posicion exista dentro del arreglos, error de nuevo si falla. Cuando termine, por fin asigna y actualiza.

```
function AsignacionA(instruccion, entorno, errores, simbolo, entornoName){
    const id = instruccion.id
    let posicion1 = instruccion.posicion1
    let posicion2 = instruccion.posicion2
    const buscar = entorno.buscarSimboloGlobal(id)
    if(buscar){
        var valor = Operaciones(instruccion.expresion, entorno, errores, simbolo)
        if(valor.hasOwnProperty('resultado')){
            valor = valor.resultado
        }
        var variable = entorno.getSimboloE(id)
        var temp = variable.resultado
        let antiguo = temp.tipo
        let nuevo = valor.tipo

        if(antiguo===nuevo){
            if(posicion2 == null){
                //Arreglos Unidimensionales
                let tamaño1 = Operaciones(posicion1, entorno, errores, simbolo)
                if(tamaño1.hasOwnProperty('resultado')){
                    tamaño1 = tamaño1.resultado
                }
                if(tamaño1.tipo === TIPO_DATO.INT){
                    if(Array.isArray(temp.valor)){
                        let salida = temp.valor[tamaño1.valor]
                        if(salida != undefined){
                            temp.valor[tamaño1.valor] = valor.valor
                            entorno.actualizar(id , temp)
                            simbolo.update(id, variable.entorno, temp.valor)
                            return null
                        }
                        errores.add("Semántico", "La posición no existe en el arreglo (U)." , expresion.linc
```

AsignacionVariable.js

La asignacion de variable funciona exactamente igual a como lo hacen con los arreglos aunque, un poco menos complejo. Aquí solo se tiene que verificar que las variables existan y que los tipos coincidan.

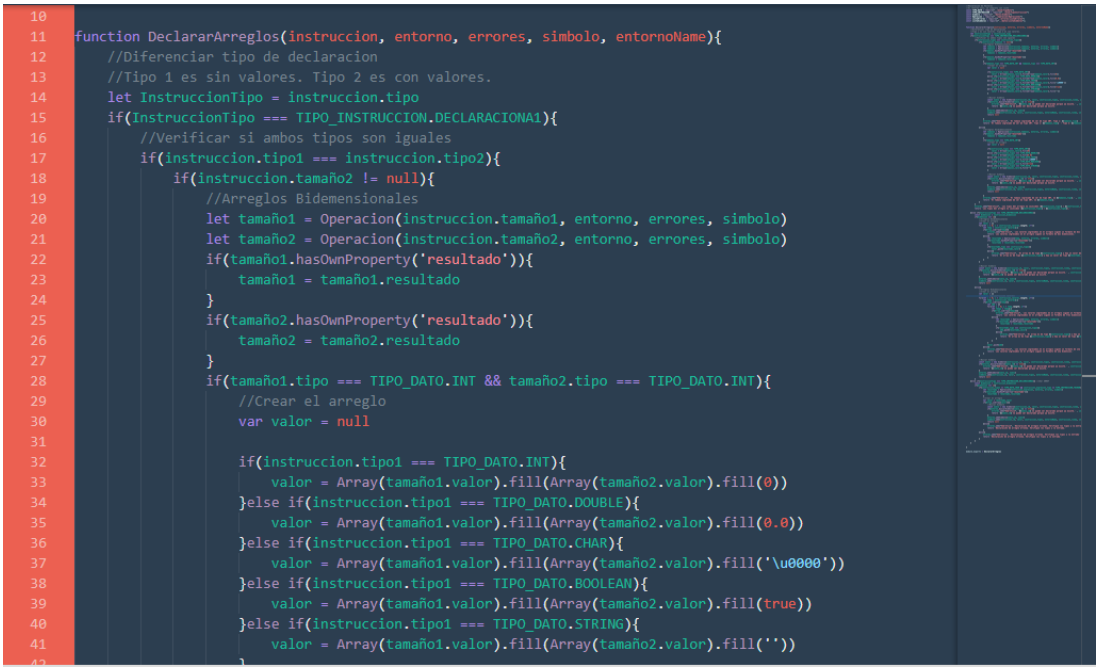
```
//Permite asignar un valor de cualquier tipo a una variable
//Si retorna null, se completo con éxito
const Operacion = require("../operacion/Operaciones")

function Asignacion(instruccion, entorno, errores, simbolo, entornoName){
    const id = instruccion.id
    const buscar = entorno.buscarSimboloGlobal(id)
    if(buscar){
        var valor = Operacion(instruccion.expresion, entorno, errores, simbolo)
        if(valor.hasOwnProperty('resultado')){
            valor = valor.resultado
        }
        var temp = entorno.getSimboloE(id)
        let variable = temp.resultado
        let antiguo = variable.tipo
        let nuevo = valor.tipo

        if(antiguo===nuevo){
            variable.valor = valor.valor
            entorno.actualizar(id , variable)
            simbolo.update(id, temp.entorno, valor.valor)
            return null
        }
        errores.add("Semántico", `${id} es de tipo ${antiguo}, no de tipo ${nuevo}.` , instruccion.linea, in
        return `${id} es de tipo ${antiguo}, no de tipo ${nuevo}.`
    }
    errores.add("Semántico", `${id} no puede recibir valores porque no existe.` , instruccion.linea, instruc
    return `${id} no puede recibir valores porque no existe.`
}

module.exports = Asignacion
```

DeclararArreglos.js

La declaracion de arreglos puede venir de tres formas en este caso: Con valores iniciales, con una declaracion de un charArray( ) o que venga vacio. Para el primer caso, solo se tiene que verificar que la entrada venga como un arreglo para que pueda ser usada en un ciclo. Se verifica tambien que los tipos concuerden conforme se va añadiendo al mismo y luego se intenta añadir a la tabla de simbolos. Al igual que antes, si existe el nombre es un error.

La declaracion con el charArray sigue el mismo patron. Lo unico que cambia es que aquí manda a llamar el metodo charArray a Operaciones para que lo devuelva como arreglo. Mismas verificaciones de siempre, y lo asigna al arreglo creado. Lo intenta guardar a la tabla y retorna.

La que viene vacio, una vez mas, se repite en el mismo proceso. Lo unico que cambia aca es que no evalua ninguna expresion o calculo. El array se va llenando con los valores por defecto en uncion de su tipo declarado.



```
function DeclararArreglos(instruccion, entorno, errores, simbolo, entornoName){
    //Diferenciar tipo de declaracion
    //Tipo 1 es sin valores. Tipo 2 es con valores.
    let InstruccionTipo = instruccion.tipo
    if(InstruccionTipo === TIPO_INSTRUCCION.DECLARACIONA1){
        //Verificar si ambos tipos son iguales
        if(instruccion.tipo1 === instruccion.tipo2){
            if(instruccion.tamaño2 != null){
                //Arreglos Bidemensionales
                let tamaño1 = Operacion(instruccion.tamaño1, entorno, errores, simbolo)
                let tamaño2 = Operacion(instruccion.tamaño2, entorno, errores, simbolo)
                if(tamaño1.hasOwnProperty('resultado')){
                    tamaño1 = tamaño1.resultado
                }
                if(tamaño2.hasOwnProperty('resultado')){
                    tamaño2 = tamaño2.resultado
                }
                if(tamaño1.tipo === TIPO_DATO.INT && tamaño2.tipo === TIPO_DATO.INT){
                    //Crear el arreglo
                    var valor = null

                    if(instruccion.tipo1 === TIPO_DATO.INT){
                        valor = Array(tamaño1.valor).fill(Array(tamaño2.valor).fill(0))
                    }else if(instruccion.tipo1 === TIPO_DATO.DOUBLE){
                        valor = Array(tamaño1.valor).fill(Array(tamaño2.valor).fill(0.0))
                    }else if(instruccion.tipo1 === TIPO_DATO.CHAR){
                        valor = Array(tamaño1.valor).fill(Array(tamaño2.valor).fill('\u0000'))
                    }else if(instruccion.tipo1 === TIPO_DATO.BOOLEAN){
                        valor = Array(tamaño1.valor).fill(Array(tamaño2.valor).fill(true))
                    }else if(instruccion.tipo1 === TIPO_DATO.STRING){
                        valor = Array(tamaño1.valor).fill(Array(tamaño2.valor).fill(''))
```

//Faltaron la explicacion de un par de metodos. Repiten el mismo patron con verificacion de tipos, busqueda en la tabla y repetir la misma ejecucion que la instrucción indica acá (el ciclo while, por ejemplo, se ejecuta en otro ciclo while).