MANUAL TECNICO

TypeWise

Juan Pedro Valle Lema
202101648

Encabezado

Desarrollador : Juan Pedro Valle Lema

Carnet: 202101648

Curso: Organización de Lenguajes y Compiladores 1

Sección: C

Lenguaje usado: TypeScript y JavaScript

IDE Utilizada: Visual Studio Code

Nombre del Sistema: Windows 10

Programa: TypeWise

Backend: localhost:5000

Frontend: localhost:3000

Librerias o Frameworks Utilizados:

Desarrollo Backend:

Node Js en su versión 18.16.0

Jison

Cors

Express

Desarrollo Frontend:

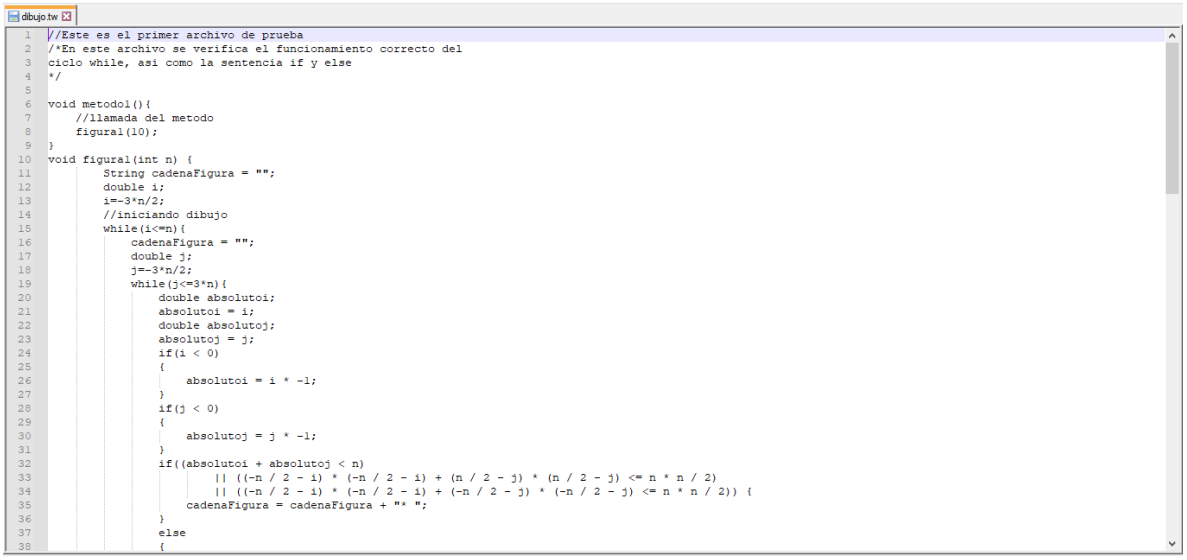
React

Principio, técnica o paradigma aplicado de programación

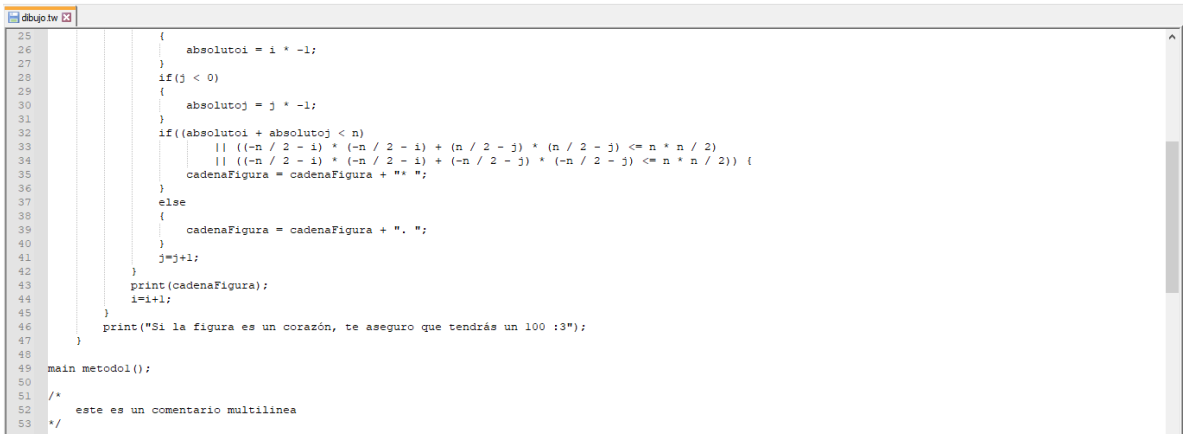
Se utilizo el paradigma de programación orientada a objetos y el patrón interprete esto ya que de esta forma el desarrollo del programa se volvió mucho más fácil al manejar clases abstractas como lo eran la clase abstracta expresión y la clase abstracta instrucción las cuales eran usadas para generar otras como la clase de instrucción Print o la clase de Expresión Aritmética. También se utilizo una arquitectura de tipo cliente servidor en el que el usuario del programa realiza peticiones de tipo post a nuestro servidor backend desde la interfaz gráfica es decir el frontend.

Archivo

Es importante saber que la única extensión de archivo aceptada es la extensión tw. Este es un archivo de texto plano que contiene la información que usara nuestro programa para realizar sus diversas funciones como la generación de autómatas o el análisis de cadenas.


dibujo.tw

```
//Este es el primer archivo de prueba
/*En este archivo se verifica el funcionamiento correcto del
ciclo while, asi como la sentencia if y else
*/

void metodo1(){
    //llamada del metodo
    figura1(10);
}
void figura1(int n) {
        String cadenaFigura = "";
        double i;
        i=-3*n/2;
        //iniciando dibujo
        while(i<=n){
            cadenaFigura = "";
            double j;
            j=-3*n/2;
            while(j<=3*n){
                double absolutoi;
                absolutoi = i;
                double absolutoj;
                absolutoj = j;
                if(i < 0)
                {
                    absolutoi = i * -1;
                }
                if(j < 0)
                {
                    absolutoj = j * -1;
                }
                if((absolutoi + absolutoj < n)
                        || ((-n / 2 - i) * (-n / 2 - i) + (n / 2 - j) * (n / 2 - j) <= n * n / 2)
                        || ((-n / 2 - i) * (-n / 2 - i) + (-n / 2 - j) * (-n / 2 - j) <= n * n / 2)) {
                    cadenaFigura = cadenaFigura + "* ";
                }
                else
                {
```


dibujo.tw

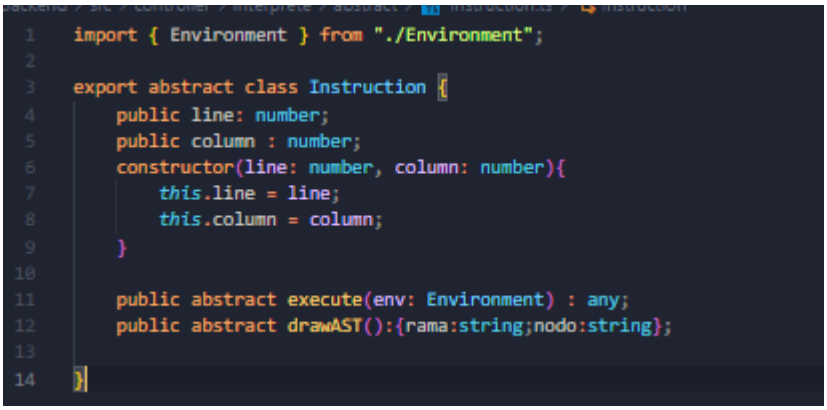
```
                {
                    absolutoi = i * -1;
                }
                if(j < 0)
                {
                    absolutoj = j * -1;
                }
                if((absolutoi + absolutoj < n)
                        || ((-n / 2 - i) * (-n / 2 - i) + (n / 2 - j) * (n / 2 - j) <= n * n / 2)
                        || ((-n / 2 - i) * (-n / 2 - i) + (-n / 2 - j) * (-n / 2 - j) <= n * n / 2)) {
                    cadenaFigura = cadenaFigura + "* ";
                }
                else
                {
                    cadenaFigura = cadenaFigura + ". ";
                }
                j=j+1;
            }
            print(cadenaFigura);
            i=i+1;
        }
        print("Si la figura es un corazón, te aseguro que tendrás un 100 :3");
    }

main metodo1();

/*
    este es un comentario multilinea
*/
```

Estos archivos cuentan de 1 sola parte que puede verse dividida por entornos, es decir tenemos un entorno global en el cual podemos tener instrucciones como la declaración de variables, la declaración de funciones o la declaración de vectores y listas. En este espacio también se encontrará la sentencia main que cuenta de la palabra reservada main y una llamada de función. La función main dicta que es la primera función que se ejecutara en nuestro programa. Luego esta el entorno local donde se tienen ya otras instrucciones como lo pueden ser sentencias cíclicas o de control entre las que se encuentra el ciclo if, el ciclo for y el ciclo while entre otros. También en este se puede ejecutar cualquier tipo de operación con las variables ya sea lógica, relacional o aritmética, en este entorno también es posible utilizar sentencias de transferencia o utilizar las funciones nativas de nuestro programa.

Clases Abstractas

Instrucción

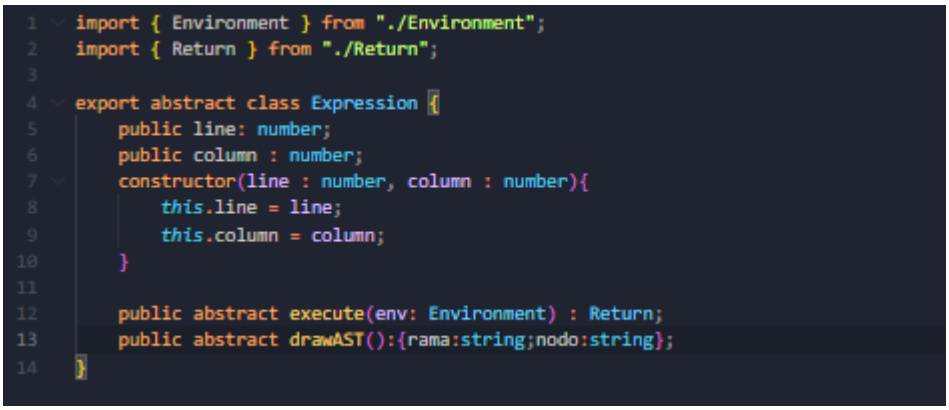
Esta clase abstracta es utilizada como base para todas las clases que sean de tipo instrucción esto quiere decir clases que interactúan con el programa pero que no necesariamente retornan algún tipo de valor o resultado en forma de tupla es decir valor y tipo del valor.



```
import { Environment } from "./Environment";

export abstract class Instruction {
    public line: number;
    public column : number;
    constructor(line: number, column: number){
        this.line = line;
        this.column = column;
    }

    public abstract execute(env: Environment) : any;
    public abstract drawAST():{rama:string;nodo:string};

}
```

Expresión

Esta clase abstracta es utilizada como base para todas las clases que sean de tipo Expresión, estas deben retornar un valor en forma de tupla en la cual se obtenga el valor real del dicho y su tipo es decir si es un entero o un doble o un booleano.



```
import { Environment } from "./Environment";
import { Return } from "./Return";

export abstract class Expression {
    public line: number;
    public column : number;
    constructor(line : number, column : number){
        this.line = line;
        this.column = column;
    }

    public abstract execute(env: Environment) : Return;
    public abstract drawAST():{rama:string;nodo:string};
}
```

Environment

Esta clase es utilizada para generar los diferentes entornos en los que se ejecuta el programa iniciando por el entorno global y luego continuando a otros como el entorno local dentro de una función, en esta clase también se tienen funciones como guardarVariable lo que nos permite guardar una variable y su entorno para su posterior uso en donde sea posible por su alcance. Otra función importante es guardarFuncion la cual nos permite guardar las diversas funciones o métodos que se crearan durante el programa.

```ts
import { Symbol } from "./Symbol";
import { tipo } from "./Return";
import { printList } from "../reports/PrintList";
import { TablaSimbolos,ListaTabla } from "../reports/TablaSimbolos";
import { Funcion } from "../Instructions/Funcion";

export class Environment {
    private variables = new Map<string, Symbol>();
    private funciones = new Map<string, Funcion>();

    constructor(private anterior: Environment | null,private nombreEntorno:string) {
        this.variables = new Map<string, Symbol>();
        this.funciones = new Map<string, Funcion>();
    }

    public guardar(id: string, valor:any, tipo:tipo, linea:number,columna:number) {
        let env: Environment | null = this;
        let tipon:string = "";
        if(!env.variables.has(id.toLowerCase())) {
            env.variables.set(id.toLowerCase(), new Symbol(valor,id,tipo));
            if(tipo == 0){
                tipon = "int";
            } else if(tipo == 1){
                tipon = "double";
            } else if(tipo == 2){
                tipon = "boolean";
            } else if(tipo == 3){
                tipon = "char";
            } else if(tipo == 4){
                tipon = "string";
            }
            ListaTabla.push(new TablaSimbolos(id.toLowerCase(),"variable",tipon,env.nombreEntorno,linea.toString(),columna.toString()));
        } else {
            printList.push("Error, la variable " + id + " ya existe en el entorno, linea " + linea + " y columna " + columna);
        }
    }
```

```ts
    public guardarFuncion(id:string,funcion:Funcion) {
        let tipo:string = "";
        let tipo2:string = "";
        let env:Environment | null = this;
        if(!env.funciones.has(id.toLowerCase())) {
            env.funciones.set(id.toLowerCase(),funcion);
            if(funcion.tipo == 0){
                tipo = "int";
                tipo2 = "funcion";
            } else if(funcion.tipo == 1){
                tipo = "double";
                tipo2 = "funcion";
            } else if(funcion.tipo == 2){
                tipo = "boolean";
                tipo2 = "funcion";
            } else if(funcion.tipo == 3){
                tipo = "char";
                tipo2 = "funcion";
            } else if(funcion.tipo == 4){
                tipo = "string";
                tipo2 = "funcion";
            } else if(funcion.tipo == 6){
                tipo = "void";
                tipo2 = "metodo";
            }

            ListaTabla.push(new TablaSimbolos(id.toLowerCase(),tipo2,tipo,env.nombreEntorno,funcion.line.toString(),funcion.column.toString()));
        } else {
            printList.push("Error, la funcion " + id + " ya existe en el entorno");
        }
    }

    public getFuncion(id:string):Funcion | null {
        let env:Environment | null = this;
        while(env != null) {
            if(env.funciones.has(id.toLowerCase())) {
                return env.funciones.get(id.toLowerCase())!;
            }
            env = env.anterior;
        }
        return null;
    }
```

```
    public getVariable(id: string): Symbol | null {
        let env: Environment | null = this;

        while(env != null) {
            if(env.variables.has(id.toLowerCase())) {
                return env.variables.get(id.toLowerCase())!;
            }
            env = env.anterior;
        }
        return null;
    }

    public getGlobal(): Environment {
        let env: Environment | null = this;

        while(env.anterior != null) {
            env = env.anterior;
        }
        return env;
    }
}
```

Symbol

Esta clase nos permite formar símbolos es decir un objeto que tiene un valor un id y un tipo

```
import { tipo,Return } from "./Return";


export class Symbol {
    public valor:any;
    public id:string;
    public tipo: tipo;

    constructor(valor:any, id:string, tipo:tipo) {
        this.valor = valor;
        this.id = id.toLowerCase();
        this.tipo = tipo;
    }
}
```

Return

Esta clase nos permite crear el retorno usado para la clase expresión el cual es un valor y un type que proviene de un enum de los diferentes tipos que puede tener una variable.

```
export enum tipo {
    INT = 0,
    DOUBLE = 1,
    BOOLEAN = 2,
    CHAR = 3,
    STRING = 4,
    NULL = 5,
    VOID = 6,
    ARRAY = 7,
    RETURN = 8,
    BREAK = 9,
    CONTINUE = 10
}

export type Return = {
    value: any,
    type: tipo
}
```

Array

Esta clase se utiliza para poder crear un objeto de vector y por medio de este agregar valores o obtenerlos.

```
import { Symbol } from "./Symbol";

export class Array{
    public values : Symbol[];

    constructor(){
        this.values = [];
    }

    public getAttribute(index: number){
        return this.values[index];
    }

    public setAttribute(index: number, value: Symbol){
        this.values[index] = value;
    }
}
```

Clases extendidas de Expresión

Acceso

Esta es la que permite generar una instancia de acceder al valor de una variable por medio de su ID. Estas clase también maneja un método drawAST con el cual se obtiene su nodo y rama usado para el reporte de grafico de AST.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";

export class Acceso extends Expression {

    constructor(public id:string,linea:number,columna:number) {
        super(linea,columna);
    }

    public execute(env:Environment): Return {
        const value = env.getVariable(this.id);
        if(value) {
            return {value: value.valor, type: value.tipo};
        } else {
            return {value: null, type: tipo.NULL};
        }
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoAcceso${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0)
        const nodoIDPrincipal = `nodoID${id2.toString()}`;
        let ramaAcceso = `${nodoPrincipal}[label="Acceso"];\n`
        ramaAcceso += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
        ramaAcceso += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        return {rama:ramaAcceso,nodo:nodoPrincipal};
    }
}
```

Aritmética

Esta clase se encarga del manejo de todas las posibles operaciones Aritméticas dentro de nuestro programa por medio de la obtención de los operadores izquierdo y derecho, también la obtención del operador de esta para saber qué tipo de operación se realizará.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";
import { tipoAritmetica } from "../utils/TipoAritmetica";
import { tablaSuma,tablaResta,tablaMultiplicacion,tablaDivision,tablaModulo,tablaPotencia } from "../utils/TablaDominante";

export class Aritmetica extends Expression {
    constructor(private izquierdo: Expression, private derecho: Expression, private operador: tipoAritmetica, linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env: Environment): Return {
        if(this.operador == tipoAritmetica.SUMA){
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            const tipoDominante = tablaSuma[operador1.type][operador2.type];
            switch(tipoDominante){
                case tipo.INT:
                    if(operador1.type == tipo.BOOLEAN) {
                        operador1.value = operador1.value ? 1 : 0;
                    }
                    if(operador2.type == tipo.BOOLEAN) {
                        operador2.value = operador2.value ? 1 : 0;
                    }
                    if(operador1.type == tipo.CHAR) {
                        operador1.value = operador1.value.charCodeAt(0);
                    }
                    if(operador2.type == tipo.CHAR) {
                        operador2.value = operador2.value.charCodeAt(0);
                    }
                    return {value: operador1.value + operador2.value, type: tipo.INT};
                case tipo.DOUBLE:
                    if(operador1.type == tipo.BOOLEAN) {
                        operador1.value = operador1.value ? 1 : 0;
                    }
                    if(operador2.type == tipo.BOOLEAN) {
                        operador2.value = operador2.value ? 1 : 0;
                    }
                    if(operador1.type == tipo.CHAR) {
                        operador1.value = operador1.value.charCodeAt(0);
                    }
                    if(operador2.type == tipo.CHAR) {
                        operador2.value = operador2.value.charCodeAt(0);
                    }
                    return {value: operador1.value + operador2.value, type: tipo.DOUBLE};
                case tipo.STRING:
                    return {value: operador1.value + operador2.value, type: tipo.STRING};
            }
        }
        else if(this.operador == tipoAritmetica.RESTA) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            const tipoDominante = tablaResta[operador1.type][operador2.type];
            switch(tipoDominante){
                case tipo.INT:
                    if(operador1.type == tipo.BOOLEAN) {
                        operador1.value = operador1.value ? 1 : 0;
                    }
                    if(operador2.type == tipo.BOOLEAN) {
                        operador2.value = operador2.value ? 1 : 0;
                    }
                    if(operador1.type == tipo.CHAR) {
                        operador1.value = operador1.value.charCodeAt(0);
                    }
                    if(operador2.type == tipo.CHAR) {
                        operador2.value = operador2.value.charCodeAt(0);
                    }
                    return {value: operador1.value - operador2.value, type: tipo.INT};
                case tipo.DOUBLE:
                    if(operador1.type == tipo.BOOLEAN) {
                        operador1.value = operador1.value ? 1 : 0;
                    }
                    if(operador2.type == tipo.BOOLEAN) {
                        operador2.value = operador2.value ? 1 : 0;
                    }
                    if(operador1.type == tipo.CHAR) {
                        operador1.value = operador1.value.charCodeAt(0);
                    }
                    return {value: operador1.value - operador2.value, type: tipo.DOUBLE};
            }
        }
```

```
        else if(this.operador == tipoAritmetica.POTENCIA) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            const tipoDominante = tablaPotencia[operador1.type][operador2.type];
            switch(tipoDominante){
                case tipo.INT:
                    return {value: Math.pow(operador1.value, operador2.value), type: tipo.INT};
                case tipo.DOUBLE:
                    return {value: Math.pow(operador1.value, operador2.value), type: tipo.DOUBLE};
            }
        }
        else if(this.operador == tipoAritmetica.MODULO) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            const tipoDominante = tablaModulo[operador1.type][operador2.type];
            switch(tipoDominante){
                case tipo.DOUBLE:
                    return {value: operador1.value % operador2.value, type: tipo.DOUBLE};
            }
        }
        else if(this.operador == tipoAritmetica.MENOSUNARIO) {
            const operador2 = this.izquierdo.execute(env);
            if(operador2.type == tipo.INT) {
                return {value: -operador2.value, type: tipo.INT};
            }
            else if(operador2.type == tipo.DOUBLE) {
                return {value: -operador2.value, type: tipo.DOUBLE};
            }
        }
        return {value: null, type: tipo.NULL};
    }
```

Backend > src > controller > interprete > Expressions > Aritmetica.ts > Aritmetica > execute

jison

```
    }
    else if(this.operador == tipoAritmetica.MULTIPLICACION) {
        const operador1 = this.izquierdo.execute(env);
        const operador2 = this.derecho.execute(env);
        const tipoDominante = tablaMultiplicacion[operador1.type][operador2.type];
        switch(tipoDominante){
            case tipo.INT:
                if(operador1.type == tipo.CHAR) {
                    operador1.value = operador1.value.charCodeAt(0);
                }
                if(operador2.type == tipo.CHAR) {
                    operador2.value = operador2.value.charCodeAt(0);
                }
                return {value: operador1.value * operador2.value, type: tipo.INT};
            case tipo.DOUBLE:
                if(operador1.type == tipo.CHAR) {
                    operador1.value = operador1.value.charCodeAt(0);
                }
                if(operador2.type == tipo.CHAR) {
                    operador2.value = operador2.value.charCodeAt(0);
                }
                return {value: operador1.value * operador2.value, type: tipo.DOUBLE};
        }
    }
    else if(this.operador == tipoAritmetica.DIVISION) {
        const operador1 = this.izquierdo.execute(env);
        const operador2 = this.derecho.execute(env);
        const tipoDominante = tablaDivision[operador1.type][operador2.type];
        switch(tipoDominante){
            case tipo.DOUBLE:
                if(operador1.type == tipo.CHAR) {
                    operador1.value = operador1.value.charCodeAt(0);
                }
                if(operador2.type == tipo.CHAR) {
                    operador2.value = operador2.value.charCodeAt(0);
                }
                return {value: operador1.value / operador2.value, type: tipo.DOUBLE};
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id =  Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoAritmetica${id.toString()}`;
        let ramaA = `${nombreNodo}[label="Aritmetica"];\n`;
        if(this.operador == tipoAritmetica.MENOSUNARIO) {
            const id2 =  Math.floor(Math.random() * (999 - 0) + 0);
            const codigoRama = `nodoAritmeticaLix${id2.toString()}`;
            let nodoMU = `${codigoRama}[label="Menos Unario"];\n`;
            ramaA += nodoMU;
            ramaA += `${nombreNodo} -> ${codigoRama};\n`;
            const codigoRamas: {rama:string, nodo:string} = this.izquierdo.drawAST();
            ramaA += codigoRamas.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas.nodo};\n`;
        } else {
            const codigoRamas: {rama:string, nodo:string} = this.izquierdo.drawAST();
            ramaA += codigoRamas.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas.nodo};\n`;
            const id2 =  Math.floor(Math.random() * (999 - 0) + 0);
            const codigoRama = `nodoAritmeticaLop${id2.toString()}`;
            let nodoOperador = `${codigoRama}[label="${tipoAritmetica[this.operador]}"];\n`;
            ramaA += nodoOperador;
            ramaA += `${nombreNodo} -> ${codigoRama};\n`;
            const codigoRamas2: {rama:string, nodo:string} = this.derecho.drawAST();
            ramaA += codigoRamas2.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas2.nodo};\n`;
        }
        return {rama: ramaA, nodo: nombreNodo};
    }
}
```

Decremento

Esta clase se encarga de manejar las instancias en las que se tenga la expresión ID --; tener esa expresión significa que el valor numérico de esa id se le restara un 1. Esta clase también maneja una función drawAST lo cual nos sirve al momento de realizar el reporte del AST.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";

export class Decremento extends Expression {
    constructor(private id:string,linea:number,columna:number) {
        super(linea,columna);
    }

    public execute(env:Environment): Return {
        if(this.id != null) {
            let value = env.getVariable(this.id);
            if(value != null) {
                if(value.tipo == tipo.INT) {
                    value.valor = value.valor - 1;
                    return {value: value.valor, type: value.tipo};
                } else if(value.tipo == tipo.DOUBLE) {
                    value.valor = value.valor - 1.0;
                    return {value: value.valor, type: value.tipo};
                } else {
                    return {value: null, type: tipo.NULL};
                }
            } else {
                throw new Error("Variable no encontrada");
            }
        }
       return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoDecremento${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoID${id2.toString()}`;
        let ramaDecremento = `${nodoPrincipal}[label="Decremento"];\n`
        ramaDecremento += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
        ramaDecremento += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        return {rama:ramaDecremento,nodo:nodoPrincipal};
    }
}
```

Incremento

Esta clase se encarga de manejar las instancias en las que se tenga la expresión ID ++; tener esa expresión significa que el valor numérico de esa id se le sumara un 1. Esta clase también maneja una función drawAST lo cual nos sirve al momento de realizar el reporte del AST.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";

export class Incremento extends Expression {
    constructor(private id:string,linea:number,columna:number) {
        super(linea,columna);
    }

    public execute(env:Environment): Return {
        if(this.id != null) {
            let value = env.getVariable(this.id);
            if(value != null) {
                if(value.tipo == tipo.INT) {
                    value.valor = value.valor + 1;
                    return {value: value.valor, type: value.tipo};
                } else if(value.tipo == tipo.DOUBLE) {
                    value.valor = value.valor + 1.0;
                    return {value: value.valor, type: value.tipo};
                } else {
                    return {value: null, type: tipo.NULL};
                }
            } else {
                throw new Error("Variable no encontrada");
            }
        }
       return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoIncremento${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoID${id2.toString()}`;
        let ramaIncremento = `${nodoPrincipal}[label="Incremento"];\n`
        ramaIncremento += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
        ramaIncremento += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        return {rama:ramaIncremento,nodo:nodoPrincipal};
    }
}
```

Length

Esta clase es la encargada de manejar la funcionalidad de la función nativa length la cual nos permite obtener la longitud de una cadena es decir la longitud de un string. También se tiene la función drawAST para el reporte.

```ts
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class Length extends Expression{
    constructor(private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return{
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.STRING){
            return  {value: valor.value.length, type: tipo.INT};
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoLength${id.toString()}`;
        let ramaLength = `${nodoPrincipal}[label="Length"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaLength += codigoRama.rama;
        ramaLength += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaLength,nodo:nodoPrincipal};
    }
}
```

Llamada Función

Esta clase se encargaba de manejar los llamados que realizamos a una función creada teniendo también posibilidad de que si retorna algo dentro de esta es decir sea una función el valor si sea devuelto con el tipo de la función mientras que si es un método no se retornara un valor especifico.

```ts
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class LlamadaFuncion extends Expression {
    constructor(private id:string, private argumentos: Array<Expression>, line:number, column:number){
        super(line,column);
    }

    public execute(env:Environment):any {
        const funcion = env.getFuncion(this.id);
        if(funcion != null) {
            const envFuncion = new Environment(env.getGlobal(),"Parametro");
            if(funcion.parametros.length == this.argumentos.length) {
                for(let i = 0; i < funcion.parametros.length; i++) {
                    const valor = this.argumentos[i].execute(env);
                    const parametro = funcion.parametros[i].execute(envFuncion);
                    if(valor.type == parametro.type) {
                        envFuncion.guardar(parametro.value,valor.value,valor.type,this.line,this.column);
                    } else {
                        console.log("Error, el parametro " + parametro.value + " no es del tipo " + valor.type + " linea " + this.line + " y columna " + this.column);
                    }
                }
                const retornar = funcion.statement.execute(envFuncion);
                //console.log(retornar);
                if(retornar != undefined){
                    if(retornar.type ==tipo.RETURN){
                            return {value:retornar.value , type:retornar.tipo};
                    }
                }
            } else {
                console.log("Error, la funcion " + this.id + " no tiene la cantidad de parametros correcta, linea " + this.line + " y columna " + this.column);
            }
        } else {
            console.log("Error, la funcion " + this.id + " no existe, linea " + this.line + " y columna " + this.column);
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoLlamadaFunc${id.toString()}`;
        let ramaLlamada = `${nodoPrincipal}[label="Llamada Funcion"];`;
        ramaLlamada += `nodoEXLlama${nodoPrincipal}[label="${this.id}"];`;
        ramaLlamada += `${nodoPrincipal} -> nodoEXLlama${nodoPrincipal};\n`;
        return {rama:ramaLlamada,nodo:nodoPrincipal};
    }
}
```

Lógica

Esta clase se encarga del manejo de todas las posibles operaciones lógicas dentro de nuestro programa por medio de la obtención de los operadores izquierdo y derecho, también la obtención del operador de esta para saber qué tipo de operación se realizará es importante saber que el retorno de esta operación siempre será de tipo booleano.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";
import { tipoLogico } from "../utils/TipoLogico";

export class Logica extends Expression {
    constructor(private izquierdo: Expression,private derecho: Expression, private operador: tipoLogico, linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env:Environment): Return {
        if(this.operador == tipoLogico.NOT) {
            const operador1 = this.derecho.execute(env);
            return {value:!(operador1.value),type:tipo.BOOLEAN}
        } else if(this.operador == tipoLogico.AND) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            return {value:(operador1.value && operador2.value),type:tipo.BOOLEAN}
        } else if(this.operador == tipoLogico.OR) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            return {value:(operador1.value || operador2.value),type:tipo.BOOLEAN}
        }
        return {value: null, type: tipo.NULL};
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id =  Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoLogicon${id.toString()}`;
        let ramaA = `${nombreNodo}[label="Logico"];\n`;
        if(this.operador == tipoLogico.NOT) {
            const id2 =  Math.floor(Math.random() * (999 - 0) + 0);
            const codigoRama = `nodoLogicoLix${id2.toString()}`;
            let nodoMU = `${codigoRama}[label="NOT"];\n`;
            ramaA += nodoMU;
            ramaA += `${nombreNodo} -> ${codigoRama};\n`;
            const codigoRamas: {rama:string, nodo:string} = this.izquierdo.drawAST();
            ramaA += codigoRamas.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas.nodo};\n`;
        } else {
            const codigoRamas: {rama:string, nodo:string} = this.izquierdo.drawAST();
            ramaA += codigoRamas.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas.nodo};\n`;
            const id2 =  Math.floor(Math.random() * (999 - 0) + 0);
            const codigoRama = `nodoLogicoLop${id2.toString()}`;
            let nodoOperador = `${codigoRama}[label="${tipoLogico[this.operador]}"];\n`;
            ramaA += nodoOperador;
            ramaA += `${nombreNodo} -> ${codigoRama};\n`;
            const codigoRamas2: {rama:string, nodo:string} = this.derecho.drawAST();
            ramaA += codigoRamas2.rama;
            ramaA += `${nombreNodo} -> ${codigoRamas2.nodo};\n`;
        }
        return {rama: ramaA, nodo: nombreNodo};
    }
}
```

Parámetros

Esta clase se encarga del manejo de los parámetros de una función.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";

export class Parametros extends Expression {
    constructor(private tipo:tipo,private id:string,line:number,column:number){
        super(line,column);
    }

    public execute(env: Environment): Return {
        return {value: this.id, type: this.tipo}
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoParametros${id.toString()}`;
        let ramaPa = `${nodoPrincipal}[label="${tipo[this.tipo]}"];\n`
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoIDPara${id2.toString()}`;
        ramaPa += `${nodoIDPrincipal}[label="${this.id}"];\n`;
        ramaPa += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        return {rama:ramaPa,nodo:nodoPrincipal};
    }
}
```

Primitivo

Esta clase nos permite obtener el valor primitivo de una variable y su tipo.

```
import { Expression } from "../abstract/Expression"
import { tipo,Return } from "../abstract/Return";

export class Primitivo extends Expression {
    constructor(line: number, column: number, private value: any , private tipo: tipo) {
        super(line, column);
    }

    public execute(): Return {
        switch(this.tipo) {
            case tipo.INT:
                return {value: parseInt(this.value), type: tipo.INT};
            case tipo.DOUBLE:
                return {value: parseFloat(this.value), type: tipo.DOUBLE};
            case tipo.BOOLEAN:
                if(this.value.toString().toLowerCase() === "true"){
                    return {value: true, type: tipo.BOOLEAN};
                }
                return {value: false, type: tipo.BOOLEAN};
            case tipo.CHAR:
                return {value: this.value, type: tipo.CHAR};
            case tipo.STRING:
                return {value: this.value, type: tipo.STRING};
            default:
                return {value: null, type: tipo.NULL};
        }

    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoPrimitivo${id.toString()}`;
        const ramaPrimitivo = `${nodoPrincipal}[label="${this.value.toString()}"];\n`
        // nodoPrimitivo${nodoPrincipal}[label="${this.value.toString()}"];\n
        // ${nodoPrincipal} -> nodoPrimitivo${nodoPrincipal};\n;
        return {rama:ramaPrimitivo,nodo:nodoPrincipal};
    }

}
```

Relacional

Esta clase se encarga del manejo de todas las posibles operaciones relacionales dentro de nuestro programa por medio de la obtención de los operadores izquierdo y derecho, también la obtención del operador de esta para saber qué tipo de operación se realizará es importante saber que el retorno de esta operación siempre será de tipo booleano.

```ts
import { Expression } from "../abstract/Expression";
import { Return, tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";
import { tipoRelacional } from "../utils/TipoRelacional";

export class Relacional extends Expression {
    constructor(private izquierdo: Expression, private derecho: Expression, private operador: tipoRelacional,linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env: Environment): Return {
        if(this.operador == tipoRelacional.IGUALACION) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value == operador2.value, type: tipo.BOOLEAN};
        } else if(this.operador == tipoRelacional.DIFERENTE) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value != operador2.value, type: tipo.BOOLEAN};
        } else if(this.operador == tipoRelacional.MENORQUE) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value < operador2.value, type: tipo.BOOLEAN};
        } else if(this.operador == tipoRelacional.MENORIGUAL) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value <= operador2.value, type: tipo.BOOLEAN};
        } else if(this.operador == tipoRelacional.MAYORQUE) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value > operador2.value, type: tipo.BOOLEAN};
        } else if(this.operador == tipoRelacional.MAYORIGUAL) {
            const operador1 = this.izquierdo.execute(env);
            const operador2 = this.derecho.execute(env);
            if(operador1.type == tipo.CHAR) {
                operador1.value = operador1.value.charCodeAt(0);
            }
            if(operador2.type == tipo.CHAR) {
                operador2.value = operador2.value.charCodeAt(0);
            }
            return {value: operador1.value >= operador2.value, type: tipo.BOOLEAN};
        }
        return {value: null, type: tipo.NULL};
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoRelacional${id.toString()}`;
        let ramaRelacional = `${nodoPrincipal}[label="Relacional"];\n`
        const codigoRama: {rama: string, nodo: string} = this.izquierdo.drawAST();
        ramaRelacional += codigoRama.rama;
        ramaRelacional += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoRama = `nodoRelacionalLOL${id2.toString()}`;
        let nodoVar = `${nodoRama}[label="${tipoRelacional[this.operador]}"];\n`
        ramaRelacional += nodoVar;
        ramaRelacional += `${nodoPrincipal} -> ${nodoRama};\n`
        const codigoRama2: {rama: string, nodo: string} = this.derecho.drawAST();
        ramaRelacional += codigoRama2.rama;
        ramaRelacional += `${nodoPrincipal} -> ${codigoRama2.nodo};\n`
        return {rama: ramaRelacional, nodo: nodoPrincipal};
    }
}
```

Round

Esta clase es la encargada de manejar la funcionalidad de la función nativa round la cual nos permite redondear un valor de los decimales de un double a su entero más próximo. También se tiene la función drawAST para el reporte.

```
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class Round extends Expression{
    constructor(private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return {
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.DOUBLE){
            return {value: Math.round(valor.value), type: tipo.INT};
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoRound${id.toString()}`;
        let ramaRound = `${nodoPrincipal}[label="Round"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaRound += codigoRama.rama;
        ramaRound += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaRound,nodo:nodoPrincipal};
    }
}
```

ToLowerUpper

Esta clase es la encargada de manejar la funcionalidad de las funciónes nativas toLower y toUpper las cuales nos permiten hacer que una cadena sea vuelta a todo mayúscula o todo minúscula. También se tiene la función drawAST para el reporte.

```
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class ToLowerUpper extends Expression{
    constructor(private tipo:number, private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return {
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.STRING){
            if(this.tipo == 1){
                return {value: valor.value.toLowerCase(), type: tipo.STRING};
            } else if(this.tipo == 2){
                return {value: valor.value.toUpperCase(), type: tipo.STRING};
            }
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        if(this.tipo == 1){
            const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoToLower${id.toString()}`;
        let ramaPToLower = `${nodoPrincipal}[label="ToLower"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaPToLower += codigoRama.rama;
        ramaPToLower += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaPToLower,nodo:nodoPrincipal};
        } else if(this.tipo == 2){
            const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoToUpper${id.toString()}`;
        let ramaPToUpper = `${nodoPrincipal}[label="ToUpper"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaPToUpper += codigoRama.rama;
        ramaPToUpper += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaPToUpper,nodo:nodoPrincipal};
        }
       return {rama:"",nodo:""};
    }
}
```

ToString

Esta clase es la encargada de manejar la funcionalidad de la función nativa toString la cual nos permite convertir un valor de tipo entero, double o boolean a tipo string. También se tiene la función drawAST para el reporte.

```
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class ToString extends Expression{
    constructor(private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return{
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.BOOLEAN) {
            if(valor.value == true){
                valor.value = "true";
                return {value: valor.value, type: tipo.STRING};
            } else {
                valor.value = "false";
                return {value: valor.value, type: tipo.STRING};
            }
        } else if(valor.type == tipo.INT){
            let cadena:number = valor.value;
            valor.value = cadena.toString();
            return {value: valor.value, type: tipo.STRING};
        } else if(valor.type == tipo.DOUBLE){
            let cadena:number = valor.value;
            valor.value = cadena.toString();
            return {value: valor.value, type: tipo.STRING};
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoToString${id.toString()}`;
        let ramaToString = `${nodoPrincipal}[label="ToString"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaToString += codigoRama.rama;
        ramaToString += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaToString,nodo:nodoPrincipal};
    }
}
```

Truncate

Esta clase es la encargada de manejar la funcionalidad de la función nativa truncate la cual nos permite truncar los valores decimales y solo tener el entero es decir quitar los decimales. También se tiene la función drawAST para el reporte.

```typescript
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { Return,tipo } from "../abstract/Return";

export class Truncate extends Expression{
    constructor(private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return {
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.DOUBLE){
            return {value: Math.trunc(valor.value), type: tipo.DOUBLE};
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoTruncate${id.toString()}`;
        let ramaTruncate = `${nodoPrincipal}[label="Truncate"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaTruncate += codigoRama.rama;
        ramaTruncate += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaTruncate,nodo:nodoPrincipal};
    }
}
```

TypeOf

Esta clase es la encargada de manejar la funcionalidad de la función nativa typeOf la cual nos permite obtener el tipo de dato de un valor en específico. También se tiene la función drawAST para el reporte.

```
import { Return,tipo } from "../abstract/Return";

export class TypeOf extends Expression{
    constructor(private expresion:Expression, line:number, column:number){
        super(line, column);
    }


    public execute(env:Environment):Return{
        const valor = this.expresion.execute(env);
        if(valor.type == tipo.STRING){
            valor.value = "STRING";
            return {value: valor.value, type: tipo.STRING};
        } else if(valor.type == tipo.BOOLEAN ) {
            valor.value = "BOOLEAN";
            return {value: valor.value, type: tipo.STRING};
        } else if(valor.type == tipo.CHAR) {
            valor.value = "CHAR";
            return {value: valor.value, type: tipo.STRING};
        } else if(valor.type == tipo.INT){
            valor.value = "INT";
            return {value: valor.value, type: tipo.STRING};
        } else if(valor.type == tipo.DOUBLE){
            valor.value = "DOUBLE";
            return {value: valor.value, type: tipo.STRING};
        }
        return {value: null, type: tipo.NULL};
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoTypeoff${id.toString()}`;
        let ramaTypeoff = `${nodoPrincipal}[label="Typeoff"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expresion.drawAST();
        ramaTypeoff += codigoRama.rama;
        ramaTypeoff += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaTypeoff,nodo:nodoPrincipal};
    }
}
```

Clases extendidas de Instrucción

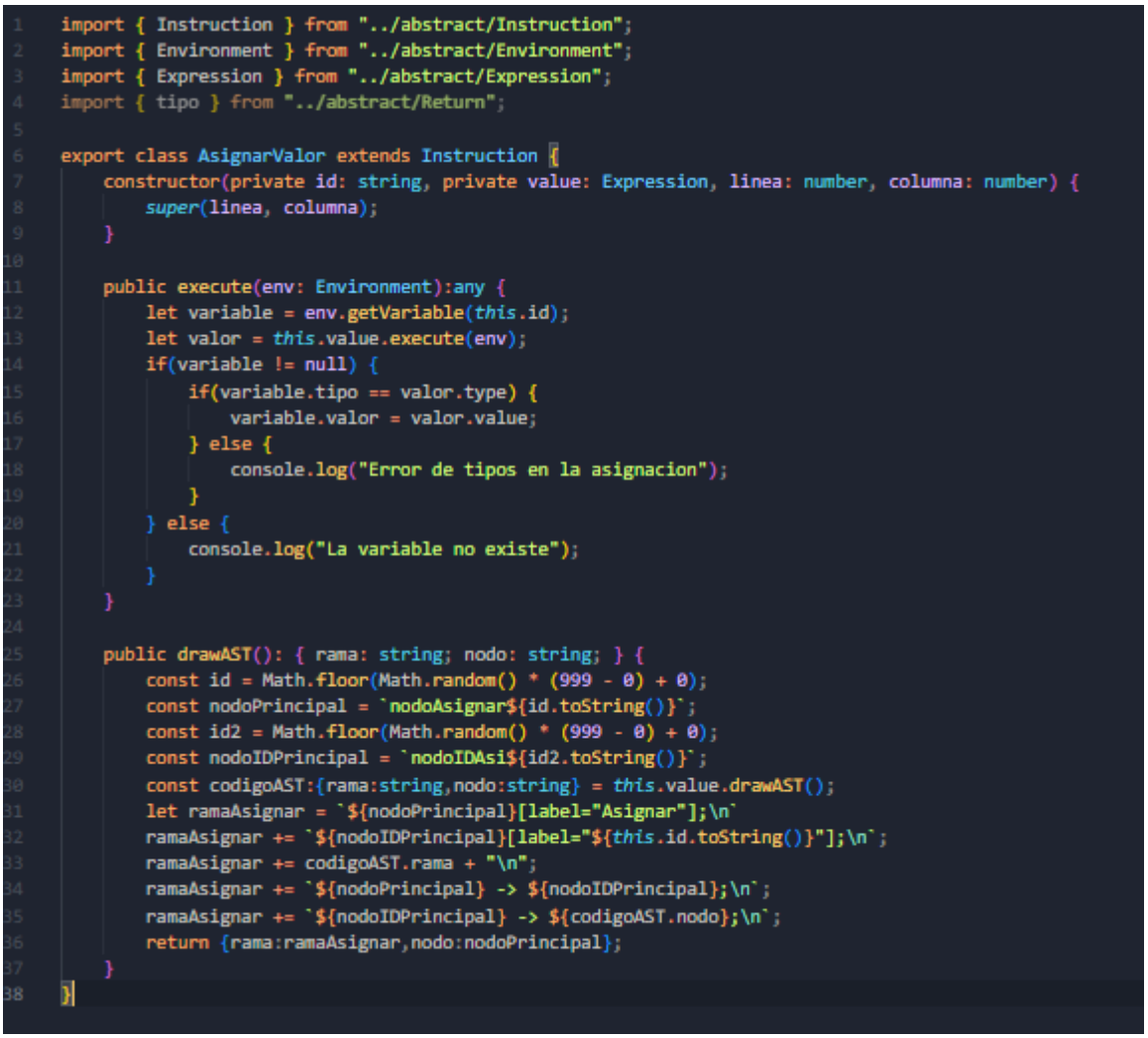
EReturn

Esta clase nos permitía generar instancias de la sentencia return ya sea dentro de funciones o dentro de sentencias. Pudiendo ser un return junto a un valor que necesitaba devolver o solo return que causaría una ruptura en el ciclo que se encuentre.

```
import { Expression } from "../abstract/Expression";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";
import { Instruction } from "../abstract/Instruction";

export class EReturn extends Instruction {
    constructor(private value:Expression, line:number, column:number){
        super(line,column);
    }

    public execute(env: Environment){
        if(this.value != null && this.value != undefined){
            let rvalue = this.value.execute(env);
            //console.log("obtengo el valor de return")
            return {value:rvalue.value, type: tipo.RETURN, tipo:rvalue.type};
        }
        return this
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoReturn${id.toString()}`;
        let ramaReturn = `${nombreNodo}[label="Return"];`;
        const codigoRama2 : {rama:string, nodo:string} = this.value.drawAST();
        ramaReturn += codigoRama2.rama;
        ramaReturn += `${nombreNodo} -> ${codigoRama2.nodo};\n`;
        return {rama:ramaReturn, nodo:nombreNodo};
    }
}
```

AsignarValor

Esta clase nos permitía manejar el asignar un nuevo valor a una variable existente siempre y cuando el tipo de valor a asignar fuera igual al de la variable que se tenia previamente.



```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression";
import { tipo } from "../abstract/Return";

export class AsignarValor extends Instruction {
    constructor(private id: string, private value: Expression, linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env: Environment):any {
        let variable = env.getVariable(this.id);
        let valor = this.value.execute(env);
        if(variable != null) {
            if(variable.tipo == valor.type) {
                variable.valor = valor.value;
            } else {
                console.log("Error de tipos en la asignacion");
            }
        } else {
            console.log("La variable no existe");
        }
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoAsignar${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoIDAsi${id2.toString()}`;
        const codigoAST:{rama:string,nodo:string} = this.value.drawAST();
        let ramaAsignar = `${nodoPrincipal}[label="Asignar"];\n`
        ramaAsignar += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
        ramaAsignar += codigoAST.rama + "\n";
        ramaAsignar += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        ramaAsignar += `${nodoIDPrincipal} -> ${codigoAST.nodo};\n`;
        return {rama:ramaAsignar,nodo:nodoPrincipal};
    }
}
```

Casteo

Esta clase nos permite manejar el casteo que es convertir un dato durante la asignación a otro tipo para que sea compatible a la variable a la que se le quiere asignar. En este caso los casteos solo eran posibles entre los tipos CHAR, INT y DOUBLE.

```
import { Instruction } from "../abstract/Instruction";
import { Return,tipo } from "../abstract/Return";
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression";

export class Casteo extends Instruction {
    constructor (private tipo:tipo,private value:Expression,line:number,column:number){
        super(line,column);
    }

    public execute(env:Environment){
        let value = this.value.execute(env);
        if(this.tipo == tipo.INT){
            if(value.type == tipo.CHAR || value.type == tipo.DOUBLE){
                if(value.type == tipo.CHAR){
                    return {value: parseInt(value.value.charCodeAt(0)), type: this.tipo};
                }else {
                    return {value: parseInt(value.value), type: this.tipo};
                }
            }
        } else if (this.tipo == tipo.DOUBLE){
            if(value.type == tipo.CHAR || value.type == tipo.INT){
                if(value.type == tipo.CHAR){
                    return {value: parseFloat(value.value.charCodeAt(0)), type: this.tipo};
                }else {
                    return {value: parseFloat(value.value), type: this.tipo};
                }
            }
        } else if (this.tipo == tipo.CHAR){
            if(value.type == tipo.INT || value.type == tipo.DOUBLE){
                if(value.type == tipo.INT){
                    return {value: String.fromCharCode(value.value), type: this.tipo};
                }
            }
        }
}
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoCasteo${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoIDCasteo${id2.toString()}`;
        const codigoAST:{rama:string,nodo:string} = this.value.drawAST();
        let ramaCasteo = `${nodoPrincipal}[label="Casteo"];\n`
        ramaCasteo += `${nodoIDPrincipal}[label="${tipo[this.tipo]}"];\n`;
        ramaCasteo += codigoAST.rama + "\n";
        ramaCasteo += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        ramaCasteo += `${nodoIDPrincipal} -> ${codigoAST.nodo};\n`;
        return {rama:ramaCasteo,nodo:nodoPrincipal};
    }
}
```

Declaration

Esta clase nos permite manejar la declaración de variables ya sea si se le asigna un valor inicial o si solo se declara la variable sin ningún valor es decir tienen un valor de inicialización predeterminado.

```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression";
import { tipo } from "../abstract/Return";

export class Declaration extends Instruction {
    private id: string;
    private tipo: tipo;
    private value: Expression | null;

    constructor(id:string, tipo:tipo, value: Expression | null, linea:number, columna:number) {
        super(linea,columna);
        this.id = id;
        this.tipo = tipo;
        this.value = value;
    }

    public execute(env: Environment): any {
        if(this.value != null) {
            const value = this.value.execute(env);
            env.guardar(this.id,value.value,this.tipo,this.line,this.column);
        } else {
            if(this.tipo == 0) {
                env.guardar(this.id,0,this.tipo,this.line,this.column);
            } else if(this.tipo == 1) {
                env.guardar(this.id,0.0,this.tipo,this.line,this.column);
            } else if(this.tipo == 2) {
                env.guardar(this.id,true,this.tipo,this.line,this.column);
            } else if(this.tipo == 3) {
                env.guardar(this.id,'\u0000',this.tipo,this.line,this.column);
            } else if(this.tipo == 4) {
                env.guardar(this.id,"",this.tipo,this.line,this.column)
            }
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoDeclarar${id.toString()}`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoIDD${id2.toString()}`;
        if(this.value != null) {
            const codigoAST:{rama:string,nodo:string} = this.value.drawAST();
            let ramaDeclarar = `${nodoPrincipal}[label="Declarar"];\n`
            ramaDeclarar += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
            ramaDeclarar += codigoAST.rama + "\n";
            ramaDeclarar += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
            ramaDeclarar += `${nodoIDPrincipal} -> ${codigoAST.nodo};\n`;
            return {rama:ramaDeclarar,nodo:nodoPrincipal};
        } else {
            let ramaDeclarar = `${nodoPrincipal}[label="Declarar"];\n`
            ramaDeclarar += `${nodoIDPrincipal}[label="${this.id.toString()}"];\n`;
            ramaDeclarar += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
            return {rama:ramaDeclarar,nodo:nodoPrincipal};
        }
    }
```

Default

Esta clase nos permitía obtener una instancia del caso default para la sentencia switch case.

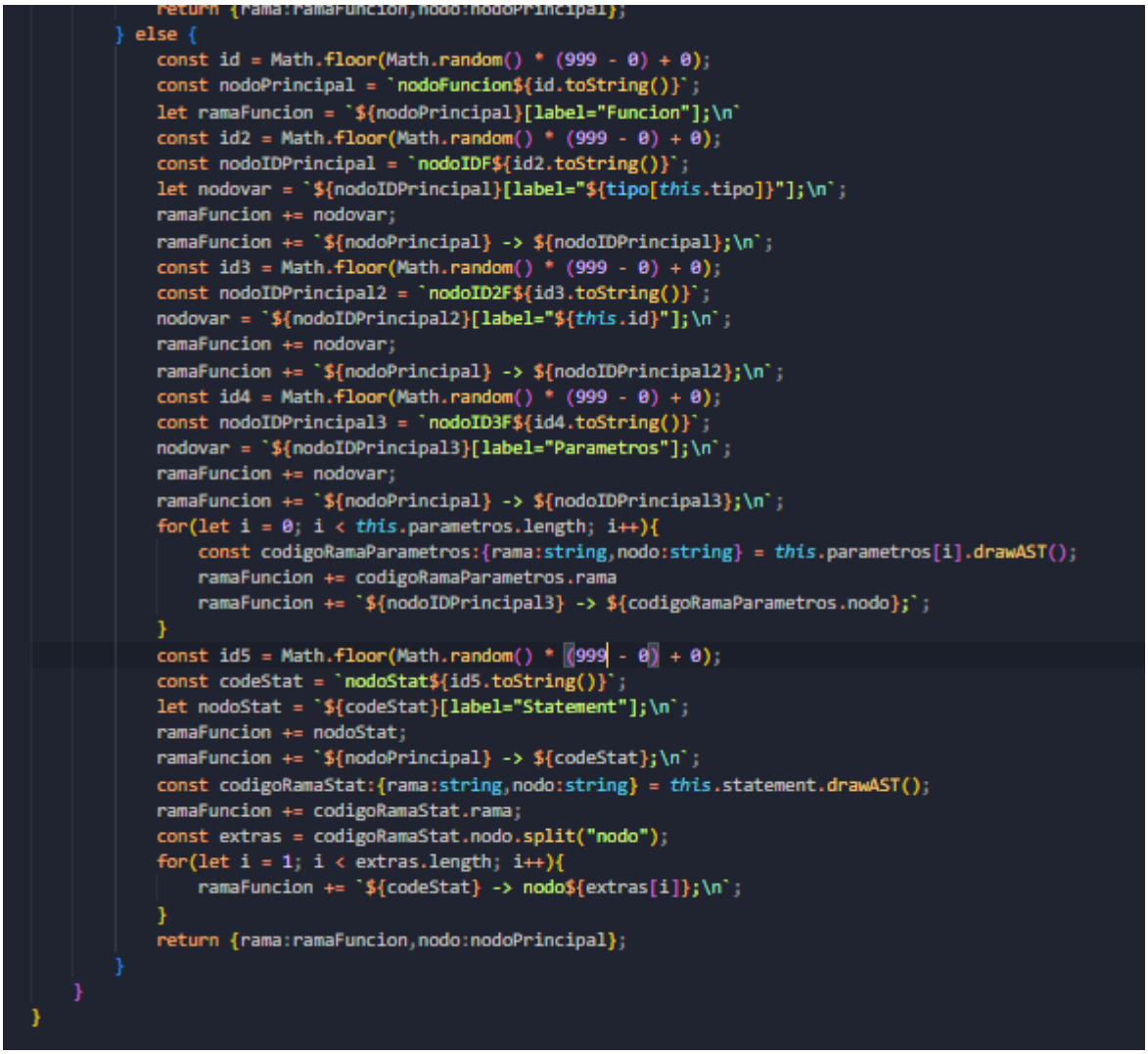
```
import { Environment } from "../abstract/Environment";
import { Instruction } from "../abstract/Instruction";

export class Default extends Instruction{
    constructor(linea:number, columna:number){
        super(linea,columna);
    }

    public execute(env: Environment) {
        return this;
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999-0) + 0);
        let nombreNodo = `nodoDefaultXD${id.toString()}`;
        let rama = `${nombreNodo}[label="Default"];`;
        return {rama:rama,nodo:nombreNodo};
    }
}
```

Funcion

Esta instrucción nos permitia guardar una función en el entorno global para poder llamarla posteriormente junto a sus parámetros y instrucciones.

```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression";
import { tipo } from "../abstract/Return";

export class Funcion extends Instruction{
    constructor(public tipo:tipo, private id:string, public parametros:Array<Expression>, public statement:Instruction, line:number, column:number){
        super(line, column);
    }

    public execute(env:Environment) {
        //save function in the environment
        env.guardarFuncion(this.id,this);
    }
```

```
public drawAST(): { rama: string; nodo: string; } {
    if(this.tipo == tipo.VOID){
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoMetodo${id.toString()}`;
        let ramaFuncion = `${nodoPrincipal}[label="Metodo"];\n`
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal = `nodoIDMet${id2.toString()}`;
        let nodovar = `${nodoIDPrincipal}[label="${tipo[this.tipo]}"];\n`;
        ramaFuncion += nodovar;
        ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
        const id3 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal2 = `nodoID2Met${id3.toString()}`;
        nodovar = `${nodoIDPrincipal2}[label="${this.id}"];\n`;
        ramaFuncion += nodovar;
        ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal2};\n`;
        const id4 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoIDPrincipal3 = `nodoID3Met${id4.toString()}`;
        nodovar = `${nodoIDPrincipal3}[label="Parametros"];\n`;
        ramaFuncion += nodovar;
        ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal3};\n`;
        for(let i = 0; i < this.parametros.length; i++){
            const codigoRamaParametros:{rama:string,nodo:string} = this.parametros[i].drawAST();
            ramaFuncion += codigoRamaParametros.rama
            ramaFuncion += `${nodoIDPrincipal3} -> ${codigoRamaParametros.nodo};`;
        }
        const id5 = Math.floor(Math.random() * (999 - 0) + 0);
        const codeStat = `nodoStat${id5.toString()}`;
        let nodoStat = `${codeStat}[label="Statement"];\n`;
        ramaFuncion += nodoStat;
        ramaFuncion += `${nodoPrincipal} -> ${codeStat};\n`;
        const codigoRamaStat:{rama:string,nodo:string} = this.statement.drawAST();
        ramaFuncion += codigoRamaStat.rama;
        const extras = codigoRamaStat.nodo.split("nodo");
        for(let i = 1; i < extras.length; i++){
            ramaFuncion += `${codeStat} -> nodo${extras[i]};\n`;
        }
        return {rama:ramaFuncion,nodo:nodoPrincipal};
```



```
            return {rama:ramaFuncion,nodo:nodoPrincipal};
        } else {
            const id = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoPrincipal = `nodoFuncion${id.toString()}`;
            let ramaFuncion = `${nodoPrincipal}[label="Funcion"];\n`
            const id2 = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoIDPrincipal = `nodoIDF${id2.toString()}`;
            let nodovar = `${nodoIDPrincipal}[label="${tipo[this.tipo]}"];\n`;
            ramaFuncion += nodovar;
            ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal};\n`;
            const id3 = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoIDPrincipal2 = `nodoID2F${id3.toString()}`;
            nodovar = `${nodoIDPrincipal2}[label="${this.id}"];\n`;
            ramaFuncion += nodovar;
            ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal2};\n`;
            const id4 = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoIDPrincipal3 = `nodoID3F${id4.toString()}`;
            nodovar = `${nodoIDPrincipal3}[label="Parametros"];\n`;
            ramaFuncion += nodovar;
            ramaFuncion += `${nodoPrincipal} -> ${nodoIDPrincipal3};\n`;
            for(let i = 0; i < this.parametros.length; i++){
                const codigoRamaParametros:{rama:string,nodo:string} = this.parametros[i].drawAST();
                ramaFuncion += codigoRamaParametros.rama
                ramaFuncion += `${nodoIDPrincipal3} -> ${codigoRamaParametros.nodo};`;
            }
            const id5 = Math.floor(Math.random() * (999 - 0) + 0);
            const codeStat = `nodoStat${id5.toString()}`;
            let nodoStat = `${codeStat}[label="Statement"];\n`;
            ramaFuncion += nodoStat;
            ramaFuncion += `${nodoPrincipal} -> ${codeStat};\n`;
            const codigoRamaStat:{rama:string,nodo:string} = this.statement.drawAST();
            ramaFuncion += codigoRamaStat.rama;
            const extras = codigoRamaStat.nodo.split("nodo");
            for(let i = 1; i < extras.length; i++){
                ramaFuncion += `${codeStat} -> nodo${extras[i]};\n`;
            }
            return {rama:ramaFuncion,nodo:nodoPrincipal};
        }
    }
}
```

IBreak

Esta instrucción nos permite obtener una instancia de la sentencia de transferencia break.

```
import { Environment } from "../abstract/Environment";
import { Instruction } from "../abstract/Instruction";
import { tipo } from "../abstract/Return";

export class IBreak extends Instruction{
    constructor(line: number, column: number){
        super(line, column);
    }

    public execute(env: Environment) {
        return this
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999-0) + 0);
        let nombreNodo = `nodoBreak${id.toString()}\n`;
        let rama = `${nombreNodo}[label="Break"];\n`;
        return {rama:rama,nodo:nombreNodo};
    }
}
```

IContinue

Esta instrucción nos permite obtener una instancia de la sentencia de transferencia continue.

```
import { Environment } from "../abstract/Environment";
import { Instruction } from "../abstract/Instruction";
import { tipo } from "../abstract/Return";

export class IContinue extends Instruction{
    constructor(line: number, column: number){
        super(line, column);
    }

    public execute(env: Environment) {
        return this;
    }

    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999-0) + 0);
        let nombreNodo = `nodoContinue${id.toString()}`;
        let rama = `${nombreNodo}[label="Continue"];`;
        return {rama:rama,nodo:nombreNodo};
    }
}
```

InsDoWhile

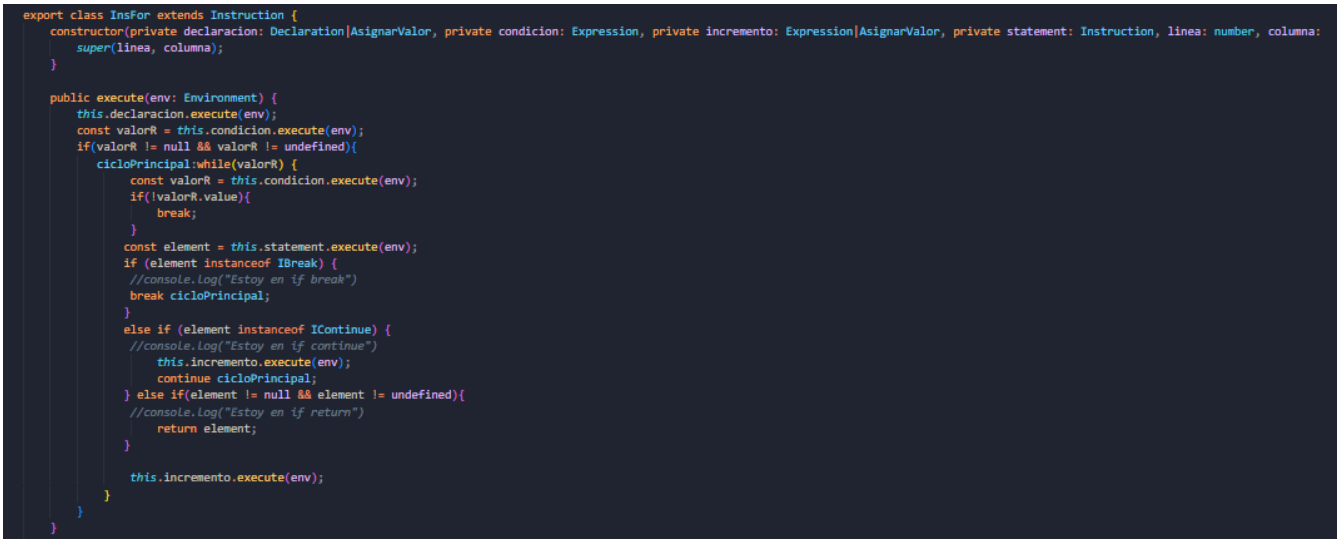
Esta clase nos permite manejar la sentencia cíclica do While verificando por medio de este mismo ciclo la condición del do while para que se siga ejecutando o se detenga.

```ts
import { Instruction } from "../abstract/Instruction";
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { IBreak } from "./IBreak";
import { IContinue } from "./IContinue";

export class InsDoWhile extends Instruction {
    constructor(private condition: Expression, private statement: Instruction, line: number, column: number) {
        super(line, column);
    }

    public execute(env: Environment) {
        let valorR = this.condition.execute(env);
        cicloPrincipal: do {
            const element = this.statement.execute(env);
            if (element instanceof IBreak) {
              // console.log("Estoy en if break")
               break cicloPrincipal;
              }
              else if (element instanceof IContinue) {
               //console.log("Estoy en if continue")
                  valorR = this.condition.execute(env);
                   continue cicloPrincipal;
              } else if(element != null && element != undefined){
              // console.log("Estoy en if return")
                   return element;
              }
            valorR = this.condition.execute(env);
        } while(valorR.value);
    }
```

```ts
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoDoWhile${id.toString()}`;
        let ramaDoWhile = `${nombreNodo}[label="DoWhile"];`;
        const id4 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoDo = `nodeDo${id4.toString()}`;
        let ramaDo = `${nodoDo}[label="Do"];`;
        ramaDoWhile += ramaDo;
        ramaDoWhile += `${nombreNodo} -> ${nodoDo};\n`;
        const codeRamaI : {rama:string, nodo:string} = this.statement.drawAST();
        const id3 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoSta = `nodeStatementWh${id3.toString()}`
        let ramaStatement = `${nodoSta}[label="Statement"];\n`;
        ramaDoWhile += ramaStatement;
        ramaDoWhile += `${nodoDo} -> ${nodoSta};\n`;
        ramaDoWhile += codeRamaI.rama;
        const ramaExtra = codeRamaI.nodo.split("nodo");
        for(let i = 1; i < ramaExtra.length; i++){
            ramaDoWhile += `${nodoSta} -> nodo${ramaExtra[i]};\n`;
        }
        const id5 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoWhile = `nodeWhile${id5.toString()}`;
        let ramaWhile = `${nodoWhile}[label="While_DoWhile"];\n`;
        ramaDoWhile += ramaWhile;
        ramaDoWhile += `${nombreNodo} -> ${nodoWhile};\n`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoCOndicion = `nodeConditionW${id2.toString()}`;
        let ramaCondicion = `${nodoCOndicion}[label="Condicion"];\n`;
        ramaDoWhile += ramaCondicion;
        ramaDoWhile += `${nodoWhile} -> ${nodoCOndicion};\n`;
        const codigoRama2 : {rama:string, nodo:string} = this.condition.drawAST();
        ramaDoWhile += codigoRama2.rama;
        ramaDoWhile += `${nodoCOndicion} -> ${codigoRama2.nodo};\n`;
        return {rama:ramaDoWhile,nodo:nombreNodo};
    }
}
```

InsFor

Esta clase nos permite manejar el ciclo For en este se debe realizar varias cosas durante el ciclo while usado como ejecución de este ya que debe realizarse un incremento una asignación y la operación relacional para que este pueda funcionar correctamente, también se debe observar si no se tiene una sentencia de transferencia.



```
export class InsFor extends Instruction {
    constructor(private declaracion: Declaration|AsignarValor, private condicion: Expression, private incremento: Expression|AsignarValor, private statement: Instruction, linea: number, columna:
        super(linea, columna);
    }

    public execute(env: Environment) {
        this.declaracion.execute(env);
        const valorR = this.condicion.execute(env);
        if(valorR != null && valorR != undefined){
            cicloPrincipal:while(valorR) {
                const valorR = this.condicion.execute(env);
                if(!valorR.value){
                    break;
                }
                const element = this.statement.execute(env);
                if (element instanceof IBreak) {
                //console.Log("Estoy en if break")
                break cicloPrincipal;
                }
                else if (element instanceof IContinue) {
                //console.Log("Estoy en if continue")
                    this.incremento.execute(env);
                    continue cicloPrincipal;
                } else if(element != null && element != undefined){
                //console.Log("Estoy en if return")
                    return element;
                }

                this.incremento.execute(env);
            }
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoFor${id.toString()}`;
        let ramaFor = `${nombreNodo}[label="For"];`;
        const codigoRama : {rama:string, nodo:string} = this.declaracion.drawAST();
        ramaFor += codigoRama.rama;
        ramaFor += `${nombreNodo} -> ${codigoRama.nodo};\n`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoCOndicion = `nodeConditionFOr${id2.toString()}`;
        let ramaCondicion = `${nodoCOndicion}[label="Condicion"];\n`;
        ramaFor += ramaCondicion;
        ramaFor += `${nombreNodo} -> ${nodoCOndicion};\n`;
        const codigoRama2 : {rama:string, nodo:string} = this.condicion.drawAST();
        ramaFor += codigoRama2.rama;
        ramaFor += `${nodoCOndicion} -> ${codigoRama2.nodo};\n`;
        const codigoRama3 : {rama:string, nodo:string} = this.incremento.drawAST();
        ramaFor += codigoRama3.rama;
        ramaFor += `${nombreNodo} -> ${codigoRama3.nodo};\n`;

        const codeRamaI : {rama:string, nodo:string} = this.statement.drawAST();
        const id3 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoSta = `nodeStatementFor${id3.toString()}`
        let ramaStatement = `${nodoSta}[label="Statement"];\n`;
        ramaFor += ramaStatement;
        ramaFor += `${nombreNodo} -> ${nodoSta};\n`;
        ramaFor += codeRamaI.rama;
        const ramaExtra = codeRamaI.nodo.split("nodo");
        for(let i = 1; i < ramaExtra.length; i++){
            ramaFor += `${nodoSta} -> nodo${ramaExtra[i]};\n`;
        }
        return {rama:ramaFor,nodo:nombreNodo};
    }
}
```

InsIf

Esta clase se uso para manejar el ciclo if en el cual se debe revisar cosas como una condición y si después de este se tienen posibles sentencias else if o else aunque estas no son indispensables por lo que también pueden llegar a ser nulas.

```
export class InsIf extends Instruction {
    constructor(private condicion: Expression, private statement: Instruction, private inElse: Instruction, linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env: Environment) {
        const valorR = this.condicion.execute(env);
        if(valorR.value) {
            return this.statement.execute(env);
        } else if(this.inElse != null) {
            return this.inElse.execute(env);
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoIF${id.toString()}`;
        let ramaIf = `${nombreNodo}[label="IF"];`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoCOndicion = `nodeConditionIf${id2.toString()}`;
        let ramaCondicion = `${nodoCOndicion}[label="Condicion"];\n`;
        ramaIf += ramaCondicion;
        ramaIf += `${nombreNodo} -> ${nodoCOndicion};\n`;
        const codigoRama2 : {rama:string, nodo:string} = this.condicion.drawAST();
        ramaIf += codigoRama2.rama;
        ramaIf += `${nodoCOndicion} -> ${codigoRama2.nodo};\n`;

        const codeRamaI : {rama:string, nodo:string} = this.statement.drawAST();
        const id3 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoSta = `nodeStatementIF${id3.toString()}`
        let ramaStatement = `${nodoSta}[label="Statement"];\n`;
        ramaIf += ramaStatement;
        ramaIf += `${nombreNodo} -> ${nodoSta};\n`;
        ramaIf += codeRamaI.rama;
        const ramaExtra = codeRamaI.nodo.split("nodo");
        for(let i = 1; i < ramaExtra.length; i++){
            ramaIf += `${nodoSta} -> nodo${ramaExtra[i]};\n`;
        }
        if(this.inElse != null) {
            const codeRamaI2 : {rama:string, nodo:string} = this.inElse.drawAST();
            const id2 = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoSta2 = `nodeStatementIF${id2.toString()}`
            let ramaStatement2 = `${nodoSta2}[label="Statement"];\n`;
            ramaIf += ramaStatement2;
            ramaIf += `${nombreNodo} -> ${nodoSta2};\n`;
            ramaIf += codeRamaI2.rama;
            const ramaExtra2 = codeRamaI2.nodo.split("nodo");
            for(let i = 1; i < ramaExtra2.length; i++){
                ramaIf += `${nodoSta2} -> nodo${ramaExtra2[i]};\n`;
            }
        }
        return {rama:ramaIf,nodo:nombreNodo};
    }
```

InsSwitch

Esta clase manejaba la sentencia switch en la que se tiene una condición y una lista de casos que deben ser verificados hasta que se cumpla alguno o se tenga un break.

```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression";
import { IBreak } from "./IBreak";
import { Default } from "./Default";

export class InsSwitch extends Instruction {
    constructor(private condicion: Expression, private casos:any,linea: number, columna: number) {
        super(linea, columna);
    }

    public execute(env: Environment) {
        const condicionevaluar = this.condicion.execute(env)
        cicloP:for (var i = 0; i < this.casos.length; i++) {
            var bandera_break = false;
            const aux = this.casos[i];
            const valorcondicion = aux[0].execute(env);
            if(condicionevaluar.value == valorcondicion.value || valorcondicion instanceof Default){
                const auxval = aux[1];
                const retorno =auxval.execute(env);
                if(retorno != undefined || retorno != null){
                if(retorno instanceof IBreak){
                    bandera_break = true;
                }
                }
            }
            if(bandera_break){
                break cicloP;
            }
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoSwitch${id.toString()}`;
        let ramaWhile = `${nombreNodo}[label="Switch"];`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoCOndicion = `nodeConditionW${id2.toString()}`;
        let ramaCondicion = `${nodoCOndicion}[label="Condicion"];\n`;
        ramaWhile += ramaCondicion;
        ramaWhile += `${nombreNodo} -> ${nodoCOndicion};\n`;
        const codigoRama2 : {rama:string, nodo:string} = this.condicion.drawAST();
        ramaWhile += codigoRama2.rama;
        ramaWhile += `${nodoCOndicion} -> ${codigoRama2.nodo};\n`;

        for(let i=0;i<this.casos.length;i++){
            const aux = this.casos[i];
            const id3 = Math.floor(Math.random() * (999 - 0) + 0);
            const nodoCase = `nodeCaseW${id3.toString()}`;
            let ramaCase = `${nodoCase}[label="Case"];\n`;
            ramaWhile += ramaCase;
            ramaWhile += `${nombreNodo} -> ${nodoCase};\n`;
            const valorcondicion = aux[0].drawAST();
            ramaWhile += valorcondicion.rama;
            ramaWhile += `${nodoCase} -> ${valorcondicion.nodo};\n`;
        }
        return {rama:ramaWhile,nodo:nombreNodo};
    }
}
```

InsWhile

Esta clase nos permite manejar el ciclo while en el que solo vamos revisando una condición hasta que ya no se cumpla y se rompa dicho ciclo a menos que se encuentre una sentencia de transferencia.

```
import { Instruction } from "../abstract/Instruction";
import { Expression } from "../abstract/Expression";
import { Environment } from "../abstract/Environment";
import { IBreak } from "./IBreak";
import { IContinue } from "./IContinue";
import { Incremento } from "../Expressions/Incremento";
import { Decremento } from "../Expressions/Decremento";

export class InWhile extends Instruction {
    constructor(private condition: Expression, private statement: Instruction, line: number, column: number) {
        super(line, column);
    }

    public execute(env: Environment) {
        let valorR = this.condition.execute(env);
        cicloPrincipal:while(valorR.value) {
            const element = this.statement.execute(env);
            if (element instanceof IBreak) {
                //console.log("Estoy en if break")
                break cicloPrincipal;
               }
               else if (element instanceof IContinue) {
               // console.log("Estoy en if continue")
                   valorR = this.condition.execute(env);
                    continue cicloPrincipal;
               } else if(element != null && element != undefined){
               // console.log("Estoy en if return")
                    return element;
               }
            valorR = this.condition.execute(env);
        }
    }
```

```
    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nombreNodo = `nodoWhile${id.toString()}`;
        let ramaWhile = `${nombreNodo}[label="While"];`;
        const id2 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoCOndicion = `nodeConditionW${id2.toString()}`;
        let ramaCondicion = `${nodoCOndicion}[label="Condicion"];\n`;
        ramaWhile += ramaCondicion;
        ramaWhile += `${nombreNodo} -> ${nodoCOndicion};\n`;
        const codigoRama2 : {rama:string, nodo:string} = this.condition.drawAST();
        ramaWhile += codigoRama2.rama;
        ramaWhile += `${nodoCOndicion} -> ${codigoRama2.nodo};\n`;

        const codeRamaI : {rama:string, nodo:string} = this.statement.drawAST();
        const id3 = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoSta = `nodeStatementWh${id3.toString()}`
        let ramaStatement = `${nodoSta}[label="Statement"];\n`;
        ramaWhile += ramaStatement;
        ramaWhile += `${nombreNodo} -> ${nodoSta};\n`;
        ramaWhile += codeRamaI.rama;
        const ramaExtra = codeRamaI.nodo.split("nodo");
        for(let i = 1; i < ramaExtra.length; i++){
            ramaWhile += `${nodoSta} -> nodo${ramaExtra[i]};\n`;
        }
        return {rama:ramaWhile,nodo:nombreNodo};
    }
}
```

Main

Esta clase permit el manejo de la sentencia main, que es la sentencia con la que se inicia el programa esta consta de un llamado a una función que se ejecutara.

```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { LlamadaFuncion } from "../Expressions/LlamadaFuncion";

export class InsMain extends Instruction{
    constructor(public funcion: LlamadaFuncion, linea:number, columna:number){
        super(linea, columna);
    }

    public execute(env:Environment) {
        this.funcion.execute(env);
    }


    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoMain${id.toString()}`;
        const nodoFuncion = this.funcion.drawAST();
        let ramaMain = `${nodoPrincipal}[label="Main"];\n`
        ramaMain += `${nodoPrincipal} -> ${nodoFuncion.nodo};\n`;
        ramaMain += nodoFuncion.rama;
        return {rama:ramaMain,nodo:nodoPrincipal};
    }
}
```

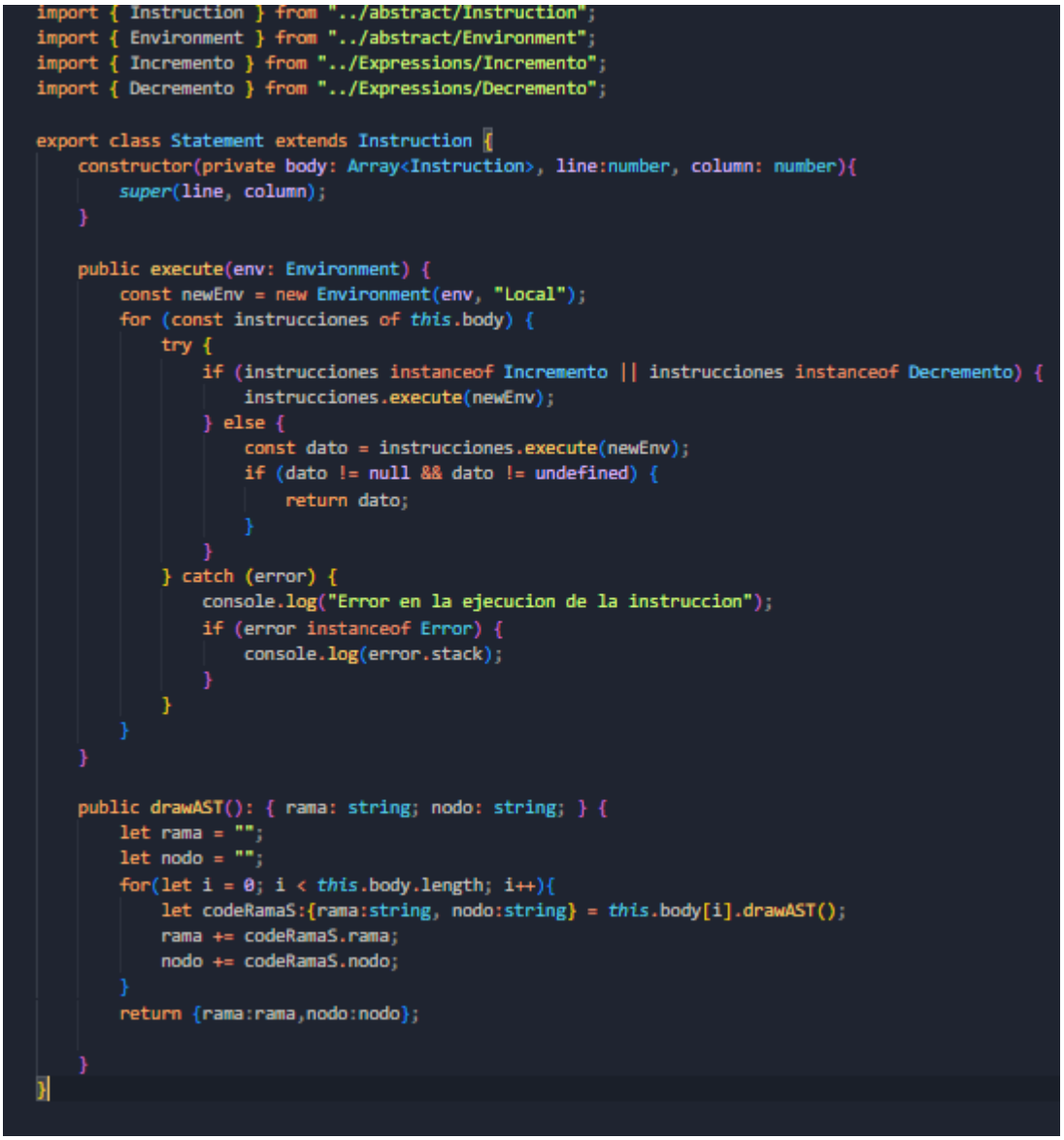
Print

Esta clase nos permite manejar la sentencia print en nuestro programa enviando cada cosa que aparezca en una de estas sentencias a una lista que posteriormente se enviara a la consola vista por el cliente.

```
import { Environment } from "../abstract/Environment";
import { Expression } from "../abstract/Expression"
import { Instruction } from "../abstract/Instruction"
import { printList } from "../reports/PrintList";

export class Print extends Instruction {
    constructor(line: number, column: number, private expression: Expression) {
        super(line, column);
    }

    public execute(env: Environment): void {
       const value = this.expression.execute(env);
       printList.push(value.value)
        console.log("Desde CMD: " ,value.value);
    }


    public drawAST(): { rama: string; nodo: string; } {
        const id = Math.floor(Math.random() * (999 - 0) + 0);
        const nodoPrincipal = `nodoPrint${id.toString()}`;
        let ramaPrint = `${nodoPrincipal}[label="Print"];\n`
        const codigoRama:{rama:string,nodo:string} = this.expression.drawAST();
        ramaPrint += codigoRama.rama;
        ramaPrint += `${nodoPrincipal} -> ${codigoRama.nodo};\n`
        return {rama:ramaPrint,nodo:nodoPrincipal};
    }
```

Statement

Esta clase permite el manejo de todas las sentencias que se utilizan en las funciones o en los ciclos es decir maneja todo lo que sucede en el espacio entre llaves de cada uno de estos.



```
import { Instruction } from "../abstract/Instruction";
import { Environment } from "../abstract/Environment";
import { Incremento } from "../Expressions/Incremento";
import { Decremento } from "../Expressions/Decremento";

export class Statement extends Instruction {
    constructor(private body: Array<Instruction>, line:number, column: number){
        super(line, column);
    }

    public execute(env: Environment) {
        const newEnv = new Environment(env, "Local");
        for (const instrucciones of this.body) {
            try {
                if (instrucciones instanceof Incremento || instrucciones instanceof Decremento) {
                    instrucciones.execute(newEnv);
                } else {
                    const dato = instrucciones.execute(newEnv);
                    if (dato != null && dato != undefined) {
                        return dato;
                    }
                }
            } catch (error) {
                console.log("Error en la ejecucion de la instruccion");
                if (error instanceof Error) {
                    console.log(error.stack);
                }
            }
        }
    }

    public drawAST(): { rama: string; nodo: string; } {
        let rama = "";
        let nodo = "";
        for(let i = 0; i < this.body.length; i++){
            let codeRamaS:{rama:string, nodo:string} = this.body[i].drawAST();
            rama += codeRamaS.rama;
            nodo += codeRamaS.nodo;
        }
        return {rama:rama,nodo:nodo};

    }
}
```

Reportes Generados

El ejecutar nuestro programa nos permite obtener 3 reportes diferentes para el usuario:

- AST
- Tabla de Símbolos
- Reporte de Errores

Cada reporte se genera en el momento en el que el usuario usa el botón de Ejecutar. El contenido del archivo se procesa mediante la librería de análisis léxico y sintáctico Jison; la cual tiene como trabajo ver que la estructura de nuestro archivo sea correcta y entendible para su flujo.

AST:

Para el ast se utilizó la función drawAST en cada una de las expresiones y instrucciones del programa para posteriormente terminar de obtener el AST con una pequeña serie de instrucciones en el controlador de nuestro Backend. El AST es un grafo que nos permite ver el flujo en el que se analiza nuestro programa.

```
let drawast = `
digraph AST {
    nodoPrincipal[label="AST"];\n
`;
for(const inst of ast){
    const dAST = inst.drawAST();
    drawast += `${dAST.rama}\n`;
    drawast += `nodoPrincipal -> ${dAST.nodo};`;
}

drawast += "\n}";
```

Ejemplo AST:

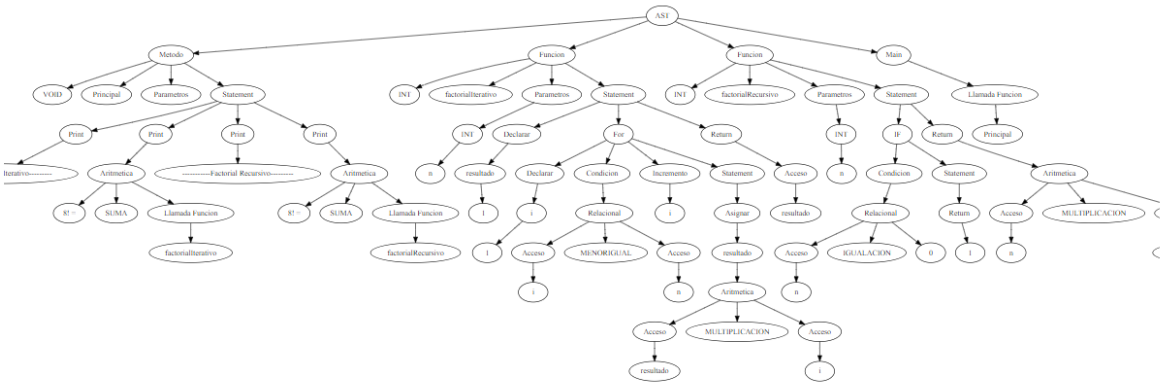


Tabla de Simbolos

Para la tabla de símbolos se utiliza un arreglo al cual se le añaden todas las variables, funciones y métodos que se están guardando por medio de las funciones que guardan dichas dentro del Environment, en ese espacio se envia datos como el tipo, si es una variable función o método, su entorno es decir es local o es global y también la línea y columna donde se encuentra.

```
export class TablaSimbolos {

    constructor(public id:string, public tipo:string,public tipoPrimitivo:string,public entorno:string, public linea:string, public columna:string) {

    }
}

export let ListaTabla:Array<TablaSimbolos> = [];
```

```
let tablaSimbol = `digraph TablaSimbolos {\n
    parent [shape = none\n
    label=<\n
    <table border= '1' cellspacing="0">\n
    <tr><td bgcolor="#FFD352">Identificador</td><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Entorno</td><td bgcolor="#FFD352">Linea</td><t
    const idsExistentes = new Set();
    for(const fila of ListaTabla){
        if(!idsExistentes.has(fila.id)){
            tablaSimbol += `<tr><td>${fila.id}</td><td>${fila.tipo}</td><td>${fila.tipoPrimitivo}</td><td>${fila.entorno}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\
            idsExistentes.add(fila.id);
        }
    }
    tablaSimbol += `</table>>];\n}`;
```

```
public guardar(id: string, valor:any, tipo:tipo, linea:number,columna:number) {
    let env: Environment | null = this;
    let tipon:string = "";
    if(!env.variables.has(id.toLowerCase())) {
        env.variables.set(id.toLowerCase(), new Symbol(valor,id,tipo));
        if(tipo == 0){
            tipon = "int";
        } else if(tipo == 1){
            tipon = "double";
        } else if(tipo == 2){
            tipon = "boolean";
        } else if(tipo == 3){
            tipon = "char";
        } else if(tipo == 4){
            tipon = "string";
        }
        ListaTabla.push(new TablaSimbolos(id.toLowerCase(),"variable",tipon,env.nombreEntorno,linea.toString(),columna.toString()));
    } else {
        printList.push("Error, la variable " + id + " ya existe en el entorno, linea " + linea + " y columna " + columna);
    }
}
```

```
public guardarFuncion(id:string,funcion:Funcion) {
    let tipo:string = "";
    let tipo2:string = "";
    let env:Environment | null = this;
    if(!env.funciones.has(id.toLowerCase())) {
        env.funciones.set(id.toLowerCase(),funcion);
        if(funcion.tipo == 0){
            tipo = "int";
            tipo2 = "funcion";
        } else if(funcion.tipo == 1){
            tipo = "double";
            tipo2 = "funcion";
        } else if(funcion.tipo == 2){
            tipo = "boolean";
            tipo2 = "funcion";
        } else if(funcion.tipo == 3){
            tipo = "char";
            tipo2 = "funcion";
        } else if(funcion.tipo == 4){
            tipo = "string";
            tipo2 = "funcion";
        } else if(funcion.tipo == 6){
            tipo = "void";
            tipo2 = "metodo";
        }

        ListaTabla.push(new TablaSimbolos(id.toLowerCase(),tipo2,tipo,env.nombreEntorno,funcion.line.toString(),funcion.column.toString()));
    } else {
        printList.push("Error, la funcion " + id + " ya existe en el entorno");
    }
}
```

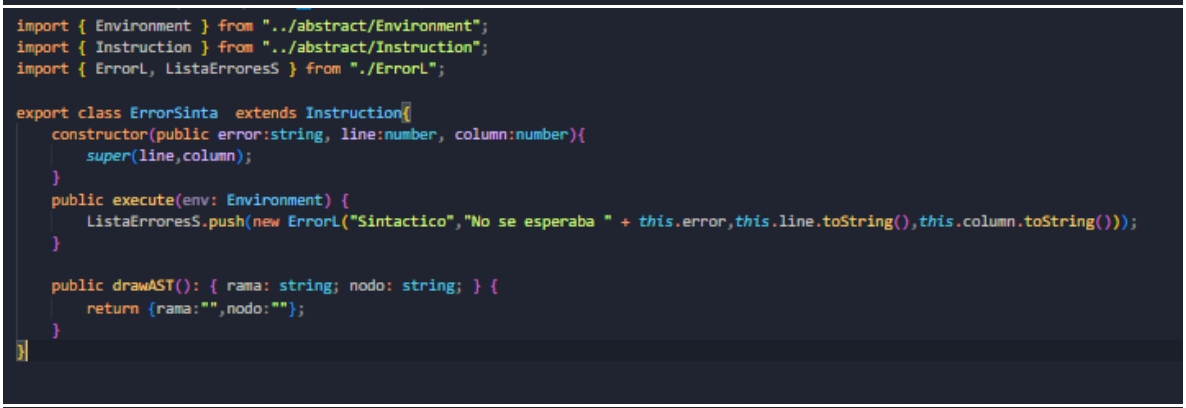
Ejemplo Tabla de Simbolos:

Identificador	Tipo	Tipo	Entorno	Linea	Columna
principal	metodo	void	Global	1	0
factorialiterativo	funcion	int	Global	8	0
factorialrecursivo	funcion	int	Global	16	0
n	variable	int	Parametro	3	20
resultado	variable	int	Local	9	4
i	variable	int	Local	10	9

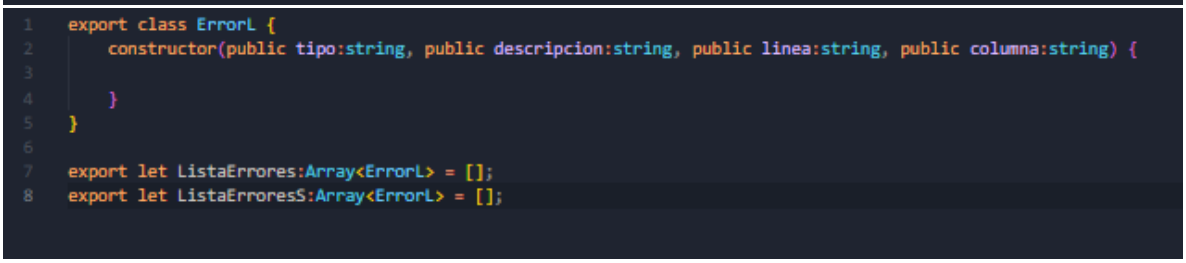
Tabla de Errores:

¨Para la creación del reporte de errores se utilizarón dos clases una para manejar errores léxicos y otra para manejar errores sintácticos, posteriormente se utilizarón dos listas de una clase ErroL para almacenar finalmente estos errores para poder enviarlos tanto por consola como en el reporte. Los errores son añadidos a cada una de estas desde el archivo .jison es decir desde el analizador léxico y sintactico.

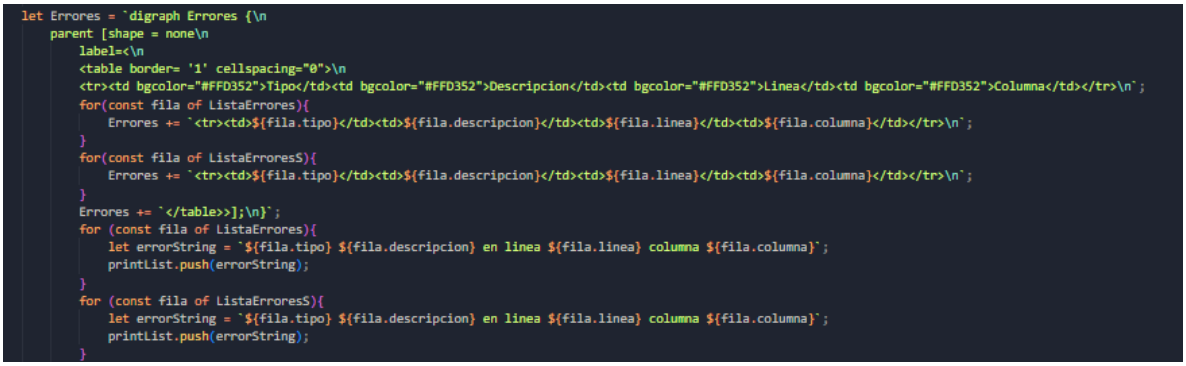
```
import { Instruction } from "../abstract/Instruction";
import { ErrorL,ListaErrores } from "./ErrorL";

export class ErrorLexico extends Instruction{
    constructor(public error:string ,line:number, column:number){
        super(line,column);
    }

    public execute() {

        ListaErrores.push(new ErrorL("Léxico", "El caracter "+this.error+" no pertenece al lenguaje",  this.line.toString(), this.column.toString() ));

    }
    public drawAST(): {rama: string, nodo:string} {

        return {rama: "", nodo:""}
    }

}
```



```
import { Environment } from "../abstract/Environment";
import { Instruction } from "../abstract/Instruction";
import { ErrorL, ListaErroresS } from "./ErrorL";

export class ErrorSinta  extends Instruction{
    constructor(public error:string, line:number, column:number){
        super(line,column);
    }
    public execute(env: Environment) {
        ListaErroresS.push(new ErrorL("Sintactico","No se esperaba " + this.error,this.line.toString(),this.column.toString()));
    }

    public drawAST(): { rama: string; nodo: string; } {
        return {rama:"",nodo:""};
    }
}
```



```
export class ErrorL {
    constructor(public tipo:string, public descripcion:string, public linea:string, public columna:string) {


    }
}

export let ListaErrores:Array<ErrorL> = [];
export let ListaErroresS:Array<ErrorL> = [];
```



```
let Errores = `digraph Errores {\n
    parent [shape = none\n
        label=<\n
        <table border= '1' cellspacing="0">\n
        <tr><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Descripcion</td><td bgcolor="#FFD352">Linea</td><td bgcolor="#FFD352">Columna</td></tr>\n`;
        for(const fila of ListaErrores){
            Errores += `<tr><td>${fila.tipo}</td><td>${fila.descripcion}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\n`;
        }
        for(const fila of ListaErroresS){
            Errores += `<tr><td>${fila.tipo}</td><td>${fila.descripcion}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\n`;
        }
        Errores += `</table>>];\n}`;
        for (const fila of ListaErrores){
            let errorString = `${fila.tipo} ${fila.descripcion} en linea ${fila.linea} columna ${fila.columna}`;
            printList.push(errorString);
        }
        for (const fila of ListaErroresS){
            let errorString = `${fila.tipo} ${fila.descripcion} en linea ${fila.linea} columna ${fila.columna}`;
            printList.push(errorString);
        }
```

Ejemplo Reporte Errores

Editor

```
void Principal(){
    print("----------Factorial Iterativo---------
    print("8! = " + factorialIterativo(8));
    print("----------Factorial Recursivo---------
    print("8! = " + factorialRecursivo(8));
}
á
int factorialIterativo(int n){
    int resultado = 1;
    for (int i = 1; i <= n; i++) {
        resultado = resultado * i;
    }
    return resultado
```

Consola

```
----------Factorial Iterativo---------
----------Factorial Recursivo---------
8! = 40320
Léxico El caracter á no pertenece al lenguaje en linea 7 c
Sintactico No se esperaba resultado en linea 14 columna 0
```

Tipo	Descripcion	Linea	Columna
Léxico	El caracter á no pertenece al lenguaje	7	0
Sintactico	No se esperaba resultado	14	0

Abrir

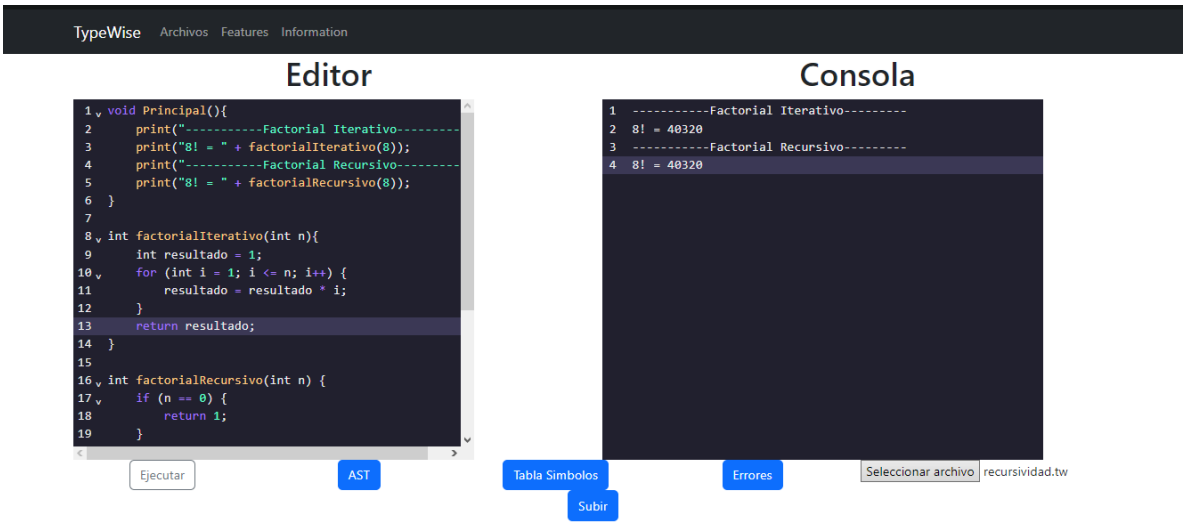
Este metodo se usaban para el manejo del archivo, esto por medio en todos los casos de el buscador de archivos que react brinda.

```
const handleFile = () => {
    const file = document.getElementById("file").files[0];
    if(file){
        const reader = new FileReader();
        reader.onload = function (e) {
            const contents = e.target.result;
            setEditor(contents);
            setCodigoEditor(contents);
        };
        reader.readAsText(file);
    }
}
```

```
<input type="file" id="file" accept=".tw"/>
```

Interfaz Utilizada

La interfaz se creo por medio del uso del framework React, el cual utiliza el lenguaje de programación JavaScript para ir generando diferentes módulos los cuales tienen un parecido bastante alto a html que son utilizados para ir generando cada una de las piezas que conformaran finalmente lo que es la interfaz de usuario. La interfaz corre en el localhost:3000



Servidor

El servidor se creo por medio de node Js y el uso de librerías como cors y express para poder generar un servidor local que en este caso es localhost:5000 y se tiene un solo endpoint que es localhost:5000/interprete/interpretar lo que permite realizar todo el proceso de análisis y ejecución del programa. Cors se utilizo para poder realizar la conexión de backend y frontend así como él envió de datos junto a la librería de axios.

```ts
import express from 'express';
import cors from 'cors';
import router from './routes/interprete'

const app = express();
const allowedOrigins = ['http://localhost:3000'];
const options: cors.CorsOptions = {
    origin: allowedOrigins
}
app.use(cors(options));
app.use(express.json());
const PORT = 5000;

// app.get('/ping', (req,res) => {
//     console.log('compiladores 1 si sale');
//     res.send('pongCompiladores1')
// });

app.use('/interprete', router);
```

```ts
import express from 'express';
import { interpreteController } from '../controller/controllerInterprete';
const router = express.Router();

router.get('/ping1', interpreteController.ping);
router.post('/interpretar', interpreteController.interpretar);

export default router;
```


kend > src > controller > controllerInterprete.ts > InterpreteController > interpretar
import { ListaTabla } from "./interprete/reports/TablaSimbolos";
import { ListaErrores, ListaErroresS } from "./interprete/reports/ErrorL";
import { Funcion } from "./interprete/Instructions/Funcion";
import { Declaration } from "./interprete/Instructions/Declaration";
import { InsMain } from "./interprete/Instructions/Main";
class InterpreteController {
//metodo ping
public ping(req: Request, res: Response) {
res.send("Pong interprete controller OLC1")
}
public interpretar(req: Request, res: Response) {
var parser = require("./interprete/gramatica");
ListaErrores.splice(0,ListaErrores.length);
const codigo = req.body.codigo;
console.log(codigo);
try {
const ast = parser.parse(codigo);
try {
printList.splice(0,printList.length);
ListaTabla.splice(0,ListaTabla.length);
ListaErroresS.splice(0,ListaErroresS.length);
const globalEnv = new Environment(null,"Global");
// for (const inst of ast) {
// inst.execute(globalEnv);
// }
for(const inst of ast){
if(inst instanceof Declaration){
inst.execute(globalEnv);
} else if(inst instanceof Funcion){
inst.execute(globalEnv);
}
}
for(const inst of ast){
if(inst instanceof InsMain){
inst.execute(globalEnv);
}
}
let drawast = `
digraph AST {
nodoPrincipal[label="AST"];\n
`;

```
        for(const inst of ast){
            const dAST = inst.drawAST();
            drawast += `${dAST.rama}\n`;
            drawast += `nodoPrincipal -> ${dAST.nodo};`;
        }

        drawast += "\n}";

        let tablaSimbol = `digraph TablaSimbolos {\n
            parent [shape = none\n
            label=<\n
            <table border= '1' cellspacing="0">\n
            <tr><td bgcolor="#FFD352">Identificador</td><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Entorno</td><td bgcolor="#FFD352">Linea</td><t
            const idsExistentes = new Set();
            for(const fila of ListaTabla){
                if(!idsExistentes.has(fila.id)){
                    tablaSimbol += `<tr><td>${fila.id}</td><td>${fila.tipo}</td><td>${fila.tipoPrimitivo}</td><td>${fila.entorno}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\
                    idsExistentes.add(fila.id);
                }
            }
            tablaSimbol += `</table>>];\n}`;
            console.log(ListaErrores.length)
        let Errores = `digraph Errores {\n
            parent [shape = none\n
                label=<\n
                <table border= '1' cellspacing="0">\n
                <tr><td bgcolor="#FFD352">Tipo</td><td bgcolor="#FFD352">Descripcion</td><td bgcolor="#FFD352">Linea</td><td bgcolor="#FFD352">Columna</td></tr>\n`;
                for(const fila of ListaErrores){
                    Errores += `<tr><td>${fila.tipo}</td><td>${fila.descripcion}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\n`;
                }
                for(const fila of ListaErroresS){
                    Errores += `<tr><td>${fila.tipo}</td><td>${fila.descripcion}</td><td>${fila.linea}</td><td>${fila.columna}</td></tr>\n`;
                }
                Errores += `</table>>];\n}`;
                for (const fila of ListaErrores){
                    let errorString = `${fila.tipo} ${fila.descripcion} en linea ${fila.linea} columna ${fila.columna}`;
                    printList.push(errorString);
                }
                for (const fila of ListaErroresS){
                    let errorString = `${fila.tipo} ${fila.descripcion} en linea ${fila.linea} columna ${fila.columna}`;
                    printList.push(errorString);
                }
        res.json({consola: printList.join("\n"), errores:Errores, ast: drawast, tablaSimbolos: tablaSimbol});
    } catch (error) {
        console.log(error);
```

```
                console.log(error);
                res.json({
                    consola:error,
                    errores: error,
                });
            }


        } catch (err) {
            console.log(err);
            res.json({
                consola: err,
                errores: err,
            });
        }
    }
}

export const interpreteController = new InterpreteController();
```

Recibimiento de datos en el Frontend con axios

```
const interpretar = async () => {
    console.log("ejecutando");
    try {
        setConsola("Ejecutando...");
        if (editor === "") {
            setConsola("No hay nada que ejecutar");
            console.log("No hay codigo a interpretar");
        } else {
            console.log(editor);
            const response = await axios.post('http://localhost:5000/interprete/interpretar', { codigo: editor });
            console.log(response.data);
            const { consola, errores, ast, tablaSimbolos } = response.data;
            console.log(consola);
            console.log("ast", ast);
            console.log("tablaS", tablaSimbolos);
            console.log("errores", errores);
            setDot(ast);
            setDot2(tablaSimbolos);
            setDot3(errores);
            setConsola(consola);
        }
    } catch (error) {
        console.log(error);
        setConsola("Error en Servidor");
    }
}
```