Manual técnico

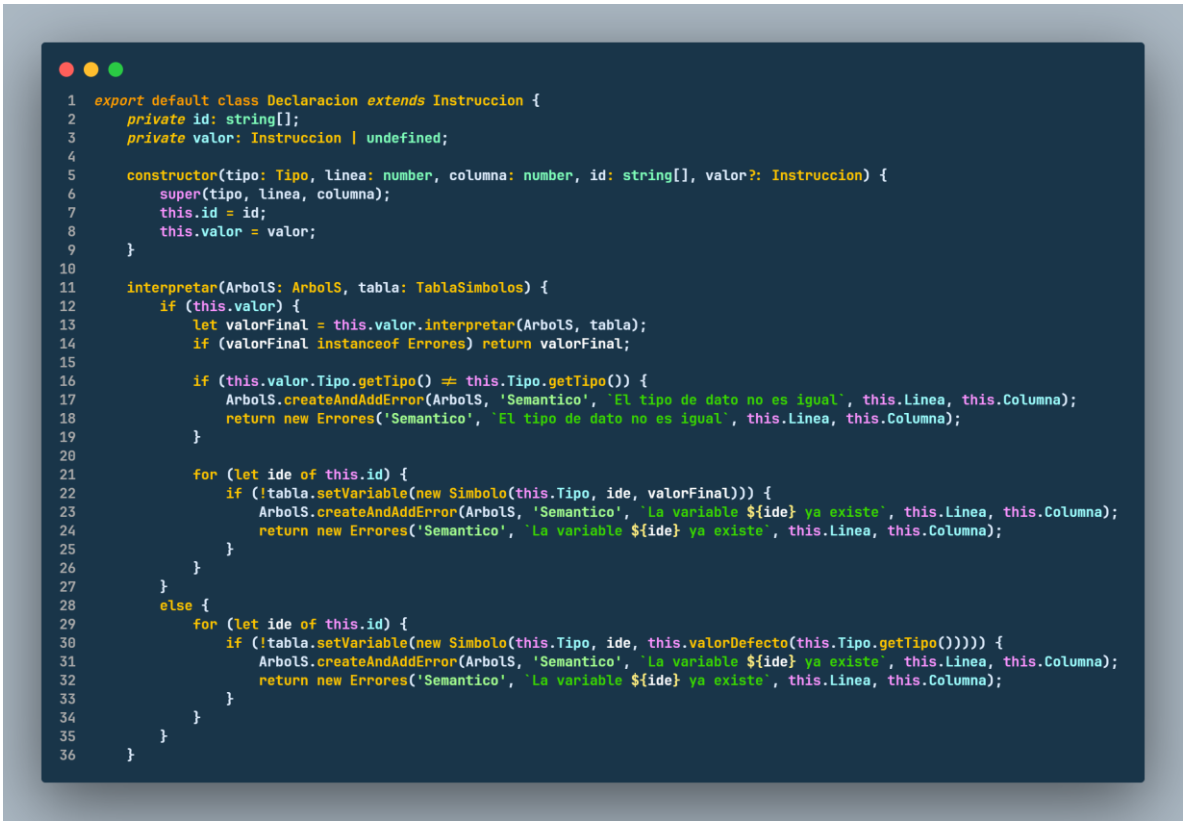
En este proyecto se creo un interprete para un lenguaje parecido a c, el cual cuenta con las siguientes funciones y características:

Declaración de variables

Acá se pueden declarar variables las cuales serán guardadas en un entorno de la función que se este manejando o si es en un ámbito global se guardaran en su dicha tabla de símbolos, la forma para declarar una variable es la siguiente.

```
declaracionv: tipo
            | tipo ids IGUAL
            | ids IGUAL
```

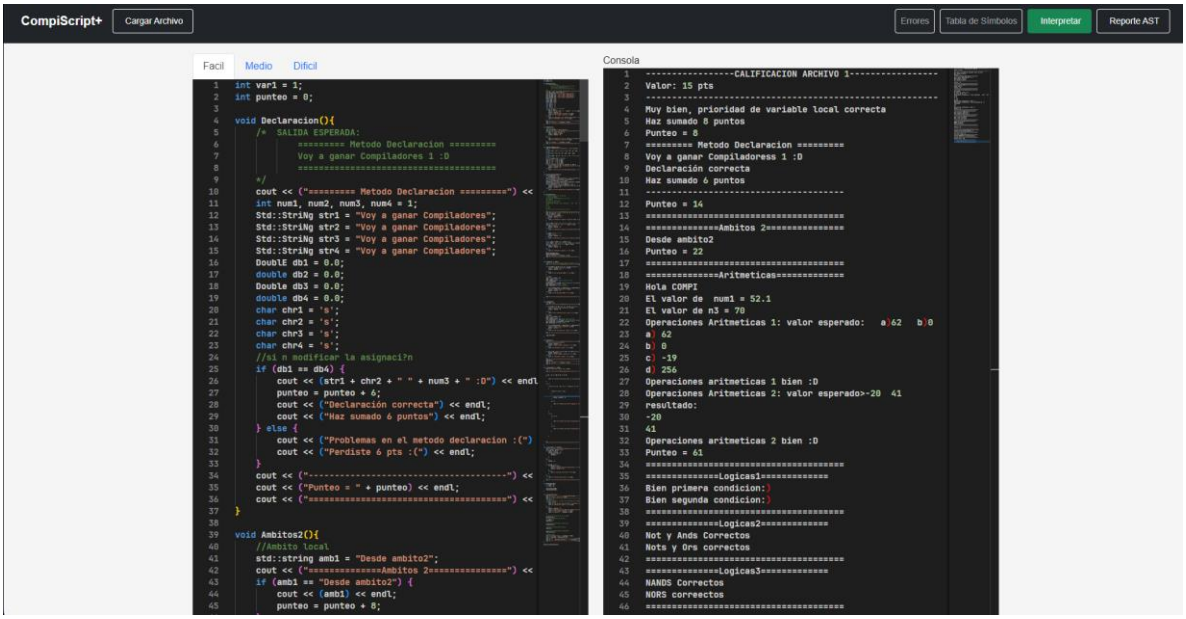
El cual puede venir con un su respectivo tipo su id o lista de ids, ya que se pueden declarar múltiples variables en una misma línea, y opcionalmente su valor que contendrán si no se le asigna valor se le colocarán unos valores por defecto



```
export default class Declaracion extends Instruccion {
    private id: string[];
    private valor: Instruccion | undefined;

    constructor(tipo: Tipo, linea: number, columna: number, id: string[], valor?: Instruccion) {
        super(tipo, linea, columna);
        this.id = id;
        this.valor = valor;
    }

    interpretar(ArbolS: ArbolS, tabla: TablaSimbolos) {
        if (this.valor) {
            let valorFinal = this.valor.interpretar(ArbolS, tabla);
            if (valorFinal instanceof Errores) return valorFinal;

            if (this.valor.Tipo.getTipo() != this.Tipo.getTipo()) {
                ArbolS.createAndAddError(ArbolS, 'Semantico', `El tipo de dato no es igual`, this.Linea, this.Columna);
                return new Errores('Semantico', `El tipo de dato no es igual`, this.Linea, this.Columna);
            }

            for (let ide of this.id) {
                if (!tabla.setVariable(new Simbolo(this.Tipo, ide, valorFinal))) {
                    ArbolS.createAndAddError(ArbolS, 'Semantico', `La variable ${ide} ya existe`, this.Linea, this.Columna);
                    return new Errores('Semantico', `La variable ${ide} ya existe`, this.Linea, this.Columna);
                }
            }
        }
        else {
            for (let ide of this.id) {
                if (!tabla.setVariable(new Simbolo(this.Tipo, ide, this.valorDefecto(this.Tipo.getTipo())))) {
                    ArbolS.createAndAddError(ArbolS, 'Semantico', `La variable ${ide} ya existe`, this.Linea, this.Columna);
                    return new Errores('Semantico', `La variable ${ide} ya existe`, this.Linea, this.Columna);
                }
            }
        }
    }
```

Como podemos observar en esta parte del código se hacen múltiples verificaciones para comprobar la lógica dentro del código que se esta ingresando, tales como verificar que el tipo

de dato y el valor que se le esta asignando son del mismo tipo o de que no se asignen variables repetidas dentro de un mismo ámbito



El proyecto también cuenta con un IDE donde se pueden manejar múltiples pestañas y un control sobre los errores, lo que se imprima en consola, la tabla de símbolos, cargar archivos, y el reporte del Árbol de sintaxis generado por la entrada a evaluar, este árbol se realizo de la siguiente manera

Se declaro una clase contador, del que utilizaremos el contador para los nodos con una misma instancia para así controlar cual seria el nodo anterior al que se le tiene que conectar

```typescript
export default class Contador {
    private static instance: Contador;
    private contador: number;
    private constructor() {
        this.contador = 0;
    }
    public static getInstance(): Contador {
        if (!Contador.instance) {
            Contador.instance = new Contador();
        }
        return Contador.instance;
    }

    get(){
        this.contador++;
        return this.contador;
    }
}
//Clase para tener un contador global, una unica instancia de la clase, la cual servira
//para tener un orden en los nodos y los links que se creen en el grafo
```

Luego se manda a llamar a la instancia de la clase dentro del método de cada instrucción llamado BuildAst

```
buildAst(anterior: string): string {
        let contador = Contador.getInstance();
        let nodoCaso = `n${contador.get()}`;
        let nodoCase = `n${contador.get()}`;
        let nodoExpresion = `n${contador.get()}`;
        let nodoDosPuntos = `n${contador.get()}`;
        let nodoCodigos = `n${contador.get()}`;
        let resultado = `${nodoCaso}[label="Caso"]\n`;
        resultado += `${nodoCase}[label="CASE"]\n`;
        resultado += `${nodoCaso} → ${nodoCase}\n`;
        resultado += `${nodoExpresion}[label="Expresion"]\n`;
        resultado += this.expresion.buildAst(nodoExpresion);
        resultado += `${nodoCaso} → ${nodoExpresion}\n`;
        resultado += `${nodoDosPuntos}[label=":"]\n`;
        resultado += `${nodoCaso} → ${nodoDosPuntos}\n`;
        resultado += `${nodoCodigos}[label="Instrucciones"]\n`;
        for (let instruccion of this.instrucciones) {
            resultado += instruccion.buildAst(nodoCodigos);
        }
        resultado += `${nodoCaso} → ${nodoCodigos}\n`;
        resultado += `${anterior} → ${nodoCaso}\n`;
        return resultado;

    }
```

Acá se construye el árbol sintáctico de manera que se crea un nodo por cada instrucción ingresada de la que se conectan los nodos de los terminales y no terminales que componen dicha expresión

Y de esta manera se obtiene un archivo .dot parecido al siguiente

```
digraph G {
n0[label="Inicio"]
nCodigo[label="Codigo"]
n0 → nCodigo
n25[label="Instruccion"]
nCodigo → n25
n26[label="Declaracion"]
n27[label="0"]
n26 → n27
n28[label="Ids"]
n26 → n28
n29[label="addafsdfasdf"]
n28 → n29
n25 → n26

}
```

En el cual se observa como correctamente crea las conexiones según el contador de la clase creada.

```typescript
import TablaSimbolos from "../SimboloC/TablaSimbolos";
import Tipo from '../SimboloC/Tipo';
import ArbolS from '../SimboloC/ArbolS';
//Esta clase nos servira para crear el arbol con los nodos de instruccion que contendran
//a su vez otros nodos de instruccion o nodos de expresion para leer e interpretar ese arbol
export abstract class Instruccion {
    public Tipo: Tipo;
    public Linea: number;
    public Columna: number;

    constructor(tipo: Tipo, linea: number, columna: number){
        this.Tipo = tipo;
        this.Linea = linea;
        this.Columna = columna;
    }

    abstract interpretar(ArbolS: ArbolS, tabla: TablaSimbolos): any;

    abstract buildAst(anterior: string): string;

}

```

Para cada funcionalidad de este proyecto se utilizó esta clase abstracta, en el que se aplico lo siguiente que es el patrón interprete

El patrón intérprete es un patrón de diseño de software que se utiliza para definir una gramática para un lenguaje y proporcionar un intérprete que interpreta las sentencias de ese lenguaje. Consiste en dividir un problema en una representación recursiva de la gramática del lenguaje y luego implementar un intérprete que pueda procesar estas sentencias según la gramática definida.

Esto es lo que la clase Instrucción se encarga de hacer, ya que tiene su constructor, y los métodos que interpretarán y harán las funciones del programa.

```
;
expresion : Casteos                         { $$ = $1; }
          | NUMERO                           { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.ENTERO), $1, @1.first_line, @1.first_column);}
          | DECIMAL                          { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.DECIMAL), $1, @1.first_line, @1.first_column);}
          | CADENA                           { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.CADENA), $1, @1.first_line, @1.first_column);}
          | TRUE                             { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.BOOLEANO), true, @1.first_line, @1.first_column);}
          | FALSE                            { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.BOOLEANO), false, @1.first_line, @1.first_column);}
          | ID                               { $$ = new AccesoVar.default($1, @1.first_line, @1.first_column);}
          | PARENTESISI expresion PARENTESISD   { $$ = $2; }
          | operacion                        { $$ = $1; }
          | CARACTER                         { $$ = new Nativo.default(new Tipo.default(Tipo.TipoDato.CARACTER), $1, @1.first_line, @1.first_column);}
          | ternaryOp                        { $$ = $1; }
          | operacionRelacional              { $$ = $1; }
          | accesoVector                     { $$ = $1; }
          | funcToLower                      { $$ = $1; }
          | funcToUpper                      { $$ = $1; }
          | funcionRound                     { $$ = $1; }
          | funcionLength                    { $$ = $1; }
          | funcionTypeOf                    { $$ = $1; }
          | funciontoString                  { $$ = $1; }
          | llamada                          { $$ = $1; } //lamada puede ser una expresion por que puede retornar un valor que sea asignado a una variable o sera retornado


;
```

Para mi gramática se a utilizado esta producción de expresión, la cual cuenta con los distintos tipos de datos o valores que pueden ser utilizados para, asignar datos a una variable, guardar en un arreglo, etc, ya que en si este cubre las variantes de lo que puede ser ingresado por el usuario, un ejemplo seria el siguiente

```
void metodo1(){
    //llamada del metodo
    figura1(10);
}
void figura1(int n){
        std:: 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
                {
                    cadenaFigura = cadenaFigura + ". ";
                }
                j=j+1;
            }
            cout << cadenaFigura << endl;
            i=i+1;
        }
        cout << "Si la figura es un corazón, tas bien :3"<< endl;
    }

execute metodo1();
```

En este método podemos observar la expresión que es lo que este contenido dentro del if, dentro de las impresiones etc.