Universidad de San Carlos de Guatemala

Facultad de Ingenieria

Organización de Lenguajes y Compiladores 1

Auxiliar: Daniel Acabal

Proyecto 2 CompiScript+

Manual Técnico

Helen Janet Rodas Castro

202200066

Primer Semestre

Guatemala 21 de abril del 2024

Introducción

El desarrollo de software es un campo amplio y multidisciplinario que requiere de una sólida comprensión de los fundamentos teóricos y prácticos de la informática. En particular, la construcción de compiladores, que son herramientas fundamentales en la creación de software, involucra la aplicación de conocimientos sobre análisis léxico, sintáctico y semántico.

En el contexto del curso de Organización de Lenguajes y Compiladores 1, ha puesto en marcha un nuevo proyecto, requerido por la Escuela de Ciencias y Sistemas de la Facultad de Ingeniería, que consiste en crear un lenguaje de programación para que los estudiantes del curso de Introducción a la Programación y Computación 1 aprendan a programar y tener conocimiento de todas las generalidades de un lenguaje de programación.

El objetivo general de este proyecto es aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.

En este sentido, el presente trabajo se centra en la aplicación de los principios de análisis léxico sintáctico y semántico para la construcción de un sistema que cumpla con los requisitos mencionados y adquirir experiencia en el desarrollo de software de calidad.

Lenguaje: TypeScript

Para la implementación de este proyecto, se ha elegido TypeScript como el lenguaje de programación principal. Esta decisión se basa en una serie de consideraciones específicas relacionadas con el desarrollo de un lenguaje de programación y la implementación de sus componentes principales, incluyendo el análisis léxico, sintáctico y semántico.

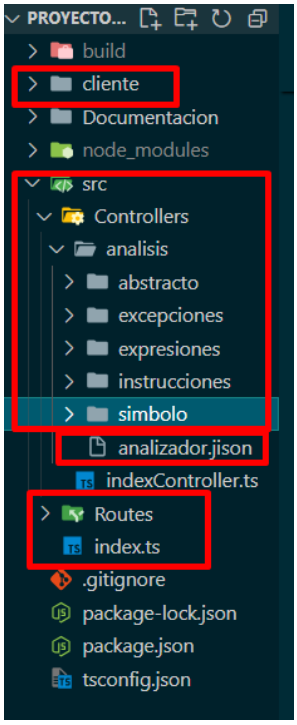
Librerías Utilizadas: Jison y React

En el desarrollo de este proyecto, se han utilizado dos librerías clave: Jison y React. Cada una de estas librerías desempeña un papel fundamental en la implementación y funcionalidad del sistema, y su elección se basa en las necesidades específicas del proyecto.

1. Jison:
    - Análisis Léxico y Sintáctico: Jison es una herramienta que se utiliza para generar analizadores léxicos y sintácticos a partir de especificaciones gramaticales. En el contexto de este proyecto, Jison se emplea para construir el analizador léxico y sintáctico del lenguaje de programación desarrollado. Esto permite definir la gramática del lenguaje y generar automáticamente un analizador que pueda procesar y comprender el código fuente del usuario.
2. React:
    - Interfaz de Usuario Dinámica: React es una biblioteca de JavaScript ampliamente utilizada para construir interfaces de usuario interactivas y dinámicas. En el contexto de este proyecto, React se utiliza para desarrollar la interfaz de usuario del entorno de desarrollo integrado (IDE) para el nuevo lenguaje de programación.

Estructura del proyecto:

El proyecto está estructurado de la siguiente manera:



Primero esta la carpeta Cliente la cual maneja toda la interfaz gráfica con react.

La carpeta src es la que contiene todas las clases de las funcionalidades del programa, dependiendo si es una clase abstracta la cual contiene las instrucciones, después excepciones que contiene errores y de ultimo quedan las expresiones e instrucciones las cuales tiene debidamente clasificadas sus clases.

También dentro de src esta el archivo .jison el cual contiene todo lo relacionado al análisis léxico del proyecto, ahí están definidas las palabras reservadas, expresiones regulares y las producciones para su análisis.

Por último, está la carpeta Routes las cuales van a contener los endpoint para que la aplicación pueda mantener su arquitectura del cliente servidor.

El programa da inicio en la parte de la interfaz grafica al llamar a la función con el endpoint para ejecutar el análisis.

```
function interpretar() {
  var entrada = editorRef.current.getValue();
  fetch('http://localhost:4000/interpretar', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ entrada: entrada }),
  })
    .then(response => response.json())
    .then(data => {
      consolaRef.current.setValue(data.Respuesta);
    })
    .catch((error) => {
      alert("Error en la ejecucion!")
      console.error('Error:', error);
    });
}
```

Luego pasa a darle funcionalidad al interpretar de la clase indexController la cual va a validar que venga una función execute para que inicie el proyecto y después valida que venga un método o puede venir errores o diferentes funcionalidades las cuales va a ejecutar en caso que sean correctas y retornara su resultado.

```
for (let i of listaErrores){
    ast.Print("Error Lexico" + i.getTipoError()+ ":" + i.getDesc() + "Fila: " + i.getFila() + "Columna: " + i.getCol() )
}
for (let i of ast.getInstrucciones()) {
    if(i instanceof Errores){
        listaErrores.push(i)
    }

    if (i instanceof Metodo) {
        i.id = i.id.toLocaleLowerCase()
        ast.addFunciones(i)
        if(i instanceof Errores){
            listaErrores.push(i)
        }
    }
    if(i instanceof Declaracion){
        i.interpretar(ast, tabla)
        if(i instanceof Errores){
            listaErrores.push(i)
        }
    }
    if (i instanceof Execute){
        if(executeEncontrado){
            ast.Print("Error Semantico: Solo se permite una instancia de Execute");
            const error = new Errores("Semántico", "Solo se permite una instancia de Execute", 0, 0);
            listaErrores.push(error);
            break;
        }
        executeEncontrado = true;
        execute = i

    }
```

En el caso que valide que es un error lo va a retornar un objeto de la clase error, el cual mas adelante lo agrego a la lista de errores ya sea léxico sintáctico o semántico.

```
export default class Errores {
    private tipoError: string
    private desc: string
    private fila: number
    private col: number

    constructor(tipo: string, desc: string, fila: number, col: number) {
        this.tipoError = tipo
        this.desc = desc
        this.fila = fila
        this.col = col
    }
```

En caso de que sea una instruccion valida ira al arbol a validar esa instrucción y regresara su funcion.

```
export default class Arbol {
    private instrucciones: Array<Instruccion>
    private consola: string
    private tablaGlobal: tablaSimbolo
    private errores: Array<Errores>
    private funciones: Array<Instruccion>

    constructor(instrucciones: Array<Instruccion>) {
        this.instrucciones = instrucciones
        this.consola = ""
        this.tablaGlobal = new tablaSimbolo()
        this.errores = new Array<Errores>
        this.funciones = new Array<Instruccion>
    }
```

En caso que sea un método ira a la clase método a validar que puede cumplir con lo que se le solicita en la clase del método, caso que no verificara si es una funcion, tomar en cuenta que esta decisión la toma basándose si viene un return o no.

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    //Esto aplica cuando es un metodo porque no trae return, es void
    if (this.tipo.getTipo() == tipoDato.VOID) {

        for (let i of this.instrucciones) {

            let instruccion = i.interpretar(arbol, tabla);

            if (instruccion instanceof Errores) {
                return instruccion;
            }

            if (instruccion instanceof Return) {
                if (instruccion.expresion != undefined) {
                    return instruccion;
                }
                return;
            }
        }
```

```
//Esto es cuando es una funcion porque valida que traiga return
let ReturnEncontrado = false;

for(let i of this.instrucciones){

    if(i instanceof Return){
        ReturnEncontrado = true;

        if(i.expresion != undefined){
            this.returnValue = i.expresion;

            return i.expresion;
        }else{
            arbol.Print("Error Semantico: El return en la funcion es indefinido"+ "Linea: " + this.linea+" columna: " +(this.col+1));
            return new Errores("SEMANTICO", "El return en la funcion es indefinido", this.linea, this.col)
        }
    }
    let resultado = i.interpretar(arbol, tabla);
    if(resultado instanceof Errores){
        return resultado;
    }

    if(resultado instanceof Return){
        if(resultado.expresion != undefined){
            ReturnEncontrado = true;
            this.returnValue = resultado.expresion;
            return resultado.expresion;
```

En el caso que sea una declaración como lo puede ser declarar una variable entonces ira a la funcion de declaración y retornara un valor de la misma clase.

```
if(this.valor.tipoDato.getTipo() == tipoDato.INTEGER && this.tipoDato.getTipo() == tipoDato.DOUBLE){
    //console.log("entro al if")
    this.identificador.forEach(id => {
        //console.log(id)
        valorFinal = parseFloat(valorFinal);
        if (!tabla.setVariable(new Simbolo(this.tipoDato, id, valorFinal))){
            arbol.Print("Error Semantico: No se puede declarar variable que ya existe:"+ this.linea+" columna: " +(this.col+1));

            let error = new Errores("Semantico", "No se puede declarar variable que ya existe", this.linea, this.col)
            listaErrores.push(error)
            return error
        }
    });
}
```

En esta parte contiene diferentes validaciones que la declaración sea el tipo esperado y no se repita.

Tomar en cuenta que se utiliza una clase abstracta la cual estará presente en las demás clases para realizar la función de interpretar que retornara el dato de la acción a realizar ya como el valor esperado que contiene.

```
export abstract class Instruccion {
    public tipoDato: Tipo
    public linea: number
    public col: number

    constructor(tipo: Tipo, linea: number, col: number) {
        this.tipoDato = tipo
        this.linea = linea
        this.col = col
    }

    abstract interpretar(arbol: Arbol, tabla: tablaSimbolos): any
    abstract ArbolGraph(anterior: string): string

}
```

Al validar que venga la función de execute va a procesar la información que tiene, por lo tanto lo va a interpretar y a partir de ahí ya se podrá ejecutar todo el análisis dentro de sus respectivas clases.

```
export default class Execute extends Instruccion {
    interpretar(arbol: Arbol, tabla: tablaSimbolo) {
        if (busqueda == null){
            arbol.Print("Error Semantico: Funcion no existente"+ this.linea+" columna: " +(this.col+1));
            return new Errores("SEMANTICO", "Funcion no existente", this.linea, this.col)
        }

        if (busqueda instanceof Metodo) {
            let newTabla = new tablaSimbolo(arbol.getTablaGlobal())
            newTabla.setNombre("Metodo Execute")


            if (busqueda.parametros.length != this.parametros.length) {
                arbol.Print("Error Semantico: Parametros invalidos"+ this.linea+" columna: " +(this.col+1));
                return new Errores("SEMANTICO", "Parametros invalidos", this.linea, this.col)
            }


            for (let i = 0; i < busqueda.parametros.length; i++) {
                // console.log(busqueda.parametros[i].tipo)
                // console.log(busqueda)
                let declaracionParametro = new Declaracion(
                    busqueda.parametros[i].tipo, this.linea, this.col,
                    busqueda.parametros[i].id, this.parametros[i])

                // declarando parametro de metodo
                let resultado = declaracionParametro.interpretar(arbol, newTabla)
                if (resultado instanceof Errores) return resultado
            }

            let resultadoFuncion: any = busqueda.interpretar(arbol, newTabla)
            if (resultadoFuncion instanceof Errores) return resultadoFuncion
```

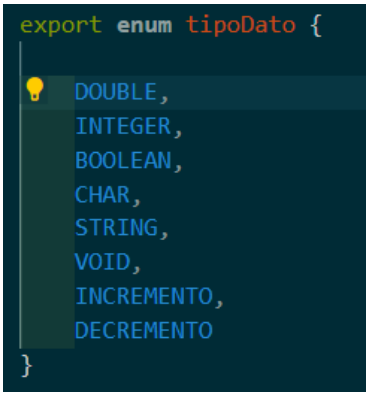
La clase tabla de símbolos es donde se almacenará todas las declaraciones hechas para poder acceder a ellas y obtener atributos como su valor, nombre o asignárselo.

```
export default class tablaSimbolo {
    private tablaAnterior: tablaSimbolo | any
    private tablaActual: Map<string, Simbolo>
    private nombre: string

    constructor(anterior?: tablaSimbolo) {
        this.tablaAnterior = anterior
        this.tablaActual = new Map<string, Simbolo>()
        this.nombre = ""
    }

    public getAnterior(): tablaSimbolo {
        return this.tablaAnterior
    }

    public setAnterior(anterior: tablaSimbolo): void {
        this.tablaAnterior = anterior
    }

    public getTabla(): Map<String, Simbolo> {
        return this.tablaActual;
    }

    public setTabla(tabla: Map<string, Simbolo>) {
        this.tablaActual = tabla
    }

    public getVariable(id: string) {
        for (let i: tablaSimbolo = this; i != null; i = i.getAnterior()) {
            let busqueda: Simbolo = <Simbolo>i.getTabla().get(id.toLocaleLowerCase())
            if (busqueda != null) return busqueda
```

La clase tipo esta para que se puedan definir los tipos de variables que van a existir, luego podemos darle su get y set para utilizarlos respectivamente.

```
export default class Tipo {
    private tipo: tipoDato

    constructor(tipo: tipoDato) {
        this.tipo = tipo
    }

    public setTipo(tipo: tipoDato) {
        this.tipo = tipo
    }

    public getTipo() {
        return this.tipo
    }
```

Estos son los tipos para utilizar.



```
export enum tipoDato {

    DOUBLE,
    INTEGER,
    BOOLEAN,
    CHAR,
    STRING,
    VOID,
    INCREMENTO,
    DECREMENTO
}
```

A continuación, se describirá como fue utilizado las expresiones y sus clases.

AccesoVar:

Va a acceder a una variable solo si cumple con los atributos que se le solicita por lo tanto se hará uso del arbol y la tabla de símbolos para buscar dicha información.

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let valorVariable: Simbolo =<Simbolo> tabla.getVariable(this.id)
    if (valorVariable == null){
        arbol.Print("Error Semantico: "+"Acceso a variable invalido " + "linea: " + this.linea + "columna:" + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "Acceso a variable invalido", this.linea, this.col)
    }
    this.tipoDato = valorVariable.getTipo()
    this.datoEncontrado = valorVariable.getValor()
    return this.datoEncontrado
}
```

Aritmeticas

Esta clase realizara las operaciones aritméticas definidas previamente según su tipo de dato que almacena. Tiene diferentes combinaciones de tipo de dato que puede venir, ya sea un único dato o dos.

```
switch (this.operacion) {
    case Operadores.SUMA:
        return this.suma(opIzq, opDer)
    case Operadores.RESTA:
        return this.resta(opIzq, opDer)
    case Operadores.MULT:
        return this.mult(opIzq, opDer)
    case Operadores.DIVI:
        return this.div(opIzq, opDer)
    case Operadores.NEG:
        return this.negacion(Unico)
    case Operadores.ARI_POTENCIA:
        return this.potencia(opIzq, opDer)
    case Operadores.ARI_MODULO:
        return this.modulo(opIzq, opDer)
    default:
        return new Errores("Semantico", "Operador Aritmetico Invalido", this.linea, this.col)
}
```

Estos son las operaciones aritméticas que es capaz de realizar:

```
export enum Operadores {
    SUMA,
    RESTA,
    MULT,
    DIVI,
    NEG,
    ARI_POTENCIA,
    ARI_MODULO
}
```

Casteos:

Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo. Para hacer esto, se colocará la palabra reservada del tipo de dato destino entre paréntesis seguido de una expresión.

El lenguaje aceptará los siguientes casteos:

- int a double
- double a int
- int a string
- int a char
- double a string
- char a int
- char a double

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let opIzq, opDer, Unico = null
    if (this.operandoUnico != null) {
        Unico = this.operandoUnico.interpretar(arbol, tabla)
        if (Unico instanceof Errores) return Unico
    } else {
        opIzq = this.operando1?.interpretar(arbol, tabla)
        if (opIzq instanceof Errores) return opIzq
        opDer = this.operando2?.interpretar(arbol, tabla)
        if (opDer instanceof Errores) return opDer
    }

    //(int) 12.4
    //(string) 20
    switch (this.operacion.getTipo()) {
        case tipoDato.INTEGER:
            return this.casteo_int(Unico)
        case tipoDato.DOUBLE:
            return this.casteo_double(Unico)
        case tipoDato.STRING:
            return this.casteo_string(Unico)
        case tipoDato.CHAR:
            return this.casteo_char(Unico)
        default:
            arbol.Print("Error Semantico: Casteo Invalido linea:"+ this.linea+" columna: " +(this.col+1));
            return new Errores("Semantico", "Casteo Invalido", this.linea, this.col)
    }
}
```

Funciones:

En este caso las funciones fueron tomadas como una instrucción hacia una variable definida, puede considerarse como la funcion toLower, toUpper o round, cada una de ellas realizara una actividad diferente.

Función tolower

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras minúsculas.

Función toupper

Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras mayúsculas.

Función round

Esta función recibe como parámetro un valor numérico. Permite redondear los números decimales según las siguientes reglas:

- Si el decimal es mayor o igual a 0.5, se aproxima al entero superior.
- Si el decimal es menor que 0.5, se aproxima al número inferior.

```
export default class Funciones extends Instruccion {
    constructor(operador: Operadores, fila: number, col: number, op1: Instruccion, op2?: Instruccion)
        this.operando2 = op2
    }
}

interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let opIzq, opDer, Unico = null
    if (this.operandoUnico != null) {
        Unico = this.operandoUnico.interpretar(arbol, tabla)
        if (Unico instanceof Errores) return Unico
    } else {
        opIzq = this.operando1?.interpretar(arbol, tabla)
        if (opIzq instanceof Errores) return opIzq
        opDer = this.operando2?.interpretar(arbol, tabla)
        if (opDer instanceof Errores) return opDer
    }

    switch (this.operacion) {
        case Operadores.SENT_TOLOWER:
            return this.casteo_lower(Unico,arbol)
        case Operadores.SENT_TOUPPER:
            return this.casteo_upper(Unico,arbol)
        case Operadores.SENT_ROUND:
            return this.casteo_round(Unico,arbol)
        default:
            arbol.Print("Error Semantico: Casteo Invalido" + this.linea+" columna: " +(this.col+1)
            return new Errores("Semantico", "Casteo Invalido", this.linea, this.col)
    }
}
```

Funciones Nativas:

En este caso las funciones fueron tomadas como una instrucción hacia una variable definida, puede considerarse como la funcion length, typeOf o toString, cada una de ellas realizara una actividad diferente.

Length

Esta función recibe como parámetro un vector, una lista o una cadena y devuelve el tamaño de este.

Typeof

Esta función retorna una cadena con el nombre del tipo de dato evaluado.

ToString

Esta función permite convertir un valor de tipo numérico o bool a texto

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let opIzq, opDer, Unico = null
    if (this.operandoUnico != null) {
        Unico = this.operandoUnico.interpretar(arbol, tabla)
        if (Unico instanceof Errores) return Unico
    } else {
        opIzq = this.operando1?.interpretar(arbol, tabla)
        if (opIzq instanceof Errores) return opIzq
        opDer = this.operando2?.interpretar(arbol, tabla)
        if (opDer instanceof Errores) return opDer
    }

    switch (this.operacion) {
        case Operadores.SENT_LENGTH:
            if(Array.isArray(Unico)){
                return this.arreglo(Unico);
            }
            return this.length(Unico,arbol)
        case Operadores.SENT_TYPEOF:
            return this.type(Unico,arbol)
        case Operadores.SENT_TOSTRING:
            return this.setString(Unico,arbol)
        default:
            return new Errores("Semantico", "Casteo Invalido", this.linea, this.col)
    }
}
```

Lógicos:

Manejo de operadores lógicos:

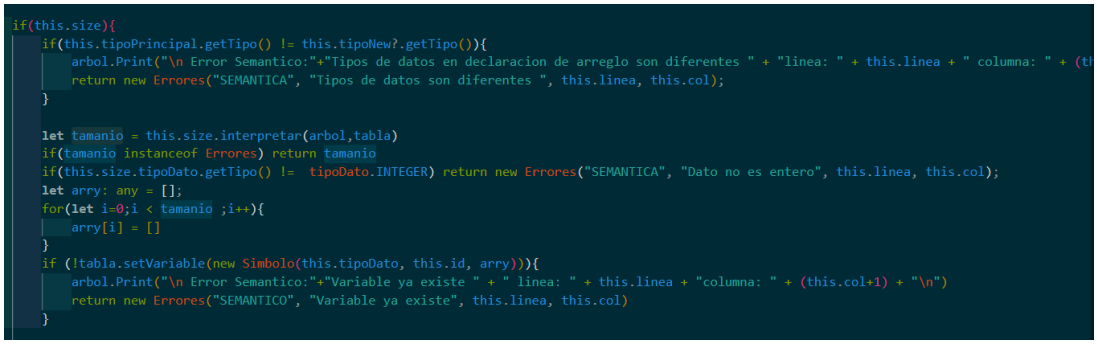
```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let opIzq, opDer, Unico = null
    if (this.operandoUnico != null) {
        Unico = this.operandoUnico.interpretar(arbol, tabla)
        if (Unico instanceof Errores) return Unico
    } else {
        opIzq = this.operando1?.interpretar(arbol, tabla)
        if (opIzq instanceof Errores) return opIzq
        opDer = this.operando2?.interpretar(arbol, tabla)
        if (opDer instanceof Errores) return opDer
    }

    switch (this.operacion) {
        case Operadores.OR:
            return this.comparacion_or(opIzq, opDer,arbol)
        case Operadores.AND:
            return this.comparacion_and(opIzq, opDer,arbol)
        case Operadores.NOT:
            return this.comparacion_not(Unico,arbol)
        default:
            return new Errores("Semantico", "Operador logico Invalido", this.linea, this.col)
    }
}
```

A continuación, se describirá como fue utilizado las instrucciones y sus clases.

Vectores:

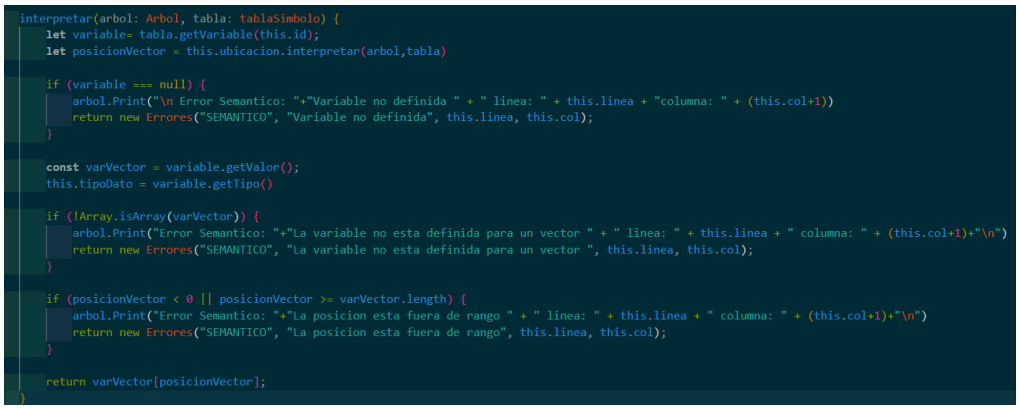
Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, bool, char o string. El lenguaje permitirá únicamente el uso de arreglos de una o dos dimensiones.

Esto es la declaración de un arreglo.



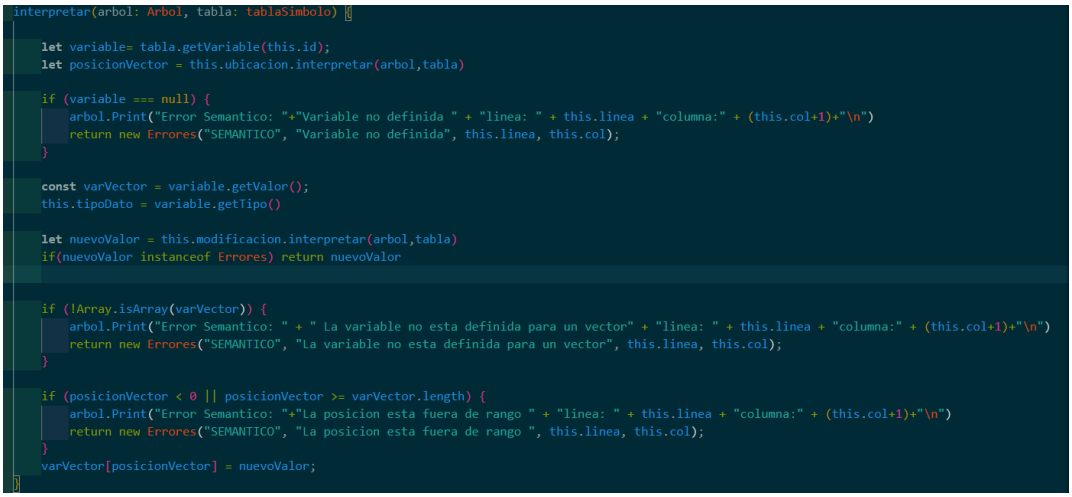
```
if(this.size){
    if(this.tipoPrincipal.getTipo() != this.tipoNew?.getTipo()){
        arbol.Print("\n Error Semantico:"+"Tipos de datos en declaracion de arreglo son diferentes " + "linea: " + this.linea + " columna: " + (th
        return new Errores("SEMANTICA", "Tipos de datos son diferentes ", this.linea, this.col);
    }

    let tamanio = this.size.interpretar(arbol,tabla)
    if(tamanio instanceof Errores) return tamanio
    if(this.size.tipoDato.getTipo() !=  tipoDato.INTEGER) return new Errores("SEMANTICA", "Dato no es entero", this.linea, this.col);
    let arry: any = [];
    for(let i=0;i < tamanio ;i++){
        arry[i] = []
    }
    if (!tabla.setVariable(new Simbolo(this.tipoDato, this.id, arry))){
        arbol.Print("\n Error Semantico:"+"Variable ya existe " + " linea: " + this.linea + "columna: " + (this.col+1) + "\n")
        return new Errores("SEMANTICO", "Variable ya existe", this.linea, this.col)
    }
```

Esto es el acceso a un vector.



```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let variable= tabla.getVariable(this.id);
    let posicionVector = this.ubicacion.interpretar(arbol,tabla)

    if (variable === null) {
        arbol.Print("\n Error Semantico: "+"Variable no definida " + " linea: " + this.linea + "columna: " + (this.col+1))
        return new Errores("SEMANTICO", "Variable no definida", this.linea, this.col);
    }

    const varVector = variable.getValor();
    this.tipoDato = variable.getTipo()

    if (!Array.isArray(varVector)) {
        arbol.Print("Error Semantico: "+"La variable no esta definida para un vector " + " linea: " + this.linea + " columna: " + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "La variable no esta definida para un vector ", this.linea, this.col);
    }

    if (posicionVector < 0 || posicionVector >= varVector.length) {
        arbol.Print("Error Semantico: "+"La posicion esta fuera de rango " + " linea: " + this.linea + " columna: " + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "La posicion esta fuera de rango", this.linea, this.col);
    }

    return varVector[posicionVector];
}
```

Esto es la modificacion de un vector.



```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {

    let variable= tabla.getVariable(this.id);
    let posicionVector = this.ubicacion.interpretar(arbol,tabla)

    if (variable === null) {
        arbol.Print("Error Semantico: "+"Variable no definida " + "linea: " + this.linea + "columna:" + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "Variable no definida", this.linea, this.col);
    }

    const varVector = variable.getValor();
    this.tipoDato = variable.getTipo()

    let nuevoValor = this.modificacion.interpretar(arbol,tabla)
    if(nuevoValor instanceof Errores) return nuevoValor


    if (!Array.isArray(varVector)) {
        arbol.Print("Error Semantico: " + " La variable no esta definida para un vector" + "linea: " + this.linea + "columna:" + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "La variable no esta definida para un vector", this.linea, this.col);
    }

    if (posicionVector < 0 || posicionVector >= varVector.length) {
        arbol.Print("Error Semantico: "+"La posicion esta fuera de rango " + "linea: " + this.linea + "columna:" + (this.col+1)+"\n")
        return new Errores("SEMANTICO", "La posicion esta fuera de rango ", this.linea, this.col);
    }
    varVector[posicionVector] = nuevoValor;
}
```

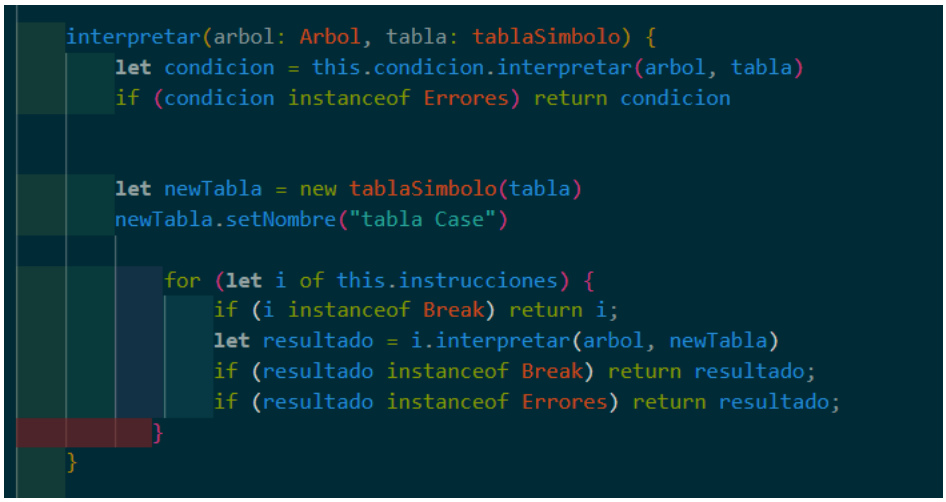
Switch:

Switch case es una estructura utilizada para agilizar la toma de decisiones múltiples, trabaja de la misma manera que lo harían sucesivos if. Estructura principal del switch, donde se indica la expresión a evaluar.

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let condicion = this.condicion.interpretar(arbol, tabla)
    if (condicion instanceof Errores) return condicion


    for (let caso of this.listaCase) {
        let casoTemp = caso.condicion.interpretar(arbol,tabla)
        if (casoTemp instanceof Errores) return casoTemp
        if (casoTemp == condicion) {
            let resultado = caso.interpretar(arbol, tabla)

            if (resultado instanceof Return) return resultado;

            if (resultado instanceof Errores) return resultado
            if (resultado instanceof Break) return;

        }
    }

    if (this.casoDefault) {
        let resultado = this.casoDefault.interpretar(arbol, tabla);
        if (resultado instanceof Errores) return resultado

        if (resultado instanceof Return) return resultado;
        if (resultado instanceof Break) return;

    }
```

Case

Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el switch



```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let condicion = this.condicion.interpretar(arbol, tabla)
    if (condicion instanceof Errores) return condicion


    let newTabla = new tablaSimbolo(tabla)
    newTabla.setNombre("tabla Case")

        for (let i of this.instrucciones) {
            if (i instanceof Break) return i;
            let resultado = i.interpretar(arbol, newTabla)
            if (resultado instanceof Break) return resultado;
            if (resultado instanceof Errores) return resultado;
        }
}
```

Default

Estructura que contiene las sentencias si en dado caso no haya salido del switch por medio de una sentencia break

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {

    let newTabla = new tablaSimbolo(tabla)
    newTabla.setNombre("tabla Default")


    for (let i of this.instrucciones) {
        if (i instanceof Break) return i;
        let resultado = i.interpretar(arbol, newTabla)
        if (resultado instanceof Break) return resultado;
    }
}
```

DeclaracionInit:

Inicializa las variables sin necesidad de asignarle un valor.

Do While:

El ciclo o bucle Do-While, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let condicion = this.condicion.interpretar(arbol, tabla)
    if (condicion instanceof Errores) return condicion

    if (this.condicion.tipoDato.getTipo() != tipoDato.BOOLEAN) {
        arbol.Print("Error Semantico: La condicion debe ser bool. linea:"+ this.linea+" columna: " +(this.col+1));
        return new Errores("SEMANTICO","La condicion debe ser bool",this.linea,this.col)
    }

    do{
        let newTabla = new tablaSimbolo(tabla)
        newTabla.setNombre("Funcion do while")
        for (let i of this.instruccion) {
            if (i instanceof Break) return;
            if (i instanceof Continue) break;
            if (i instanceof Return) return i;
            let resultado = i.interpretar(arbol, newTabla)
            if (resultado instanceof Errores) return resultado;
            if (resultado instanceof Return) return resultado;
            if (resultado instanceof Break) return;
            if (resultado instanceof Continue) break;


        }

    }while(this.condicion.interpretar(arbol, tabla))
```

For:

El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let tablaTemp  = new tablaSimbolo(tabla);

    this.variable.interpretar(arbol, tablaTemp);

    let cond = this.condicion.interpretar(arbol, tablaTemp);
    if(cond instanceof Errores) return cond;

    if(this.condicion.tipoDato.getTipo()!= tipoDato.BOOLEAN){
        arbol.Print("Error Semantico: La condicion debe ser bool. linea:"+ this.linea+" columna: " +(this.col+1));
        return new Errores("Semantico", "La condicion debe ser bool", this.linea, this.col);
    }

    while(this.condicion.interpretar(arbol,tablaTemp)){
        let nuevaTabla = new tablaSimbolo(tablaTemp);
        nuevaTabla.setNombre("tabla For");
        for(let i of this.instrucciones){
            if(i instanceof Break) return;
            if (i instanceof Continue) break;
            let resultado = i.interpretar(arbol, nuevaTabla);
            if(resultado instanceof Errores) return resultado;
            if (resultado instanceof Return) return resultado;
            if(resultado instanceof Break) return;
            if (i instanceof Continue) break;
        }
        this.inc_dec.interpretar(arbol, nuevaTabla);
    }
```

If:

La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro de la sentencia.

```
if (condicion) {
    for (let i of this.instruccionesif) {
        // if (i instanceof Break) return i;
        // if (i instanceof Return) return i;
        let resultado = i.interpretar(arbol, newTabla)
        if (resultado instanceof Break) return;
        if (resultado instanceof Return) return resultado;
        if (resultado instanceof Errores) return resultado;
    }
}else{
    if(this.instruccioneselse){
        for (let i of this.instruccioneselse) {
            if (i instanceof Break) return i;
            if (i instanceof Continue) return i;
            if (i instanceof Return) return i;
            let resultado = i.interpretar(arbol, newTabla)
            if (resultado instanceof Break) return;
            if (resultado instanceof Return) return resultado;
            if (resultado instanceof Errores) return resultado;

        }
    }
}
```

While:

El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

```
interpretar(arbol: Arbol, tabla: tablaSimbolo) {
    let cond = this.condicion.interpretar(arbol, tabla)
    if (cond instanceof Errores) return cond

    // validaciones
    if (this.condicion.tipoDato.getTipo() != tipoDato.BOOLEAN) {
        arbol.Print("Error Semantico: La condicion debe ser bool. linea:"+ this.linea+" columna: " +(this.col+1));
        return new Errores("SEMANTICO", "La condicion debe ser bool", this.linea, this.col)
    }

    while (this.condicion.interpretar(arbol, tabla)) {
        let newTabla = new tablaSimbolo(tabla)
        newTabla.setNombre("Sentencia While")
        for (let i of this.instrucciones) {
            if (i instanceof Break) return;
            if (i instanceof Continue) break;
            if (i instanceof Return) return i;
            let resultado = i.interpretar(arbol, newTabla)
            if (resultado instanceof Break) return;
            if (resultado instanceof Continue) break;
            if (resultado instanceof Errores) return resultado;

        }
    }
```

Analizador.jison:

En este archivo es donde se maneja y declara toda la gramática que se utilizara. Primero se deben importar todas las clases.

```
const Tipo = require('./simbolo/Tipo')
const Nativo = require('./expresiones/Nativo')
const Aritmeticas = require('./expresiones/Aritmeticas')
const Relacionales = require('./expresiones/Relacionales')
const Logicos = require('./expresiones/Logicos')
const Casteo = require('./expresiones/Casteos')
const Funciones = require('./expresiones/Funciones')
const FuncionesNativas = require('./expresiones/FuncionesNativas')
const IncDec = require('./expresiones/IncDec')
const Ternaria = require('./expresiones/ternario')
const AccesoVar = require('./expresiones/AccesoVar')


const Llamada = require('./instrucciones/Llamada')
const ModArrayU = require('./instrucciones/ModArrayU')
const AccesoArrayU = require('./instrucciones/AccesoArrayU')
const Execute = require('./instrucciones/Execute')
const Metodo = require('./instrucciones/Metodo')
const If = require('./instrucciones/If')
const While = require('./instrucciones/While')
const DoWhile = require('./instrucciones/doWhile')
const For = require('./instrucciones/For')
const Switch = require('./instrucciones/Switch')
const Case = require('./instrucciones/Case')
const Default = require('./instrucciones/Default')
const Break = require('./instrucciones/Break')
const Continue = require('./instrucciones/Continue')
const Print = require('./instrucciones/Print')
const Declaracion = require('./instrucciones/Declaracion')
const DeclaracionInit = require('./instrucciones/DeclaracionInit')
```

Esta es la parte de declaración de palabras reservadas:

```
//----------Tipo de dato----------
"double"                return 'DOUBLE'
"int"                   return 'INTEGER'
"bool"                  return 'BOOLEAN'
"char"                  return 'CHAR'
"std::String"           return 'STRING'
"std"                   return 'STD'
//----------Incremento y Decremento----------
"++"                    return 'INCREMENTO'
"--"                    return 'DECREMENTO'
//----------Operadores Aritmeticos----------
"+"                     return 'ARI_SUMA'
"-"                     return 'ARI_MENOS'
"*"                     return 'ARI_MULTIPLICACION'
"/"                     return 'ARI_DIVISION'
"POW"                   return 'ARI_POTENCIA'
"%"                     return 'ARI_MODULO'
//----------Operadores Relacionales----------
"!="                    return 'DIFERENCIACION'
"=="                    return 'IGUALACIONDOBLE'
"="                     return 'IGUALACION'
"<="                    return 'MENORIGUAL'
">="                    return 'MAYORIGUAL'
">"                     return 'MAYOR'
"<<"                    return 'PRINTMENOR'
"<"                     return 'MENOR'
//----------Operadores Logicos----------
"||"                    return 'OR'
"&&"                    return 'AND'
"!"                     return 'NOT'
```

Y la parte de orden de la gramática con sus producciones donde se manejan las terminales y no terminales.

```
INICIO : INSTRUCCIONES EOF                  {return $1;}
;

INSTRUCCIONES : INSTRUCCIONES INSTRUCCION   {$1.push($2); $$=$1;}
              | INSTRUCCION                 {$$=[$1];}
;

INSTRUCCION : IMPRESION             {$$=$1;}
            | DECLARACION PUNTO_COMA          {$$=$1;}
            | ASIGNACION PUNTO_COMA           {$$=$1;}
            | OPC_IF                              {$$=$1;}
            | INS_WHILE                       {$$=$1;}
            | INS_BREAK                       {$$=$1;}
            |INS_CONTINUE                   {$$=$1;}
            |INS_FOR                          {$$=$1;}
            |INS_DOWHILE                      {$$=$1;}
            | INS_SWITCH                    {$$=$1;}
            | DECLARACION_ARREGLO     {$$=$1;}
            | FUN_METODO                            {$$=$1;}
            | FUN_EXE               {$$=$1;}
            | FUN_LLAMADA PUNTO_COMA                {$$=$1;}
            | MOD_VECTOR                          {$$=$1;}
            |MOD_VECTOR_D                   {$$=$1;}
            | INS_RETURN                {$$=$1;}
            | DECLARACION_ARREGLO_D   {$$=$1;}
;
```

Para que no se encuentre ambigüedad se utiliza precedencia.

```
// Precedencias
%left 'OP_TERNARIO'
%left 'OR'
%left 'AND'
%right 'NOT'
%left 'IGUALACIONDOBLE' 'DIFERENCIACION' 'MENOR' 'MENORIGUAL' 'MAYOR' 'MAYORIGUAL'
%left 'ARI_SUMA' 'ARI_MENOS'
%left 'ARI_MULTIPLICACION' 'ARI_DIVISION' 'ARI_MODULO'
%left 'INCREMENTO','DECREMENTO'
%left signoMenos
%left 'PARENTESIS_IZQ'
%left 'SENT_LENGTH'
```