UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

Escuela de Ciencia y Sistemas

Organización de Lenguajes y Compiladores 1

PROYECTO 2

Sergio Giovanni Castro Funes - 201800723

Guatemala, 29 de abril de 2022

OBJETIVOS

Objetivo General:

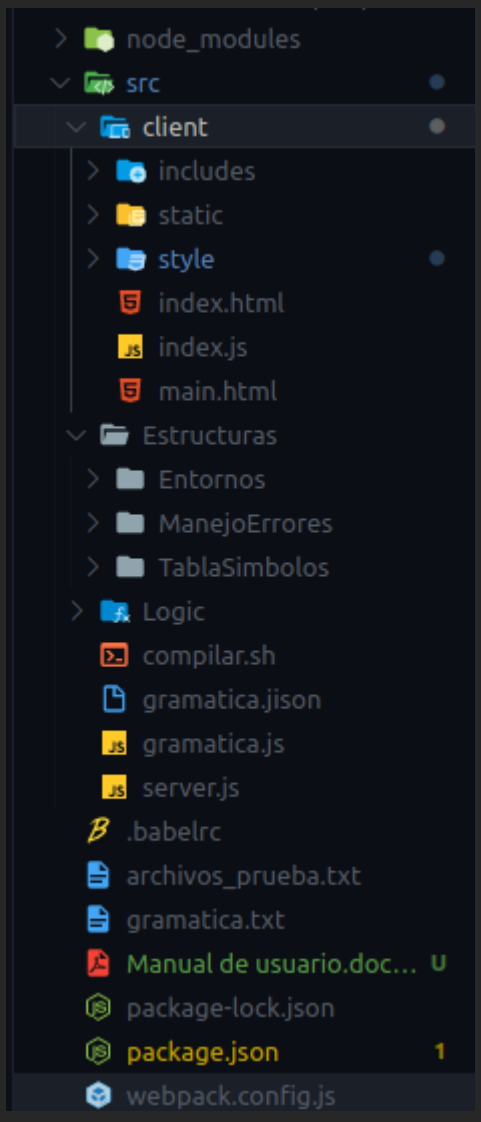
- **Aplicar los conocimientos sobre la fase de análisis léxico y sintáctico de un compilador para la realización de un intérprete sencillo, con las funcionalidades principales para que sea funcional.**

Objetivos Específicos:

- **Reforzar los conocimientos de análisis léxico y sintáctico para la creación de un lenguaje de programación.**
- **Aplicar los conceptos de compiladores para implementar el proceso de interpretación de código de alto nivel.**
- **Aplicar los conceptos de compiladores para analizar un lenguaje de programación y producir las salidas esperadas.**
- **Aplicar la teoría de compiladores para la creación de soluciones de software.**
- **Aplicar conceptos de contenedores para generar aplicaciones livianas.**
- **Generar aplicaciones utilizando arquitecturas Cliente-Servidor.**

SECCIONES

Estructura del proyecto:



-Client: se encuentra toda la parte del frontend

-Estructuras: se encuentran todas las estructuras de datos utilizadas en el proyecto

entornos: define el entorno actual de una funcion y variable

Manejo Errores: nos sirve para guardar todos los errores producidos en el analisis

Tabla simbolos: nos sirve para guardar todos los simbolos definidos en el codigo

-logic: aca estan todas las clases que manega la logica semantica del lenguaje

Archivos importantes:

-server.js : aca se declararon los enpoints de la aplicacion

-compilar.sh : nos sirve para compilar el archivo jison

-gramatica.jison : aca se declararon todas las reglas sintacticas y lexicas del proyecto

-gramatica.js: es un archivo creado al ejecutar compilar.sh

-package.json: este archivo nos sirve para instalar todas las dependencias necesarias para correr el proyecto

Frontend:

El frontend es manejado por un servidor web y las vistas se completan utilizando la libreria Nunjucks la cual se completa como una pagina padre - hijo

ejemplo :

```
{% include 'includes/sidebar.html' %}
<main class='content' id='main'>
    {% block content %}

    {% endblock content %}
</main>
<script type="text/javascript">
```

aca incluye el side bar que esta definido en otro html independiente

en block content se puede adjuntar otro html diferente heredando desde el padre

lógica frontend:

toda la logica esta manejada en un archivo llamado index.js que se encuentra en la carpeta client

run-btn: tiene una funcion jquery la cual manda los datos a un endpoint de nuestra API los datos que manda es el texto del editor lo manda al endpoint /prueba mediante un fetch espera la respuesta de nuestra API y guarda los resultados del json en variables globales para poder manejarlas en el frontend

```
code_ouput.setValue(datos.code);
tblErrores = datos.htmlErrores;
tblSimbolos = datos.simbolos;
dot = datos.dot;
```

code_ouput: es la consola de salida aca se le setea el resultado del analisis

tblErrores: guarda el html generado de los errores

tblSimbolos: guarda el html generado de los simbolos que se declararon en nuestro codigo

dot: retorna un texto plano en formato dot para luego ser analizado con otra funcion

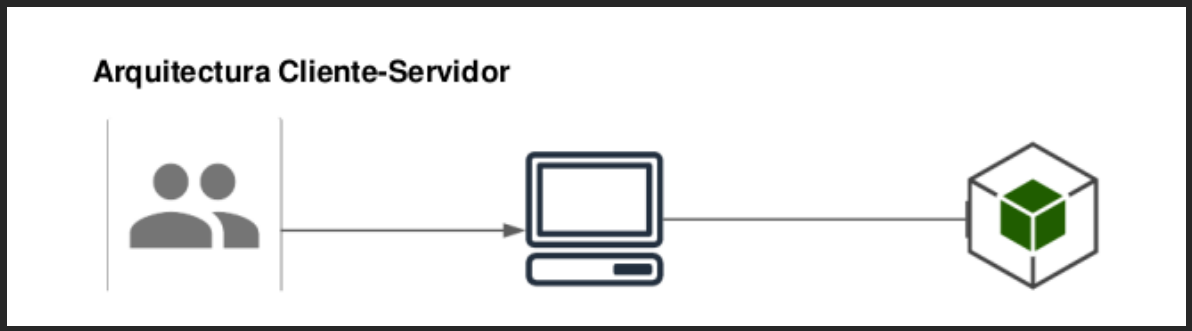
Rep_ast : genera la imagen del reporte del árbol ast para esto se utiliza la

```
$("#rep_ast").on("click", function () {
  if(dot !== ""){
    document.getElementById("contenido").className ="card animate__animated animate__jackInTheBox";
    var svg = Viz(dot, "svg");
    var data = `
    <div class="card-header" style = "border-bottom: 0px ">
        Arbol AST
    </div>
    <div class"card-body" > `
    data += svg ;
    data += `</div>`;
    document.getElementById("contenido").innerHTML = data;
  }

});
```

herramienta de viz para que nos genere la imagen y luego la adjuntamos a un div

Arquitectura aplicada: cliente-servidor



Backend:

explicacion de la generacion del árbol AST:

```
function AST_Node(name, value, tipo,entorno, fila, columna ){
    this.name = name;
    this.value = value;
    this.tipo = tipo;
    this.entorno = entorno;
    this.fila = fila;
    this.columna = columna;

    this.hijos =[];
    this.addHijos = addHijos;
    this.getHermano = getHermano;

    function addHijos(){
        for(var i=0; i<arguments.length; i++){
            this.hijos.push(arguments[i]);
            if(arguments[i == null]){
                arguments[i].padre.this;
            }      You, hace 2 semanas • backend se agrega la func
        }
    }
    function getHermano(pos){
        if(pos>this._hijos.length-1) return null;
        return this._hijos[pos];
    }
}
```

se maneja una estructura de tipo árbol N-ario en la cual un nodo puede tener tantos hijos como sean necesarios los datos que reciben son nombre del nodo, valor del nodo, tipo , entorno , linea donde se crea el nodo, columna donde se crea el nodo

los métodos que tiene esta estructura es para añadir hijos y obtener los hermanos o nodos que se encuentren en el mismo nivel

Declaracion de un nodo:

```
INIT
    : LINS EOF  {$$=new AST_Node("RAIZ","RAIZ","palabra reservada","global",this.$first_line,this._$.last_column);$$.addHijos($1);return $$};

//Valores AST name, value, tipo,entorno, fila, columna
LINS
    : LINS INSTRUCC {$1.addHijos($2);$$=$1;}
    | INSTRUCC  {$$= new AST_Node("SENTENCIAS","SENTENCIAS","SENTENCIAS","global",this._$.first_line,this._$.last_column);
                  $$.addHijos($1);}
    ;
```

El nodo Raiz es el nodo padre de todos los nodos seguido de sentencias en sentencias puede venir cualquier nodo declarado luego de sentencias $$.adhijos($1) añade a sentencias y en sentencias este mismo codigo añade los hijos que retorna INSTRUCC

manejo de errores lexico y sintacticos:

```
{ console.error('Este es un error léxico: ' + yytext + ', en la linea: ' + yylloc.first_line + ', en la columna: ' + yylloc.first_column);
TablaErrores.getInstance().insertarError(new _Error("Lexico","Caracter: \" "+yytext+"\" no es valido" ,yylloc.first_line,yylloc.first_column))
return null; }
```

TablaErrores.getInstance() llama a una instancia declarada con anterioridad de esta estructura por lo cual hace que no se pierdan los errores en cada lectura de las lineas de entrada, la misma logica aplica para los errores sintacticos

server.js:

```
You, hace 2 horas | 2 authors (You and others)
import express from 'express';
const {Interprete} =  require('./Logic/Interprete')
const nunjucks = require('nunjucks');

// initializing packages
const app = express();
const path = require('path');
const {TablaErrores} = require('./Estructuras/ManejoErrores/TablaErrores.js') ;
const {TablaSimbolos} = require('./Estructuras/TablaSimbolos/TablaSimbolos.js');
import {Entorno} from "./Estructuras/Entornos/Entorno";
import {Clasificacion} from "./Estructuras/Entornos/Entorno";
var bodyParser = require('body-parser')
const parser = require('./gramatica');
nunjucks.configure(__dirname+'/client', {
  autoescape: true,
  express: app
});


// create application/json parser
var jsonParser = bodyParser.json()
// settings
var urlencodedParser = bodyParser.urlencoded({ extended: false })
app.set('port', process.env.PORT || 9000);

app.use(bodyParser.json({ limit: '10mb', extended: true }));
app.use(bodyParser.urlencoded({ limit: '10mb', extended: true }))
app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Headers', 'Authorization, X-API-KEY, Origin, X-Requested-With, Content-Type, Accept, Access-Control-Allow-Request-
    res.header('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, DELETE');
    res.header('Allow', 'GET, POST, OPTIONS');
    next();
  });
```

configuracion inicial del servidor web:

la aplicacion tiene un limite de 10mb por json desde el bodyparse

los header definidos fueron GET, POST , OPTIONS

permitiendo el control alos header mediante la consultas via POST

la aplicacion funciona bajo el puerto 9000

se requiere express, nunjucks , bodyparser para que el servidor funcione correctamente

ENDPOINTS:

/: renderiza el html inicial via con la ayuda de nunjucks

/prueba: analiza la entrada del codigo declarado en el frontend

aca se reinician todas las estructuras para que no muestre datos de entradas anteriores se reinician las tablas de errores y simbolos

Parser.parse(): recibe la entrada para se analizada mediante jison el return de esta funcion es el arbol AST

Analizar(): recibe el resultado de parser.parse() para analizar el arbol y generar la logica del codigo

imprimir(): recibe el arbol ast para pasarlo a un archivo .dot y asi poder generar la imagen del arbol

INTERPRETE:

```
export class Interprete{
    constructor(){

    }
    analizar(root){      You, hace 2 semanas • Logic se lleva operaciones asignacion
        return this.interpretar(root);
    }

    interpretar(root){
        let op;
        let result;
        let code = "";
        let simbolo;

        if(root === undefined || root === null) return;

        switch(root.name){
            case "RAIZ":
                root.hijos.forEach(hijo => code+= this.interpretar(hijo));
                return code;
            case "SENTENCIAS":
                let resultado_sentencia;
                for(var i = 0; i <root.hijos.length; i++){
                    resultado_sentencia = this.interpretar(root.hijos[i]);
                    if(resultado_sentencia != "break"){
                        code+=resultado_sentencia;
                    }else{

                        break;
                    }
                }
                return code;
```

La clase interprete recibe un nodo puede ser cualquier nodo del arbol AST para cada nombre del nodo tiene definido ciertas operaciones por ejemplo:

Nodo Raiz : no tiene funciones especificas solo define el inicio del arbol por lo cual itera a la misma funcion interpretar con su hijo o hijos

SENTENCIAS: sentencias solo define el inicio del entorno global por lo cual no define ni crea simbolos u operaciones por lo cual itera a la misma funcion la cantidad de veces necesarias

todas estas iteraciones retornan resultados por lo cual el return code

Declaracion de simbolos y analisis semantico:

```
case "int":
    simbolo = new Simbolo(root.hijos[0].value, //nombre
    root.hijos[0].tipo, // tipo
    "variable", // clasificacion
    Entorno.getInstance().entorno(), // entorno
    0, // valor
    root.hijos[0].fila,
    root.hijos[0].columna);
    TablaSimbolos.getInstance().insertarSimbolo(simbolo);
    break;
```

aca se muestra el caso en el que una variable sea de tipo entero crea un nuevo simbolo y lo inserta a la tabla esto en dado caso no se le asigne ni un valor a la variable

ejemplo : int prueba ;

Asignación de valores a las variables:

```
op = new Operador();
result = op.ejecutar(root.hijos[1])
if(result.tipo == root.hijos[0].tipo.toLowerCase()){
    simbolo = new Simbolo(root.hijos[0].value, //nombre
    root.hijos[0].tipo, // tipo
    "variable", // clasificacion
    Entorno.getInstance().entorno(), // entorno
    result.valor, // valor
    root.hijos[0].fila,
    root.hijos[0].columna);
    TablaSimbolos.getInstance().insertarSimbolo(simbolo);
    console.log("simbolo insertado");
    break;
```

en este caso se llama a la clase Operador la cual tiene un metodo ejecutar esta se encargara de retornar el valor de la variable en dado cas esta tenga una operacion o se le quiera cambiar el valor

Clase Resultado:

```
You, hace 2 semanas | 1 author (You)
export class ResOperacion {
    constructor(tipo, valor){
        this.tipo = tipo;
        this.valor = valor;
    }       You, hace 2 semanas • Lo
}
```

esta clase retorna un objeto con dos datos el tipo y valor

esto nos sirve para guardar el resultado de una mejor manera y retornarlo desde operador a interprete

Clase Operador:

```
export class Operador{

    constructor(){

    }

    ejecutar(root){
        var Resultado1 = null;
        var Resultado2 = null;
        var Resultado = null;

        switch(root.name){
            case "EXP":
                if(root.hijos.length == 3){
                    Resultado1=this.ejecutar(root.hijos[0]);
                    Resultado2=this.ejecutar(root.hijos[2]);
                    var operacion = root.hijos[1].value;
                    console.log(operacion);
                    switch (operacion) {
                        case "+":
                        case "-":
                        case "*":
                        case "/":
                        case "^":
                        case "%":
                            return this.aritmetico(Resultado1,Resultado2,operacion,root.hijos[1].fila,root.hijos[1].columna);
                        case "==":
                        case "!=":
                            return this.igualdad(Resultado1,Resultado2,operacion,root.hijos[1].fila,root.hijos[1].columna);
                        case ">":
                        case ">=":
                        case "<":
                        case "<=":
                            return this.relacional(Resultado1,Resultado2,operacion,root.hijos[1].fila,root.hijos[1].columna);
                        case "&&":
```

la clase operador recibe cualquier tipo de nodo, tiene dos resultados para operaciones aritméticas y para operaciones simples como reasignacion de valores solo se usa el resultado1 en esta funcion estan definidos los tipos de nodos que se utilizan para manejar la logica del lenguaje

EXP puede ser cualquier expresion ya sea una variable, una operacion aritmética , un casteo o una funcion propia del lenguaje

cuando tiene 3 hijos es una operacion aritmética , logica o relacional

cuando tiene 2 hijos son operaciones de casteo

cuando tiene 1 hijo puede ser funciones manejadas por el lenjuare, retorno de variables , asignaciones de variables declaraciones de vectores reasignacion de vectores

Operaciones Logicas:

```
logicos(R1,R2,op,fila,columna){
    let tipo1 = R1.tipo;
    let tipo2 = R2.tipo;
    var res = new ResOperacion();
    if(tipo1 == "error" || tipo2 == "error"){
        res.tipo = "error";
        return res;
    }

    if(tipo1 == "boolean" && tipo2 == "boolean"){
        res.tipo = "boolean";
        switch(op){
            case "&&":
                res.valor =R1.valor&&R2.valor;
                return res;
            case "||":
                res.valor =R1.valor||R2.valor;
                return res;
        }
    }else{
        TablaErrores.getInstance().insertarError(new _Er
        res.tipo = "error";
        res.valor = "error";
        return res;
    }
}
```

aca se definen las operaciones logicas ya sea and o or

se utiliza una variable boleana a la cual su resultado sera dado por los operadores propiamente implementados por javascript

si los tipos que se reciben no son de tipo bool retorna un error y lo guarda en nuestra estructura de errores

Aritmetico():

```
aritmetico(R1,R2,operacion, fila, columna){
    let tipo1 = R1.tipo;
    let tipo2 = R2.tipo;
    var res = new ResOperacion();
    if(tipo1=="error"||tipo2=="error" ){
        res.tipo="error";
        return res;
    }else if(typeof R1.valor ==='object'|| typeof R2.valor ==='object'){
        res.tipo ="objeto";
        res.valor = "objeto";
        return res;
    }
    switch (operacion) {
        case "+":
            switch (tipo1.toLowerCase()) {
                case "int":
                    switch (tipo2.toLowerCase()) {
                        case "int":
                            res.tipo = "int";
                            res.valor = R1.valor + R2.valor;
                            return res;
                        case "double":
                            res.tipo = "double"
                            res.valor = parseFloat(R1.valor) + parseFloat(R2.valor);
                            return res;
                        case "boolean":
                            res.tipo = "int"
                            res.valor = R2.valor ? R1.valor + 1 : R1.valor ;
                            return res;
                        case "char":
                            res.tipo = "int"
                            res.valor = R1.valor + R2.valor.charCodeAt(0);
                            return res;
                        case "string":
                            res.tipo = "string"
                            res.valor = R1.valor.toString() +  R2.valor;
                            return res;
```

la funcion aritmetica recibe 2 parametros y una operacion validamos que tipo de operacion se requiere y validamos si la operacion entre tipos esta permitida si las entradas cumplen con lo solicitado se genera la operacion y se retorna el tipo y el valor especificado para el tipo de operacion entre tipos

Relacional():

```
relacional(R1,R2,op,fila,columna){
    let tipo1 = R1.tipo;
    let tipo2 = R2.tipo;
    var res = new ResOperacion();
    if(tipo1=="error"||tipo2=="error"){
        res.tipo="error";
        return res;
    }
    if(this.verificarrelacional(tipo1,tipo2)){
        if(tipo1==="char" &&  tipo2==="int" && R1.valor.length >1  ){
            R1.valor =  typeof R1.valor ==="string" ? R1.valor.charCodeAt(0) : R1.valor;
            R2.valor =  typeof R2.valor ==="string" ? R2.valor.charCodeAt(0) : R2.valor;
            console.log(R1,R2);
        }else if (tipo2==="char" &&  tipo1==="int" && R2.valor.length >1  ){
            R1.valor =  typeof R1.valor ==="string" ? R1.valor.charCodeAt(0) : R1.valor;
            R2.valor =  typeof R2.valor ==="string" ? R2.valor.charCodeAt(0) : R2.valor;
            console.log(R1,R2);
        }
        switch(op){
            case ">":
                res.tipo="boolean";
                res.valor=R1.valor>R2.valor;
                return res;
            case "<":
                res.tipo="boolean";
                res.valor=R1.valor<R2.valor;
                return res;
            case ">=":
                res.tipo="boolean";
                res.valor=R1.valor>=R2.valor;
                return res;
            case "<=":
```

esta funcion aplica el mismo concepto de la aritmetica recibe 2 parametros y un tipo de operacion si las dos entradas cumplen con lo solicitado se realiza la operacion sino da un error y lo guarda en nuestra estructura.

verifica casteo:

```
verificarCasteo(tipo1,tipo2){
    switch(tipo1){
        case "int":
            switch(tipo2){
                case "double":
                case "string":
                case "char":
                    return true;
                default:
                    return false;
            }
        case "double":
            switch(tipo2){
                case "int":
                case "string":
                    return true;
                default: return false;
            }
        case "char":
            switch(tipo2){
                case "int":
                case "double":
                    return true;
                default: return false;
            }
    }
}
```

valida el casteo antes de ejecutar algún cambio

básicamente recibe el tipo de las dos variables involucradas en el casteo

y determina mediante las reglas declaradas si el casteo es valido

Obtener valores de la tabla :

```
case "ID":
    Resultado= new ResOperacion();
    let simbolo = TablaSimbolos.getInstance().getSimbolo(root.value);
    if(simbolo==null || simbolo== undefined){
        Resultado.tipo = "error";
        Resultado.valor="Error semantico, no se le pudo asignar el valor deseado a la variable"
        TablaErrores.getInstance().insertarError(new _Error("Semantico","Error en: \" "+root.value+"
        return Resultado;
    }
    Resultado.tipo = simbolo.tipo;
    Resultado.valor=simbolo.valor;
    return Resultado;
```

se hizo un caso especial para id y cuando viene ya sea una asignacion o una operacion se va a buscar el id en la instancia de la tabla de simbolos y se obtiene el valor y el tipo de la variable