UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERIA

ESCUELA DE CIENCIAS Y SISTEMAS

LABORATORIO ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1

MANUAL TECNICO

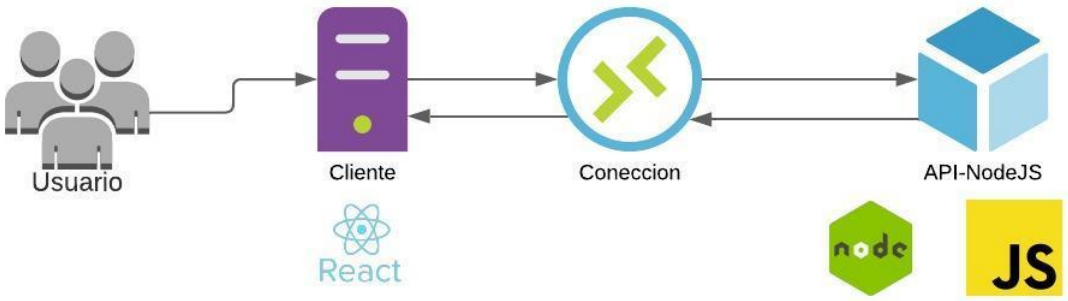
DAVID ROBERTO DIAZ PRADO

201807420

SECCION B

TYPESTY es un interprete sencillo con operaciones básicas pero puede se un buen lenguaje para aprender cosas sencillas en la programación, a continuación se explica como utilizar el lenguaje

Para este proyecto se utilizo como lenguaje principal JavaScript, se tuvo que levantar un servicio que recibiera las peticiones y mandara respuestas a nuestro frontend como se puede ver en la siguiente imagen



Ahora pasamos a la estructura que se utilizo en node js

Se utilizaron varios archivos tipo javascript para la realización de nuestro proyecto

```
router.post('/',function(req,res,next) {
  let salidaCon='';
    try {
      var nueva = req.body.informacion;
      let arbol = parser.parse(nueva);
      var salida =interprete(arbol);
      var grafico = analisis(arbol);
      salidaCon=salida.salida;
      errorT=salida.errores;
      simboT=salida.simbolos;
      res.statusCode =200;
      let absu=JSON.stringify(arbol);
      fs.writeFileSync('ast.json',absu);
      res.json(
        {
          respuesta:salidaCon,
          error:errorT,
          simbolos:simboT,
          grafico:grafico
        });
    } catch (e) {
      console.log(e);
      res.statusCode =200;
      res.json({
        respuesta:salidaCon,
        error:errorT,
        simbolos:simboT
      });
    }
```

Función donde se levanta nuestro servicio y este por medio de una petición tipo Post donde recibimos de parte de nuestro cliente la petición como se puede ver el *req.body.informacion* este parametro recibe lo que estaa dentro de nuestro text área en nuestro frontend la respuesta que se envía a nuestro cliente la podemos ver con

```
res.json(
  {
    respuesta:salidaCon,
    error:errorT,
    simbolos:simboT,
    grafico:grafico
  });
```

envía una respuesta en tipo json con la respuesta que se manda a la consola de nuestro cliente y sus respectivos reportes.

Para hacer la gramatica para nuestro lenguaje utilizamos la herramienta Jison que esta basada en Bison y este se orienta mas a javascript

La definición de nuestra gramatica la puede ver en el manual de usuario

A grandes rasgos se puede ver que aquí definimos los patrones para el análisis lexico

```
"tochararray"           return 'toCaracter';
"int"                   return 'tipoInt';
"double"                return 'tipoDouble';
"boolean"               return 'tipoBooleano';
"char"                  return 'tipoChar';
"string"                return 'tipoString';

//pueden faltar los -- o ++

([\"]("\\\""|[^"])*[^\\][\"])|[\"][\"]              {yytext=yytext.substr(1,yyleng-2); return 'cadenaaa';}
\'([^\']|"\\n"|"\\r"|"\\t")\'               {yytext=yytext.substr(1,yyleng-2); return 'caracter'};
[0-9]+"."[0-9]+\b   return 'decimall';
[0-9]+\b                        return 'entero';
([a-zA-Z])[a-zA-Z0-9_]* return 'identificador';


<<EOF>>                 return 'EOF';
```

Apartado para meter código JS

```
/lex
%{
  var lista_Errores = []
  const TIPO_ERROR        = require('./controller/Ambito/TipoError');
  const ERROR             = require("./controller/Ambito/Error")
        const TIPO_OPERACION    = require('./controller/Enums/TipoOperacion');
        const TIPO_VALOR                = require('./controller/Enums/TipoValor');
        const TIPO_DATO                 = require('./controller/Enums/TipoDato'); //para jalar el tipo de dato
        const INSTRUCCION       = require('./controller/Instruccion/Instruccion');
%}
```

Aquí se decidio la prescedencia de operadores de las expresiones

```
/* operator associations and precedence */
%left interrogacion
%left 'or'
%left 'and'
%right 'not'
%left 'igualigual' 'diferente' 'menor' 'menorigual' 'mayor' 'mayorigual'
%left 'suma' 'menos' 'masmas' 'menosmenos'
%left 'multi' 'div' 'modulo'
%left 'exponente'
left 'parA' 'parC'
```

Y esta es una producción, como se puede observar el *$$* que funciona como return

```
ASIG
    :identificador igual EXP            {$$=INSTRUCCIONES.nuevaAsignacion($1,$3);}
    |identificador corIzq EXP corDer igual EXP;
```

Enum

Los enum o enumerables las declaramos como tipo constantes ya que no cambian y asi nos evitamos algunos fallos humanos, enumeramos los tipos de instrucción, operaciones, y tipo de dato como se puede observar en la imagen estas instrucciones se encuentran en el instrucciones.js

```
 */
const TIPO_ERROR={
    LEXICO:          'ERROR_LEXICO',
    SINTACTICO:      'ERROR_SINTACTICO',
    SEMANTICO:       'ERROR_SEMANTICO'
}
const TIPO_VALOR={
    DECIMAL:         'VAL_DECIMAL',
    CADENA:          'VAL_CADENA',
    IDENTIFICADOR:   'VAL_IDENTIFICADOR',
    BANDERA:         'VAL_BANDERA',
    ENTERO:          'VAL_ENTERO',
    CARACTER:        'VAL_CARACTER'
}

//TIPOS DE OPERACIONES QUE HAY
const TIPO_OPERACION={
    SUMA:               'OP_SUMA',
    RESTA:              'OP_RESTA',
    MULTIPLICACION:     'OP_MULTIPLICACION',
    DIVISION:           'OP_DIVISION',
    MODULAR:            'OP_MOD',
    POTENCIA:           'OP_POTENCIA',
```

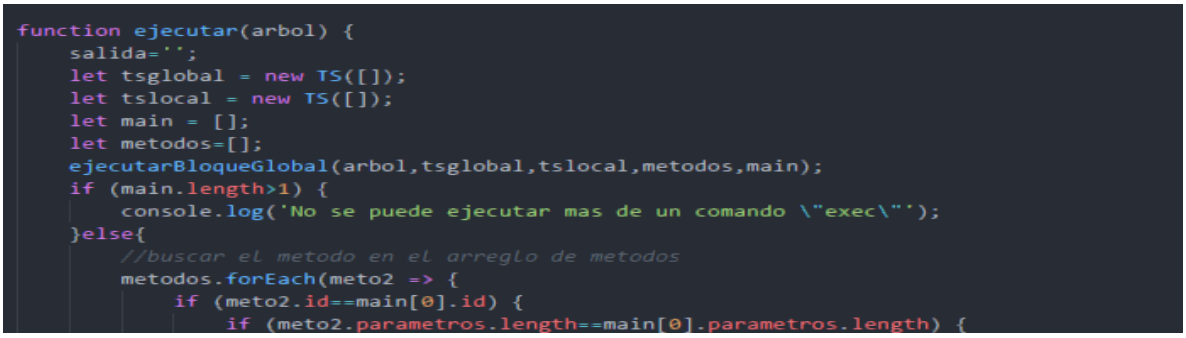
Creacion de Instrucciones

Como se observo desde la gramatica en jison se mandan valores que tienene una función con parámetros específicos para cada instrucción, acontinuacion un breve vistaso de como se realizan estas funciones.

```
const INSTRUCCIONES={
    nuevaOperacionBinaria: function(tipo,operanIzq,operanDer) {
        return {
            tipo: tipo,
            operanIzq:operanIzq,
            operanDer:operanDer
        }
    },
    nuevaOperacionUnaria: function(tipo,operanIzq) {
        return {
            tipo: tipo,
            operanIzq:operanIzq,
            operanDer:undefined
        }
    },
    nuevoValor: function(tipo,valor){
        return{
            tipo:tipo,
            valor:valor
        }
    },
    nuevaDeclaracion: function(tipo,id,expresion) {
        if (!expresion) {
```

INTERPRETE

El interprete principal recibe como entrada el árbol que genera jison y empieza a ejecutar los ámbitos la función *ejecutar(arbol)* es la encargada de empezar a ejecutar este interprete



```
function ejecutar(arbol) {
    salida='';
    let tsglobal = new TS([]);
    let tslocal = new TS([]);
    let main = [];
    let metodos=[];
    ejecutarBloqueGlobal(arbol,tsglobal,tslocal,metodos,main);
    if (main.length>1) {
        console.log('No se puede ejecutar mas de un comando \"exec\"');
    }else{
        //buscar el metodo en el arreglo de metodos
        metodos.forEach(meto2 => {
            if (meto2.id==main[0].id) {
                if (meto2.parametros.length==main[0].parametros.length) {
```

AMBITOS

La función *EjecutarBloqueGlobal*(), es la encargada de realizar las instrucciones especificas que pueden estar fuera del main como lo son instrucciones como declaración, asignación, declaración de métodos y el método main

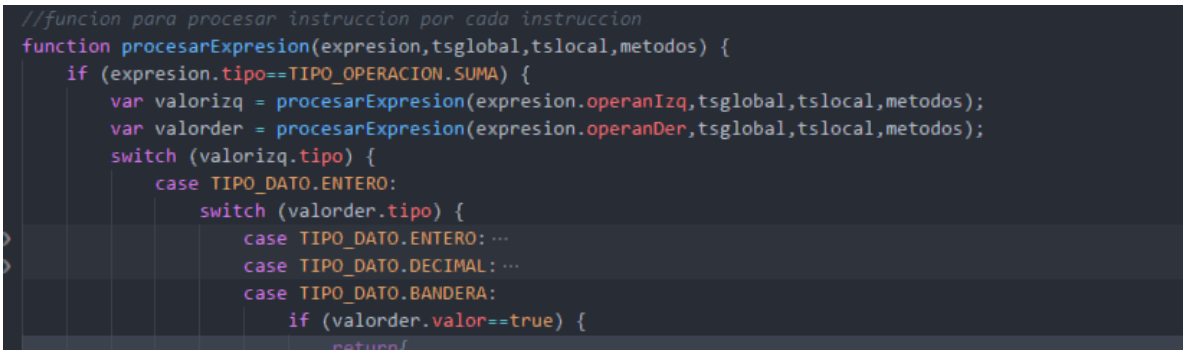
```
//DA UNA PASADA A TODO EL CODIGO PARA GUARADAR LOS METODOS Y FUNCIONES
function ejecutarBloqueGlobal(instrucciones,tsglobal,tslocal,metodos,main) {
    instrucciones.forEach((instruccion) => {
        if(instruccion.tipo==TIPO_INSTRUCCIONES.DECLARACION){
            //codigo para la declaracion
            ejecutarDeclaracionGlobal(instruccion,tsglobal,tslocal,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.ASIGNACION) {
            //codigo para Asignacion
            ejecutarAsignacionGlobal(instruccion,tsglobal,tslocal,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.METODO) {
            metodos.push(instruccion);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.MAIN) {
            main.push(instruccion);
        }
    });
}
```

Asi mismo tenemos un bloque para las instrucciones locales que siguen el mismo patrón, cada else if es un tipo de instrucción diferente y se ejecutan diferentes acciones dependiendo de la instrucción

Dentro del *ejecutarBloqueLocal* se pueden encontrar lasacciones para las instrucciones de sentencia y control, asi como sentencias cíclicas y expresiones.

```
function ejecutarBloqueLocal(instrucciones,tsglobal,tslocal,metodos) {
    for (let i = 0; i < instrucciones.length; i++) {
        var instruccion = instrucciones[i];
        if(instruccion.tipo==TIPO_INSTRUCCIONES.DECLARACION){
            //codigo para la declaracion
            ejecutarDeclaracionLocal(instruccion,tsglobal,tslocal,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.IMPRIMIR) {
            //codigo para imprimir
            ejecutarImprimir(instruccion,tsglobal,tslocal,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.FWHILE) {
            //codigo para while
            var tslocal2=new TS(tslocal._simbolos);
            ejecutarWhile(instruccion,tsglobal,tslocal2,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.DOWHILE) {
            //codigo para DOwhile
            var tslocal2=new TS(tslocal._simbolos);
            ejecutarDoWhile(instruccion,tsglobal,tslocal2,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.FOR) {
            //codigo para for
            var tslocal2=new TS(tslocal._simbolos);
            ejecutarFor(instruccion,tsglobal,tslocal2,metodos);
        }else if (instruccion.tipo==TIPO_INSTRUCCIONES.FIF) {
```

La función *procesarExpresion*() funciona y retorna el valor dependiendo de la acción u operación de la expresión realizada


//funcion para procesar instruccion por cada instruccion
function procesarExpresion(expresion,tsglobal,tslocal,metodos) {
if (expresion.tipo==TIPO_OPERACION.SUMA) {
var valorizq = procesarExpresion(expresion.operanIzq,tsglobal,tslocal,metodos);
var valorder = procesarExpresion(expresion.operanDer,tsglobal,tslocal,metodos);
switch (valorizq.tipo) {
case TIPO_DATO.ENTERO:
switch (valorder.tipo) {
case TIPO_DATO.ENTERO:
case TIPO_DATO.DECIMAL:
case TIPO_DATO.BANDERA:
if (valorder.valor==true) {