MANUAL TECNICO (Quetzal)

Quetzal es un lenguaje de programación inspirado en C, su característica principal es la inclusión de tipos implícitos. El sistema de tipos de Quetzal realiza una formalización de los tipos de C y Java. Esto permite a los desarrolladores definir variables y funciones tipadas sin perder la esencia. Otra inclusión importante de Quetzal es la simplificación de los lenguajes C y Java para poder realizar diferentes instrucciones en menos pasos.

El intérprete fue hecho con la herramienta Jison. Para instarlo debemos usar el siguiente comando

Npm install jison –g

La estructura del archivo gramatica.jison es la siguiente:

```
/* Definición lexica */
%lex
%options case-sensitive
%option yylineno
```

Aquí agregamos todas las opciones que deseamos en el interprete

```
//Expresiones regulares
num [0-9]+
id [a-zñA-ZÑ_][a-zñA-ZÑ0-9_]*

//--> Cadena validar las secuencias de escape
escapechar  [\'\"\\nrt]
escape      \\{escapechar}
aceptacion  [^\"\\]  // (^) indica que acepta todo
cadena      (\" ({escape}|{aceptacion})*\")

//--> Caracter
escapechar2  [\'\"\\nrt]
escape2      \\{escapechar2}
aceptacion2  [^\'\\]  // (^) indica que acepta todo
caracter      (\' ({escape2}|{aceptacion2})\')
```

Luego de agregar las definiciones léxicas empezamos a declarar nuestras expresiones regulares.

```
/* Comentarios */
"//".*                {/* Ignoramos los comentarios simples */}
"/*"((\*+[^/*])|([^*]))*\**"/"     {/*Ignorar comentarios con multiples lineas*/}

/* Simbolos del programa */



"++"                { console.log("Reconocio : " + yytext);  return 'INCRE' }
"--"                { console.log("Reconocio : " + yytext);  return 'DECRE' }
"=="                { console.log("Reconocio : " + yytext);  return 'IGUALIGUAL' }
"!="                 { console.log("Reconocio : " + yytext);  return 'DIFERENTE' }



"("                 { console.log("Reconocio : " + yytext);  return 'PARA' }
")"                 { console.log("Reconocio : " + yytext);  return 'PARC' }
"["                 { console.log("Reconocio : " + yytext);  return 'CORA' }
"]"                 { console.log("Reconocio : " + yytext);  return 'CORC' }
"{"                 { console.log("Reconocio : " + yytext);  return 'LLAVA' }
"}"                 { console.log("Reconocio : " + yytext);  return 'LLAVC' }
","                 { console.log("Reconocio : " + yytext);  return 'COMA' }
"."                 { console.log("Reconocio : " + yytext);  return 'PNT' }
";"                 { console.log("Reconocio : " + yytext);  return 'PYC' }
"="                { console.log("Reconocio : " + yytext);  return 'IGUAL' }
"?"                { console.log("Reconocio : " + yytext);  return 'INTERROGACION' }
":"                { console.log("Reconocio : " + yytext);  return 'DOSPUNTOS' }
```

Aquí declaramos los símbolos del programa, colocando las palabras reservadas, comentarios, espacios en blanco, todos los símbolos de expresiones regulares, etc

```
//AREA DE IMPORTS
%{


        const {Aritmetica} = require('../Expresiones/Operaciones/Aritmetica');
        const {Primitivo} = require('../Expresiones/Primitivo');
        const {Relacional} = require('../Expresiones/Operaciones/Relacionales')
        const {Logicas} = require('../Expresiones/Operaciones/Logicas')
        const {Println} = require('../Instrucciones/Println');
        const {Print} = require('../Instrucciones/Print');
        const {Tolower} = require('../Instrucciones/Tolower');
        const {Toupper} = require('../Instrucciones/Toupper');
        const {ToInt} = require('../Instrucciones/FuncionesNativas/ToInt');
        const {ToDouble} = require('../Instrucciones/FuncionesNativas/ToDouble');
        const {Round} = require('../Instrucciones/FuncionesNativas/Round');
        const {Typeof} = require('../Instrucciones/FuncionesNativas/Typeof');
        const {Tostring} = require('../Instrucciones/FuncionesNativas/Tostring');
        const {SubString} = require('../Instrucciones/SubString')
        const {TipoParse} = require('../Instrucciones/FuncionesNativas/TipoParse')
        const {CharOfPosition} = require('../Instrucciones/CharOfPosition')
        const {LenghtC} = require('../Instrucciones/LenghtC')
        const {Casteos} = require('../Instrucciones/FuncionesNativas/Casteos');
```

Despues del símbolo lex agregamos el área de imports, donde importamos las clases que vamos a utilizar para que la gramática tengan funcionalidad, se debe declarar como constante

```
/* PRECEDENCIA */

%right 'INTERROGACION'
%right 'PARA'
%right 'PNT'
%left 'OR'
%left 'ANDD' 'AND'
%right 'NOT'
%left 'IGUALIGUAL' 'DIFERENTE' 'MENORQUE' 'MENORIGUAL' 'MAYORQUE' 'MAYORIGUAL'
%left 'MAS' 'MENOS'
%left 'MULTI' 'DIV'
%left 'POT'
%right 'MOD'
%right UMINUS
```

Luego se define la asociatividad y precedencias de los operadores si la gramática es ambigua, se agrega de menor a mayor.

```
%start inicio
```

Seguido de eso le indicamos cual será nuestro símbolo inicial.

```
41  %% /* Gramática del lenguaje */
42
43  inicio : instrucciones EOF {$$ = new Ast($1); reporteGramaticalTDS.push('inicio.val := instrucciones.val EOF'); reporteGramaticalProducciones.push('<inicio> -> <instrucciones> EOF'); $$.reporteGramaticalProducci
44
45  instrucciones : instrucciones instruccion   {$$ = $1; $$.push($2); reporteGramaticalTDS.push('instrucciones.val := instrucciones.val instruccion.val'); reporteGramaticalProducciones.push('<instrucciones> -> <ins
46            | instruccion                     {$$ = new Array(); $$.push($1); reporteGramaticalTDS.push('instrucciones.val := instruccion.val'); reporteGramaticalProducciones.push('<instrucciones> -> <instruccion>
47            ;
48
49  instruccion : declaracion { $$ = $1; reporteGramaticalTDS.push('instruccion.val := declaracion.val'); reporteGramaticalProducciones.push('<instruccion> -> <declaracion>');}
50            | impresion   { $$ = $1; reporteGramaticalTDS.push('instruccion.val := impresion.val'); reporteGramaticalProducciones.push('<instruccion> -> <impresion>');}
51            | struct      { $$ = $1; reporteGramaticalTDS.push('instruccion.val := struct.val'); reporteGramaticalProducciones.push('<instruccion> -> <struct>');}
52            | asignacion_vector  { $$ = $1; reporteGramaticalTDS.push('instruccion.val := asignacion_vector.val'); reporteGramaticalProducciones.push('<instruccion> -> <asignacion_vector>');}
53            | asignacion  { $$ = $1; reporteGramaticalTDS.push('instruccion.val := asignacion.val'); reporteGramaticalProducciones.push('<instruccion> -> <asignacion>');}
54            | decl_vectores  { $$ = $1; reporteGramaticalTDS.push('instruccion.val := decl_vectores.val'); reporteGramaticalProducciones.push('<instruccion> -> <decl_vectores>');}
```

Es la última, parte de la gramatica donde cada producción incluye el código de javascript entre llaves {código_js}, la variable $$ puede tomar cualquier valor.

Para poder iniciar el flujo de nuestro programa se creó el archivo editor.js

```
const parseInput = () => {
    console.log('parsing...');
    let editorValue = currentEditor.getValue();
    const ast = gramatica.parse(editorValue);
    const controlador = new Controlador.Controlador();
    const ts_global = new TablaSimbolos.TablaSimbolos(null);
    ast.ejecutar(controlador,ts_global);
    let ts_html = controlador.graficar_ts(controlador, ts_global, "1");
    for (let tablitas of controlador.tablas) {
        ts_html += controlador.graficar_ts(controlador, tablitas, "2");
    }
    console.log(controlador.errores);
    document.getElementById("Simbolstable").innerHTML= ts_html // this is for show simbols table

    terminal.value = controlador.consola;
}


const generarAst = () => {
    console.log('Generando AST');
    let editorValue = currentEditor.getValue();
    const ast = gramatica.parse(editorValue);
    const nodo_ast = ast.recorrer();
    const grafo = nodo_ast.GraficarSintactico();
    terminalast.value = grafo;
    console.log(grafo);
}
```

La función parseInput es la que inicializa nuestro ast, controlador y llena la tabla de símbolos global, la función generarAst recorre el árbol para poder hacer el reporte en graphviz

Para poder manipular los textarea del html se declararon de la siguiente manera

```
const generateAst = document.getElementById('generateAst');
const terminal = document.getElementById('terminal');
const terminalast = document.getElementById('terminalast');
const Simbolstable = document.getElementById('Simbolstable');
```

Para mostrarlo en el html se usó esta línea para concatenar el editor.js con el index.html

```
<script type="text/javascript" src="./bundle.js"></script>
```

Se creó una clase AST para manipular el flujo de nuestro programa que tiene 3 pasadas, en la primera se guardan las funciones y métodos, la segunda ejecuta las declaraciones y por último se ejecutan todas las otras

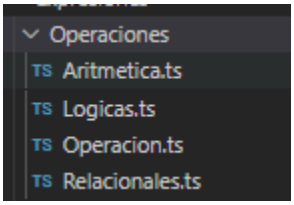
```
ejecutar(controlador: Controlador, ts: TablaSimbolos){
    let bandera_start = false;
    //1era pasada vamos a guardar las funciones y métodos del programa

    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof Funcion){
            let funcion = instruccion as Funcion;
            funcion.agregarFuncionTS(ts);
        }
    }


    //Vamos a recorrer las instrucciones que vienen desde la gramática

    //2da pasada. Se ejecuta las declaraciones de variables
    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof Declaracion){
            instruccion.ejecutar(controlador,ts);
        }
    }
    //3era pasada ejecutamos las demás instrucciones
    for(let instruccion of this.lista_instrucciones){
        if(instruccion instanceof Fmain && !bandera_start){
            instruccion.ejecutar(controlador,ts);
            bandera_start = true;
                                (alias) class Declaracion
                                import Declaracion
        } else if(!(instruccion instanceof Declaracion) && !(instruccion instanceof Funcion) && bandera_start){
            instruccion.ejecutar(controlador,ts);
        }else if(bandera_start){
            let error = new Errores("Semantico",`Solo se puede colocar un main.`,0,0);
            controlador.errores.push(error);
            controlador.append(`ERROR: Semántico, Solo se puede colocar un main.`);

        }


    }
    if(bandera_start == false){
        let error = new Errores("Semantico",`Se debe colocar un void main() para correr el programa.`,0,0);
            controlador.errores.push(error);
            controlador.append(`ERROR: Semántico, Se debe colocar un void main() para correr el programa.`);

    }
}
recorrer():Nodo{
    let raiz = new Nodo("INICIO","");

    for(let inst of this.lista_instrucciones){
        raiz.AddHijo(inst.recorrer())
    }
    return raiz;
}
```

SE creó la clase expresión la cual contiene dos métodos, getTipo para que nos devuelva el valor de la expresión y getValor para que nos devuelva el valor de la expresión

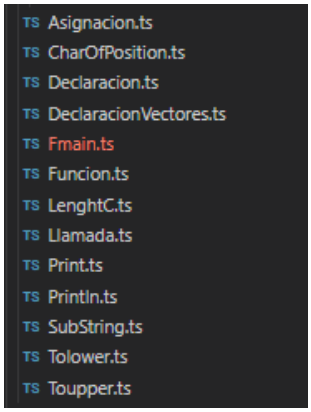
```typescript
export interface Expresion{

    /**
     * @function getTipo nos devuelve el tipo del valor de la expresion
     * @param controlador llevamos todo el control del programa
     * @param ts accede a la tabla de símbolos
     */

    getTipo(controlador : Controlador,ts: TablaSimbolos): tipo;
    /**
     * @function getValor nos devuelve el valor de la expresion
     * @param controlador llevamos todo el control del programa
     * @param ts accede a la tabla de símbolos
     */


    getValor(controlador : Controlador,ts: TablaSimbolos): any;

    recorrer() : Nodo;
}
```

SE creó la clase Instrucción la cual contiene un método. Ejecutar, la clase expresión y la clase instrucción indican que hacer con cada clase que extienda de estas.

Se crearon varias clases por cada operación que admite el programa, todas estas extienden de Expresión

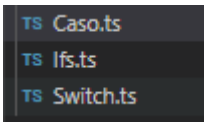


Se crearon varias clases para cada instrucción, estas extienden de instrucción

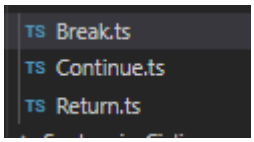


Se crearon varias clases para las sentencias cíclicas, extienden de instrucción

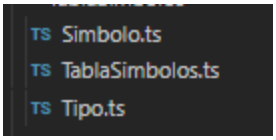
Se crearon varias clases para las sentencias de control, extienden de instrucción



Se crearon varias clases para las sentencias de control, extienden de instrucción



Para manejar la tabla de símbolo se creo una clase símbolo. Y en la clase tabla de símbolos se crearon métodos (agregar,existe,getsimbolo,existenactual) para manipular los datos:




```
export default class TablaSimbolos{
    public ant: TablaSimbolos;
    public tabla: Map<string,Simbolo>;

    //en la tabla vamos a ir guardando el nombre y todo lo que tiene
    //x , (x,0,entero)
    //y , (y,0,entero)
    //z , (z,0,entero)

    /**
     * @constructor creamos una nueva tabla.
     * @param ant indica cual es la tabla de simbolos anterior de la nueva tabla que nos servirá para le manejo de ambitos
     * Le mandamos una tabla global y otra local
     */

    constructor(ant : TablaSimbolos | any){
        this.ant = ant;
        this.tabla = new Map<string,Simbolo>();
    }

    agregar(id: string, simbolo: Simbolo){
        this.tabla.set(id.toLowerCase(),simbolo); //usamos todo minúscula porque nuestro lenguaje es caseinsensitive
    }

    existe(id: string): boolean{  // Con esto buscamos si existe la variable
        let ts: TablaSimbolos = this;

        while(ts != null){
            let existe = ts.tabla.get(id.toLowerCase());
            if(existe != null){
                return true;
            }
            ts = ts.ant
        }
        return false;
    }
```

La función agregar, agrega el símbolo a la tabla de símbolos para poder usarlo en cualquier momento que se requiera

La función existe verifica si existe el id, buscando en la tabla de símbolos la variable, si la encuentra devuelve true y si no devuelve false

La función getSimbolo obtiene el símbolo asociado al id, que contiene toda la información de este.

La función existenActual verifica si existe el id, buscando en la tabla de símbolos local la variable, si la encuentra devuelve true y si no devuelve false.

```
getSimbolo(id: string){
    let ts: TablaSimbolos = this;

    while(ts != null){
        let existe = ts.tabla.get(id.toLowerCase());
        if(existe != null){
            return existe;
        }
        ts = ts.ant
    }
    return null;
}

existeEnActual(id:string):boolean{
    let ts: TablaSimbolos = this;
    let existe = ts.tabla.get(id.toLowerCase());
    if(existe != null){
        return true;
    }
    return false;

}
```

TRADUCCION 3D

```
    traducir(controlador: Controlador, ts: TablaSimbolos): String {
        let c3d = ``;
        let funciones = `/*------FUNCIONES------*/\n`
        let temporales =`double `
        let cuerpo =``
        let encabezado = `#include <stdio.h> //Importar para el uso de Printf
#include <math.h> //Importar para el uso de libreria matematicas
float heap[16384]; //Estructura para heap
float stack[16394]; //Estructura para stack
float p; //Puntero P
float h; //Puntero H
`

let impresion = `void printString() {
    t0 = p+1;
    t1 = stack[(int)t0];
    L1:
    t2 = heap[(int)t1];
    if(t2 == -1) goto L0;
    printf("%c", (char)t2);
    t1 = t1+1;
    goto L1;
    L0:
    return;
}\n\n`

        for(let instruccion of this.lista_instrucciones){
            if(instruccion instanceof Funcion){
                let funcion = instruccion as Funcion;
                funcion.agregarFuncionTS(ts);
                funciones += instruccion.traducir(controlador,ts);
            }
        }
        let cantidadGlobales = 0;

        for(let instruccion of this.lista_instrucciones){
            if(instruccion instanceof Declaracion){
                c3d += instruccion.traducir(controlador,ts);

            }
```

Se implementó la función traducir en el AST la cual inicializa el código 3d concatenando en nuestra variable c3d todo el código traducido.

En la primera pasada traduce funciones, luego declaraciones y por último todo lo que se encuentra en el main

```
let cantidadGlobales = 0;

for(let instruccion of this.lista_instrucciones){
    if(instruccion instanceof Declaracion){
        c3d += instruccion.traducir(controlador,ts);

    }
}


ts.ambito = false;
for(let instruccion of this.lista_instrucciones){
    if(instruccion instanceof Fmain ){
        cuerpo += instruccion.traducir(controlador,ts)
    };
}


let conttemp = 0;
while(conttemp < (ts.temporal)){
    temporales += `t${conttemp}, `
    conttemp = conttemp +1;

    if (conttemp == (ts.temporal)){
        temporales += `t${conttemp};\n\n`
    }
}

c3d += encabezado
c3d += temporales
c3d += impresion
c3d += funciones
c3d += cuerpo



return c3d
}
```

Al final del método concatenamos el encabezado, la lista de temporales, el método de impresión, funciones y por último todas las instrucciones en el main

```
traducir(controlador: Controlador, ts: TablaSimbolos): String {
    let estructura = 'heap';

    let codigo = '';
    //let condicion = this.expresion.traducir(controlador,ts);
    //codigo += condicion;
    let temp = ts.getTemporalActual();
    let temp2 = ts.getTemporalActualint()


    if(this.expresion.getTipo(controlador,ts) == tipo.ENTERO || this.expresion.getTipo(controlador,ts) == tipo.BOOLEAN){
        codigo +=  this.expresion.traducir(controlador,ts)
        codigo += `    printf(\"%d\\n\", (int)t${temp2+1});\n`
        ts.QuitarTemporal(temp);
    }else if(this.expresion.getTipo(controlador,ts) == tipo.DOBLE ){
        codigo +=  this.expresion.traducir(controlador,ts)
        codigo += `    printf(\"%f\\n\", (double)t${temp2+1});\n`
        ts.QuitarTemporal(temp);
    }
    // else if(this.expresion.getTipo(controlador,ts) == tipo.DOBLE){
    //     codigo += `printf(\"%f\\n\", ${temp});\n`
    //     ts.QuitarTemporal(temp);
    // }

    else if(this.expresion.getTipo(controlador,ts) == 4 ){
        let c3d = ``;
        const temporal = ts.getTemporal();
        let temp4 = ts.getTemporal();
        let temp5 = ts.getTemporal();


        let x = 0;
        c3d += `    ${temporal} = h;\n`
        while(x < this.expresion.getValor(controlador,ts).length){

            c3d += `    heap[(int)h] = ${this.expresion.getValor(controlador,ts).charCodeAt(x)};\n`
            c3d += `    h = h+1;\n`
            x = x+1;
        }
        c3d += `    heap[(int)h] =-1;\n`;
        c3d += `    h = h+1;\n`
        c3d += `    ${temp4} = p+ ${ts.getStackActual()};\n`
        c3d += `    ${temp4} = ${temp4}+1;\n`
        c3d += `    stack[(int)${temp4}] =  ${temporal};\n`
        c3d += `    p = p+${ts.getStackActual()};\n`
        c3d += `    printString();\n`
        c3d += `    ${temp5} = stack[(int)p];\n`
        c3d += `    p = p-${ts.getStackActual()};\n`
        c3d += `    printf("%c", (char)10);\n`
        codigo += c3d;

    }
```

Para la traducción de un println verificamos primero si es de tipo entero o booleano, si es uno de ellos, traducimos la expresión y luego la imprimimos con %d, si es de tipo double la imprimimos con %f y si es de tipo cadena usamos temporales para guardar sus valores char en el heap.

Concatenamos todo al c3d y retornamos ese string.