Proyecto 2: TypeWise

Manual Técnico

2023

ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1

Damaris Julizza

Muralles Véliz

202100953

CONTENIDO

INTRODUCCION

Este manual, describe todos los aspectos técnicos del programa, a manera de familiarizar a la persona interesada con la lógica implementada en el desarrollo de este.

DESCRIPCIÓN GENERAL DEL PROGRAMA

Este programa es un intérprete sencillo capaz de reconocer un lenguaje de programación, el texto/código reconocido se analiza de forma léxica y sintáctica creando así el árbol de análisis sintáctico, tabla de símbolos y tabla de errores para dicho código.

La.

DETALLES DE DESARROLLO

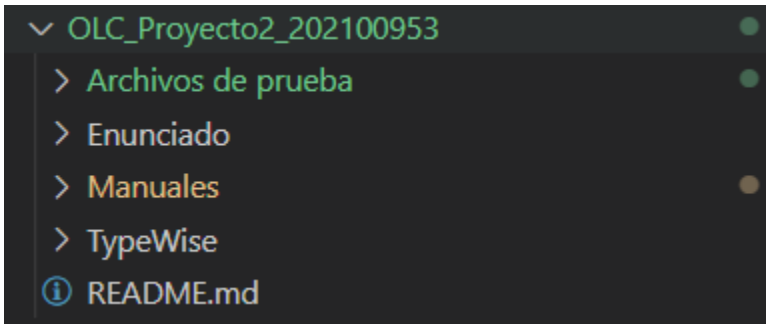
Este programa se desarrolló en lenguaje JavaScript, a continuación, se listan las versiones del IDE y librerías utilizadas en el desarrollo:

- IDE: Visual Studio Code V 1.77.3
- JavaScript
- CSS
- HTML
- Bootstrap v.5.1.3
- d3 v.7
- Grahpviz v.3
- CodeMirror v 5.65.13.

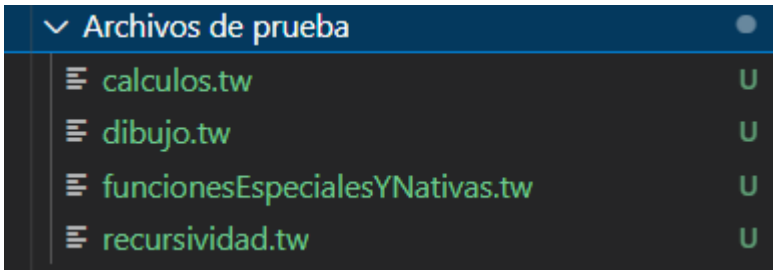
Lo detallado anteriormente, puede asegurar el correcto funcionamiento del programa.

LOGICA DEL PROGRAMA

En esta sección se explicará a detalle los paquetes, clases y archivos importantes dentro del programa:

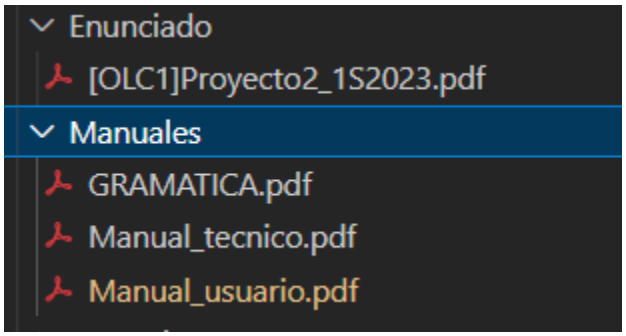


PAQUETE ARCHIVOS DE PRUBA:



Este paquete contine varios archivos de prueba para el programa.

PAQUETE ENUNCIADO Y MANUALES

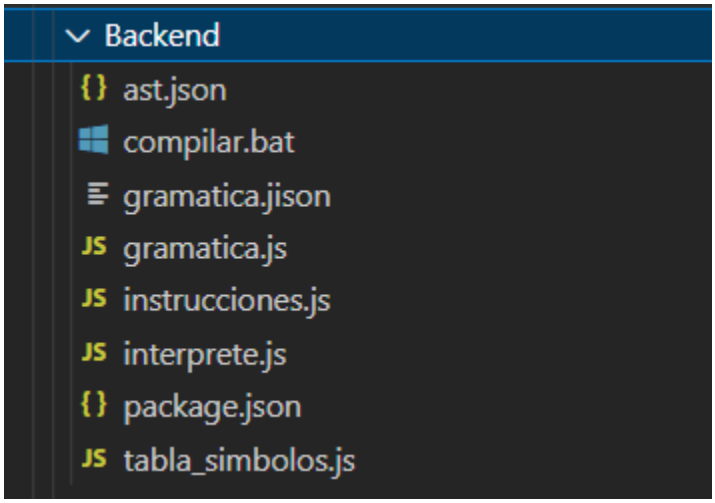


En estas carpetas se encuentran los detalles importantes para el programa, se puede encontrar el enunciado, el manual de usuario, un archivo con la gramática utilizada para los análisis léxico y sintáctico, y el presente manual.

PAQUETE TYPEWISE

En la carpeta TypeWise podremos encontrar el código que modela el programa. Dentro podemos encontrar otras dos subcarpetas que se detallaran a continuación:

BACKEND



En esta parte se encuentra el código correspondiente a los analizadores, léxico y sintáctico, el código de la estructura para formar el árbol AST, y el código para procesar cada instrucción analizada.

- ***GRAMATICA.JISON:***

  En este archivo se contiene el código jison que corresponde a los analizadores léxico y sintáctico.

  En la primera parte se encuentra la definición léxica, palabras reservadas y símbolos utilizados, esto corresponde al análisis léxico que se realiza a la entrada.

```
%{

    let ubicacion;
    let prevToken = '';
%}




%lex

%options case-insensitive

%%
[\s\r\t]+                                           // se ignoran espacios en blanco
\n                                                  // se ignoran saltos de linea
"//".*                                              // comentario simple línea
[/][*][^*]*[*]+([^/*][^*]*[*]+)*[/]                 // comentario multiple líneas

//============PALABRAS RESERVADAS=============================================

"int"               return 'T_INT'; //ya
"double"            return 'T_DOUBLE'; //ya
"boolean"           return 'T_BOOLEAN'; //ya
"char"              return 'T_CHAR'; //ya
"string"            return 'T_STRING'; //ya
"switch"            return 'T_SWITCH'; //ya
```

También se definieron las siguientes expresiones regulares:

```
//=========================EXPRESIONES=============================================

[0-9]+\.[0-9]+\b    return 'DECIMAL';
[0-9]+\b                return 'ENTERO';
"true"                  return 'BOOLEANO';
"false"                 return 'BOOLEANO';
([a-zA-Z])[a-zA-Z0-9_]* {
    yytext = yytext.toLowerCase();
    return 'IDENTIFICADOR';
}


\'[^\']\' { yytext = yytext.substr(1,yyleng-2).toLowerCase(); return 'CARACTER'; }

\"(\\.|[^"\\])*\" {
    yytext = yytext.substr(1,yyleng-2);
    yytext = yytext.replace(/\\\\/g, '\\');
    yytext = yytext.replace(/\\n/g, '\n');
    yytext = yytext.replace(/\\t/g, '\t');
    yytext = yytext.replace(/\\"/g, '\"');
    yytext = yytext.replace(/\\'/g, '\'');
    return 'CADENA';
}
```

Este código es el encargado de capturar los errores léxicos encontrados en la entrada analizada:

```
//==============================ERRORES LEXICOS=============================================
<<EOF>>      return 'EOF';
.            { console.log('Error léxico: ' + yytext + ', en la linea: ' + yylloc.first_line +
                $$ = instruccionesAPI.parseError(yytext, yylloc, yy,"lexico", 'Error léxico: '
             }

/lex
```

Las precedencias utilizadas en el lenguaje fueron las siguientes:

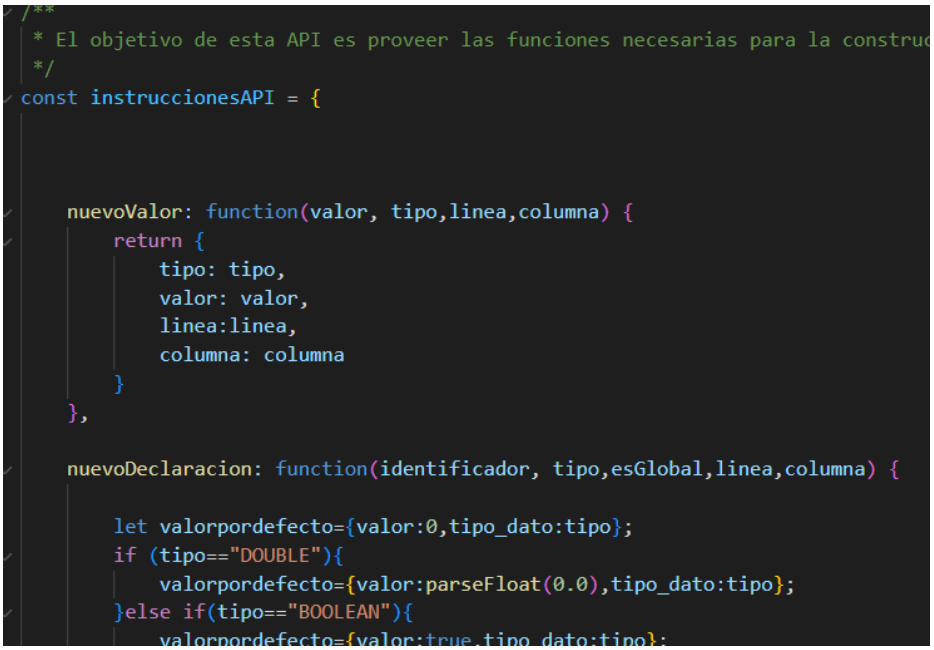
```
/*============================== PRESEDENCIA====== */
%nonassoc PARIZQ
%nonassoc 'INTERROGACION'
%left 'OR'
%left 'AND'
%right 'NOT'

%left 'DOBLEIG', 'NOIG', 'MENQUE', 'MENIGQUE', 'MAYQUE', 'MAYIGQUE'
%left 'MAS', 'MENOS'
%left 'DIVIDIDO', 'POR', "MODULO"
%nonassoc 'POTENCIA'
%right 'UMENOS'
```

Después de las precedencias se detalla la gramática que corresponde al análisis sintáctico:

```
/* ======================================GRAMATICA======= */
%start ini

%%

ini
    : instrucciones EOF {
        // cuado se haya reconocido la entrada completa retornamos el AST
        return $1;
    }
;


instrucciones
    : instrucciones instruccion     { $1.push($2); $$ = $1; }
    | instruccion                   { $$ = [$1]; }
;

instruccion
    : declaracion  PTCOMA                { $$ = $1; }
    | asignacion   PTCOMA                { $$ = $1; }
    | funcion_main     PTCOMA              { $$ = $1; }
    | funcion                    { $$ = $1; }
    | metodo { $$ = $1; }
    |vectores { $$ = $1; }
```

➢ *INSTRUCCIONES.JS:*

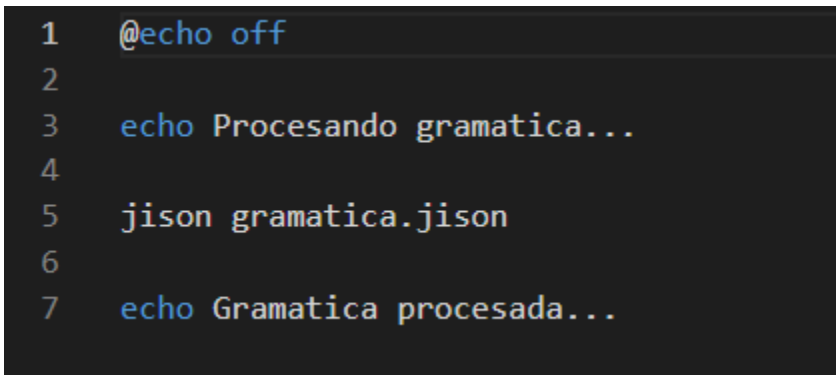
Este archivo contine la instruccionesAPI que es un objeto que organiza la información requerida para las instrucciones analizadas y mandarlos al árbol AST.



```
/**
 * El objetivo de esta API es proveer las funciones necesarias para la construc
 */
const instruccionesAPI = {



    nuevoValor: function(valor, tipo,linea,columna) {
        return {
            tipo: tipo,
            valor: valor,
            linea:linea,
            columna: columna
        }
    },

    nuevoDeclaracion: function(identificador, tipo,esGlobal,linea,columna) {

        let valorpordefecto={valor:0,tipo_dato:tipo};
        if (tipo=="DOUBLE"){
            valorpordefecto={valor:parseFloat(0.0),tipo_dato:tipo};
        }else if(tipo=="BOOLEAN"){
            valorpordefecto={valor:true,tipo_dato:tipo};
```

➢ *COMPILAR.BAT:*

En este archivo se listan unas instrucciones para poder generar el archivo gramática.js , este archivo es el mismo código de la gramática.jison pero en lenguaje de JavaScript.



```
1   @echo off
2
3   echo Procesando gramatica...
4
5   jison gramatica.jison
6
7   echo Gramatica procesada...
```

➢ *INTERPRETE.JS:*

El código en este archivo se encarga de procesar la informacion que se obtuvo de los análisis léxico y sintactico, para ello se analiza cada seccion del árbol AST, y dependiento del tipo de instrucción se realiza una serie de

códigos de forma recursiva para procesar la información como corresponde.

La función principal procesarInstruccionesGlobales recibe la lista de instrucciones del AST y analiza cada una por medio de un forEch.

```
function procesarInstruccionesGlobales(instrucciones, tablaDeSimbolos,consola) {
    consoleEditor=consola;
    instrucciones.forEach(instruccion => {
        if (erroreslist.length==0){
            if (instruccion.tipo === TIPO_INSTRUCCION.FUNCIONES) {

                procesarDeclaracionFuncion(instruccion, tablaDeSimbolos);
            } else if (instruccion.tipo === TIPO_INSTRUCCION.METODOS) {

                procesarDeclaracionFuncion(instruccion, tablaDeSimbolos);
            }
        }

    });
```

Se hacen un total de 4 recorridos: el primero para procesar la declaración de funciones y métodos, de forma que puedan ser utilizados después.

El segundo es para la declaración y asignación de variables y estructuras.

El tercero es para procesar la función o método que este declarado en la instrucción main.

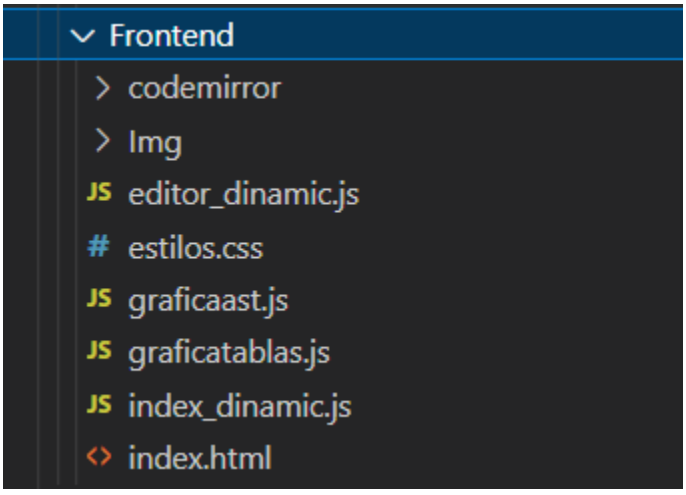
Y el cuarto recorrido es para analizar cada bloque de instrucciones en las funciones, métodos, ciclos y sentencias de control.

- ***TABLA_SIMBOLOS.JS:***

  Este archivo contine el código de la clase TS, o tabla de símbolos, este es el encargado de crear una tabla y almacenar la información para todos los símbolos/variables/funciones/métodos encontrados al procesar la información.

```
function crearSimbolo(id, tipo,tipodato,entorno,linea,columna, valor) {···
}

class TS {

    constructor (simbolos,padre) {
        this._simbolos = simbolos;
        this.padre = padre;
    }

    esLocal() {···
    }

    agregar(id, tipo, tipodato,entorno,linea,columna) {···
    }
    buscar(id) {···
    }
    actualizar(id, valor, linea, columna) {···
    }

    obtener(id, linea, columna) {···
    }
    get simbolos() {···
    }
}
```

FRONTEND



Aquí se encuentra el código correspondiente a la interfaz, podemos encontrar el código HTML, los estilos CSS aplicados y los códigos de JavaScript que hacen dinámica la interfaz.

- ***INDEX.HTML:***

  Contine el código que modela la página web.

- ***ESTILOS .CSS:***

  Es el código CSS que proporciona los estilos aplicados a la página HTML para mejorar su aspecto.

- ***INDEX_DINAMIC .JS:***

  Este archivo de Javascript contiene el código que hace dinámica página HTML, el código dentro es el encargado de modificar el DOM dependiendo lo que se esté realizando.

  Aquí también se crean las distintas funcionalidades para poder crear nuevas pestañas con editores de texto independiente, y botones independientes.

  La función agregada a los botones de ejecutar también es el encargado de iniciar el parser para los analizadores con la siguiente línea de código:

```
ast = gramatica.parse(entrada.toString());
```

  Dependiendo de si existen errores o no se comenzara la ejecución de la función procesarInstruccionesGlobales para procesar las instrucciones:

```
if (errores.length>0){
  const stringd = errores.map(item => item.consola).join("\n");
  consoleEditor.setValue("");
  consoleEditor.setValue(stringd);
  grapherror=generateERRORES(errores, 0) ;
}else{
// Procesamos las instrucciones reconocidas en nuestro AST
  grapharbol=generateDot(ast);
  consoleEditor.setValue("");
  let res=procesarInstrucionesGlobales(ast, tsGlobal,consoleEditor);
  graphts= res.ts;
  grapherror=res.errort;
}
```

En el caso de existir errores estos se mostrarán en consola y se generara un código DOT para graficar la tabla correspondiente.

- ***GRAFICAAST .JS/GRAFICATABLAS.JS:***

  En estos dos archivos se encuentran las funciones y códigos de apoyo para generar una cadena de texto con un formato dot para poder graficar en Graphviz.

```
function generateDot(instructions) {
  let dot = 'digraph G {\n';
  dot += `    bgcolor="#1c1c1c";
  node [shape=record, style=filled, fillcolor="#0e9df6", fontcolor=white]
  edge [color=white]`;
  dot += '   start [label="INSTRUCCIONES"]\n';
  let nodeId = 0;
  instructions.forEach((instruction, index) => {

    if (instruction.tipo=="SENTENCIA_MAIN"){
      dot += `node${index} [label="${instruction.tipo}"];\n`;
      dot += `node${index}_a [label="${instruction.id}"];\n   node${index}-> node${index}_a\n`;
      dot += generateInstructionDot([instruction.run], index);

      dot += `\n   start -> node${index}\n`;
    }else if(instruction.tipo=="INSTR_DECLARACION"){···
    }else if(instruction.tipo=="INSTR_DECLARACION_CON_ASIGNACION"){···
```

```
function generateERRORES(instructions, op) {
    let dot = 'digraph G {\n';
    dot += `    bgcolor="#1c1c1c";···
                </tr>`;

    instructions.forEach((instruction, index) => {
      if(op==1){
        dot += `
        <tr>
            <td bgcolor="white">${index+1}</td>
            <td bgcolor="white">${instruction.tipo}</td>
            <td bgcolor="white">${instruction.descripcion}</td>
            <td bgcolor="white">${instruction.lineaerror}</td>
            <td bgcolor="white">${instruction.columnaerror}</td>
        </tr>`;
      }else{···
      }


    });
    dot += `          </table>>];
}`;
    return dot;
}

function generateSimbolos(instructions) {···
}
```

- ***PAQUETE CODEMIRROR Y IMG:***

  En la carpeta llamada codemirror se posee el código de la herramienta codemirror, esta fue descargada de la página oficial: https://codemirror.net/

  En cuanto al paquete img, contiene las imágenes que fueron utilizados en la interfaz.