# MANUAL TECNICO

## Proyecto No. 2
## LABORATORIO ORGANIZACION DE LENGUAJES Y COMPILADORES 1 Sección C

*José Eduardo Galdámez González*
*Carne: 202109732*

# Manual Técnico:

El manual técnico que se presenta tiene como objetivo guiar a los desarrolladores en la implementación del lenguaje de programación TypeWise, que será utilizado como un intérprete para los estudiantes de OLC1 de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala.

En términos generales, el objetivo del 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. Específicamente, se busca 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 y producir las salidas esperadas, y aplicar la teoría de compiladores para la creación de soluciones de software.

El entorno de trabajo incluye un editor de texto, que permitirá a los usuarios ingresar el código fuente que será analizado. El editor se mostrará en el navegador y podrá abrir diferentes archivos al mismo tiempo, mostrando la línea actual. El diseño del editor queda a discreción del estudiante.

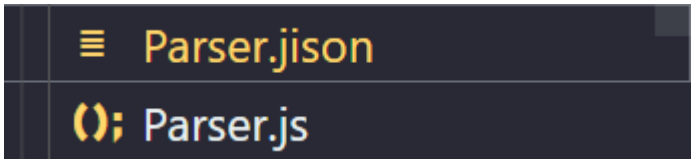
En resumen, este manual técnico proporciona información detallada sobre la implementación del lenguaje de programación TypeWise, así como el entorno de trabajo que se utilizará. El objetivo final es proporcionar una herramienta funcional y eficaz para los estudiantes de programación, utilizando los conceptos y técnicas de compiladores y análisis léxico y sintáctico.

## Manejo de área léxico y sintáctico:

En esta área se manejó con JISON es un generador de analizadores sintácticos para el lenguaje JavaScript. Es una herramienta que permite definir gramáticas formales y producir un analizador sintáctico capaz de leer cadenas de texto que siguen esa gramática.

En otras palabras, JISON se utiliza para crear analizadores sintácticos para lenguajes de programación. El objetivo de un analizador sintáctico es analizar el código fuente de un programa y determinar si cumple con la sintaxis y las reglas del lenguaje.

JISON se basa en la teoría de los autómatas finitos y los análisis LL(k), y proporciona una forma sencilla de generar analizadores sintácticos en JavaScript, lo que lo hace muy útil para proyectos que utilizan JavaScript como lenguaje de programación. Además, JISON es compatible con una amplia variedad de plataformas, lo que lo convierte en una herramienta muy versátil y fácil de usar para el análisis de código.



```
%lex
%options case-sensitive

digit                       [0-9]
cor1                        "["
cor2                        "]"
esc                         "\\"
int                         (?:[0-9]|[1-9][0-9]+)
exp                         (?:[eE][-+]?[0-9]+)
frac                        (?:\.[0-9]+)

%%

\s+                             {/* skip whitespace */}
<<EOF>>                         {return 'EOF';}

/* COMENTARIOS */
"//".*                                  {/* IGNORE */}
[/][*][^*]*[*]+([^/*][^*]*[*]+)*[/]     {/* IGNORE */}

/* ================== PALABRAS RESERVADAS ================== */
"true"                          {   return 'ttrue';     }
"false"                         {   return 'tfalse';    }
"int"                           {   return 'tinteger';  }
"boolean"                       {   return 'tboolean';  }
"double"                        {   return 'tdouble';   }
"String"                        {   return 'tstring';   }
"char"                          {   return 'tchar';   }
"if"                            {   return 'tif';       }
"while"                         {   return 'twhile';    }
"for"                           {   return 'tfor';    }
"else"                          {   return 'telse';     }
"void"                          {   return 'tvoid';     }
"return"                        {   return 'treturn';   }
"new"                           {   return 'tnew';     }
"do"                            {   return 'tdo';     }
"list"                          {   return 'tlist';     }
"add"                           {   return 'tadd';     }
"switch"                        {   return 'tswitch';     }
"case"                        {   return 'tcase';     }
"default"                       {   return 'tdefault';     }
"toLower"                     {   return 'ttoLower';     }
"toUpper"                     {   return 'ttoUpper';     }
"truncate"                      {   return 'ttruncate';     }
"round"                     {   return 'tround';     }
"length"                      {   return 'tlength';     }
"typeOf"                      {   return 'ttypeOf';     }
"toString"                      {   return 'ttoString';     }
"toCharArray"                       {   return 'ttoCharArray';     }
"main"                          {   return 'tmain';     }
"print"                           {   return 'tPrint';    }
"break"                           {   return 'tBreak';    }
```

```
/* ================== EXPRESIONES REGULARES ================== */
([a-zA-ZÑñ]|("_"[a-zA-ZÑñ]))([a-zA-ZÑñ]|[0-9]|"_")*             yytext = yytext.toLowerCase();          return 'id';
\"(?:[{cor1}|{cor2}]|["\\"]["bnrt/["\\"]]|[^"["\\"])*\"        yytext = yytext.substr(1,yyleng-2);     return 'cadena';
\'(?:{esc}["bfnrt/{esc}]|{esc}"u"[a-fA-F0-9]{4}|[^"{esc}])\'    yytext = yytext.substr(1,yyleng-2);     return 'caracter'
{int}{frac}\b                                                                                       return 'decimal'
{int}\b                                                                                             return 'entero'

//Error                                                                                         return 'entero'

/* ======================= SIGNOS ======================= */
"."                         {return '.';}
"++"                        {return '++';}
"--"                        {return '--';}
"+"                         {return '+';}
"-"                         {return '-';}
"*"                         {return '*';}
"/"                         {return '/';}
"^"                         {return '^';}
"%"                         {return '%';}
"("                         {return '(';}
")"                         {return ')';}
"=="                        {return '==';}
"="                         {return '=';}
","                         {return ',';}
":"                         {return ':';}
";"                         {return ';';}
"?"                         {return '?';}
"||"                        {return '||';}
"&&"                        {return '&&';}
"!="                        {return '!=';}
"!"                         {return '!';}
"<="                        {return '<=';}
">="                        {return '>=';}
">"                         {return '>';}
"<"                         {return '<';}
"{"                         {return '{';}
"}"                         {return '}';}
"["                         {return '[';}
"]"                         {return ']';}
. {  }


/lex
```

```
/lex

/* ================= ASOCIATIVIDAD y PRECEDENCIA DE OPERADORES ===============
/*Operaciones logicas*/
%left '||'
%left '&&'
%left '?'
%left ':'
%left '++' '--'
%left '!=' '==' '===' 
%left '>' '<' '<=' '>='

/*Operaciones numericas*/
%left '+' '-'
%left '*' '/' '%'
%right '^'
%right negativo '!' '('



%start INICIO

%% /* language grammar */

INICIO
    : SENTENCIAS EOF{ $$ = { val: 0, node: newNode(yy, yystate, $1.node,'EOF')}; return $$;}
    | EOF{ $$ = { val: 0, node: newNode(yy, yystate,'EOF')}; return $$;}
;

SENTENCIAS :    SENTENCIAS SENTENCIA
            {
                $$ = { val: 0, node: newNode(yy, yystate, $1.node, $2.node)};
            }
            |   SENTENCIA
            {
                $$ = { val: 0, node: newNode(yy, yystate, $1.node)};
            }
            | error {  }
;
```

## Implementación para el uso de backend y frontend del proyecto:

En el proyecto TypeWise, se utilizaron Node.js, TypeScript y el patrón de diseño Intérprete para crear un intérprete de lenguaje de programación.

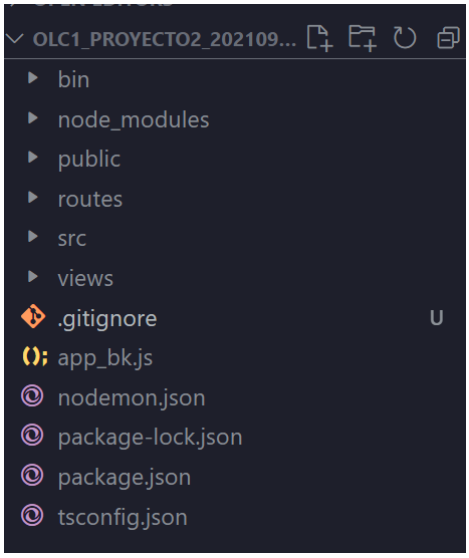
Node.js es un entorno de tiempo de ejecución de JavaScript que se utiliza para crear aplicaciones del lado del servidor. En este proyecto, Node.js se utilizó para ejecutar el intérprete en el servidor, lo que permitió que los estudiantes enviaran su código fuente al servidor para su análisis.

TypeScript es un lenguaje de programación que se basa en JavaScript y que proporciona herramientas para la escritura de código más seguro y legible. En este proyecto, TypeScript se utilizó para escribir el intérprete, lo que permitió la definición de tipos de datos y la validación estática del código.

El patrón de diseño Intérprete se utilizó para analizar el código fuente de los estudiantes y ejecutarlo. El intérprete se dividió en dos partes: el analizador sintáctico y el evaluador de expresiones. El analizador sintáctico se encargó de leer el código fuente y construir un árbol sintáctico que representara la estructura del programa. El evaluador de expresiones se encargó de recorrer el árbol sintáctico y ejecutar las instrucciones.

La combinación de Node.js, TypeScript y el patrón de diseño Intérprete permitió la creación de un intérprete de lenguaje de programación completo y funcional que se ejecuta en un entorno de servidor. El uso de TypeScript permitió una mayor seguridad y legibilidad del código, mientras que el patrón de diseño Intérprete permitió la creación de un intérprete modular y fácil de mantener.

## Estructura del proyecto Node JS:





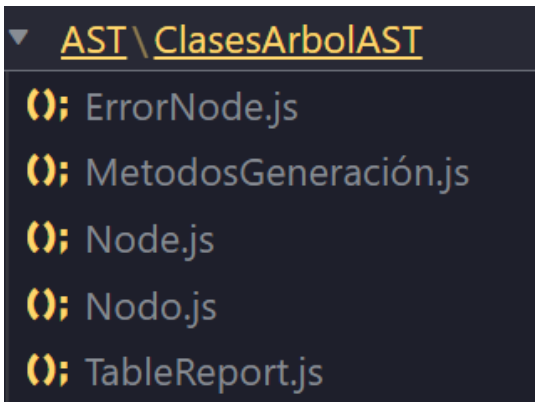
```
app.listen(port, (err) => {
    if (err) {
      console.error(err);
      return;
    }
    console.log(`Servidor funcionando en puerto: ${port}`);
  });
```

**Uso de TypeScript:** en este caso se dividió en cada forma de trabajo que se iba aplicar y el manejo del lenguaje.

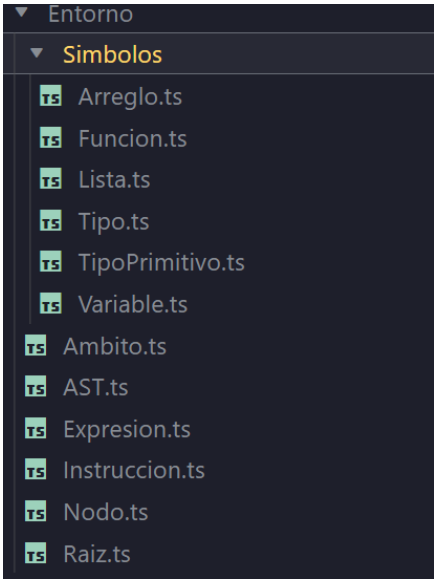
- **Analizador y AST:** encargado de ver toda el área mandar la data al js de gramática y léxica. Y manejar todo lo relacionado de esa área como construcción de árbol ast.

```
import { AST } from '../Entorno/AST';
import { Raiz } from '../Entorno/Raiz';
import Parser from './Parser';

export class Analizador
{
    entrada     :string;        // Entrada ha analizar
    archivo     :string;        // Archivo de la entrad

    constructor(entrada:string,archivo:string)
    {
        this.archivo = archivo;
        this.entrada = entrada;

    }
```

```
public Analizar(): any
{
    // let parser:any = require("./Parser").parser;
    let parser:any  = Parser.parser;
    let arbol:AST   = undefined;

    try{
        let raiz:Raiz    = parser.parse(this.entrada);
        arbol = new AST(raiz);

        // Ejecutar
        arbol.ejecutar();

        return arbol;
    }catch(e){
        console.log(e);
        return false;
    }
}
```

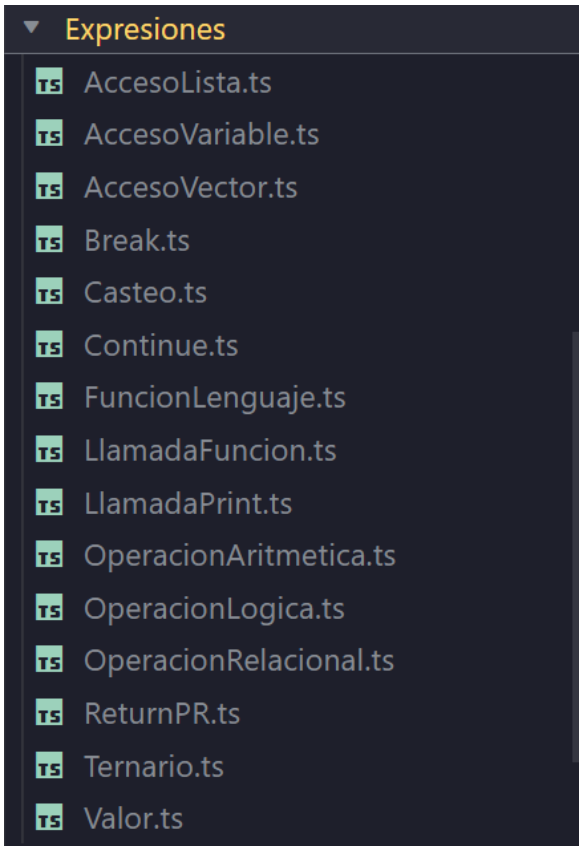



- **Entorno y símbolos:** es la esencia del patrón intérprete ya que este contiene las clases abstractas que apoyan a las demás clases para determinar si la sentencia procesada se trata de una instrucción o una expresión. Así como la clase ámbito, la cual hace referencia al entorno en el cual se encuentra en ese momento.

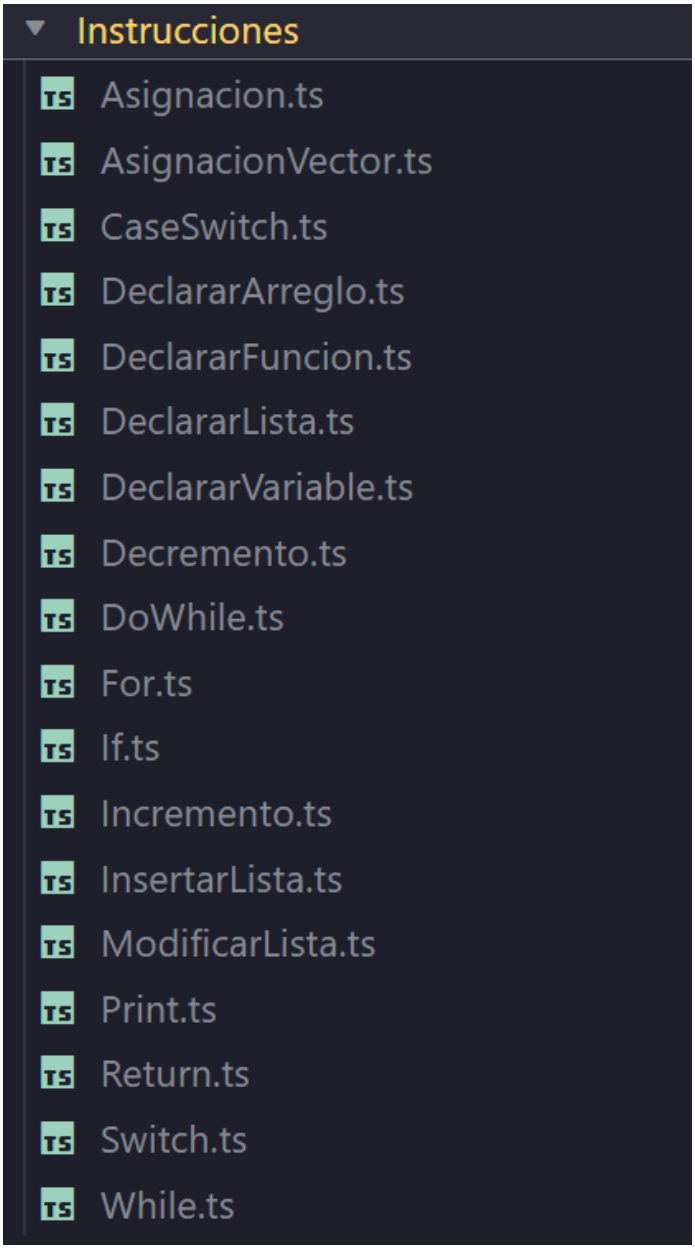




**Expresiones:** este se encarga en encontrar cada una las clases que retornan un valor. Por ello, todas estas clases heredan de la clase Expresión mencionada anteriormente.




**Instrucciones:** aquí tenemos a todas las clases que tienen como tarea realizar una acción. Por ello, heredan de la clase instrucción e implementan el método ejecutar.




```
export class Asignacion extends Instruccion {

    id:     string;
    exp:    Expresion;

    constructor(id: string, exp: Expresion, linea: number, columna: number) {

        super(linea, columna);
        this.id = id;
        this.exp = exp;
    }


    public ejecutar(actual: Ambito, global: Ambito, ast: AST) {

        let variable = actual.getVariable(this.id);
        if(variable === undefined) {
            // * ERROR *
            throw new Error("ERROR => No se ha definido la variable " + this.id);
        }
```