Manual Técnico: Editor de Código en JavaScript

Descripción General

El Editor de Código en JavaScript es una aplicación web desarrollada con JavaScript puro que permite a los usuarios escribir, interpretar y ejecutar código en un lenguaje similar a Java. Este manual técnico proporciona una visión detallada de la arquitectura, componentes y funcionamiento interno del editor, facilitando su mantenimiento, extensión y optimización.

Arquitectura del Sistema

La arquitectura del editor está diseñada siguiendo un enfoque modular, permitiendo una fácil escalabilidad y mantenimiento. Los principales componentes incluyen:

Interfaz de Usuario (UI): Gestiona la interacción con el usuario.

Motor de Interpretación: Procesa y ejecuta el código escrito en el lenguaje similar a Java.

Gestor de Proyectos: Maneja la creación, guardado y carga de proyectos de código.

Componentes Principales

Interfaz de Usuario

La UI está construida con HTML, CSS y JavaScript, proporcionando una experiencia interactiva para el usuario. Los elementos clave incluyen:

Área de Código: Editor de texto para escribir el código.

Botones de Control: Incluyen Ejecutar, Guardar y Nuevo Archivo.

Área de Salida: Muestra la salida del código ejecutado y mensajes de error.

Se uso peggy para el análisis léxico y sintactico del lenguaje


```
programa = _ dcl:Declaracion* _ { return dcl }

Declaracion = dcl:ClassDcl _ { return dcl }
            / dcl:VarDcl _ { return dcl }
            / dcl:FuncDcl _ { return dcl }
            / stmt:Stmt _ { return stmt }

VarDcl = "var" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, undefined, location()) }
       / "int" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.INT, location()) }
       / "float" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.FLOAT, location()) }
       / "char" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.CHAR, location()) }
       / "boolean" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.BOOLEAN, location()) }
       / "string" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.STRING, location()) }
       / "int" _ id:Identificador _ ";" { return new Declaration(id, null, Types.INT, location()) }
       / "float" _ id:Identificador _ ";" { return new Declaration(id, null, Types.FLOAT, location()) }
       / "char" _ id:Identificador _ ";" { return new Declaration(id, null, Types.CHAR, location()) }
       / "boolean" _ id:Identificador _ ";" { return new Declaration(id, null, Types.BOOLEAN, location()) }
       / "string" _ id:Identificador _";" { return new Declaration(id, null, Types.STRING, location()) }

FuncDcl = "void" _ id:Identificador _ "(" _ params:Parametros? _ ")" _ bloque:Bloque { return new FunctionDcl(id, params || [], bloque, location()) }


ClassDcl = "class" _ id:Identificador _ "{" _ dcls:ClassBody* _ "}" { return crearNodo('dclClase', { id, dcls }) }

ClassBody = dcl:VarDcl _ { return dcl }
          / dcl:FuncDcl _ { return dcl }

Parametros = id:Identificador _ params:("," _ ids:Identificador { return ids })* { return [id, ...params] }

Stmt = "System.out.println" _ "(" _ exp:VariasExpresiones _ ")" _ ";" { return new Print(exp, location()) }
    / "switch" _ "(" _ exp:Expresion _ ")" _ "{" _ cases:(_ "case" _ exp1:Expresion _ ":" _ stmt:(Stmt)* _ breakForSwitch:("break" _ ";" {return new Break(location())})* { return new
      stmt:( _ "default" _ ":" _ stmt:Stmt{return stmt})? _ "}" { return new Switch(exp, cases, stmt, location()) }
    / Bloque:Bloque { return Bloque }
    / "if" _ "(" _ cond:Expresion _ ")" _ stmtTrue:Stmt
      stmtFalse:(
        _ "else" _ stmtFalse:Stmt { return stmtFalse }
      )? { return new If(cond, stmtTrue, stmtFalse) }

    / "while" _ "(" _ cond:Expresion _ ")" _ stmt:Stmt { return new While(cond, stmt, location()) }
      / "for" _ "(" _ init:ForInit _ cond:Expresion _ ";" _ inc:Incremento _ ")" _ stmt:Stmt {
      return new For(init, cond, inc, stmt, location())
    }
    / "break" _ ";" { return new Break(location()) }
    / "continue" _ ";" { return new Continue(location()) }
    / "return" _ exp:Expresion? _ ";" { return new ReturnDcl(exp, location()) }
    / exp:Expresion _ ";" { return new ExpressionStatement(exp, location()) }
```

Las declaraciones se manejan de esta manera:

```
VarDcl = "var" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, undefined, location()) }
       / "int" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.INT, location()) }
       / "float" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.FLOAT, location()) }
       / "char" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.CHAR, location()) }
       / "boolean" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.BOOLEAN, location()) }
       / "string" _ id:Identificador _ "=" _ exp:Expresion _ ";" { return new Declaration(id, exp, Types.STRING, location()) }
       / "int" _ id:Identificador _ ";" { return new Declaration(id, null, Types.INT, location()) }
       / "float" _ id:Identificador _ ";" { return new Declaration(id, null, Types.FLOAT, location()) }
       / "char" _ id:Identificador _ ";" { return new Declaration(id, null, Types.CHAR, location()) }
       / "boolean" _ id:Identificador _ ";" { return new Declaration(id, null, Types.BOOLEAN, location()) }
       / "string" _ id:Identificador _";" { return new Declaration(id, null, Types.STRING, location()) }
```

Permite guardar el tipo en un objeto.

```
Stmt = "System.out.println" _ "(" _ exp:VariasExpresiones _ ")" _ ";" { return new Print(exp, location()) }
    / "switch" _ "(" _ exp:Expresion _ ")" _ "{" _ cases:(_ "case" _ exp1:Expresion _ ":" _ stmt:(Stmt)* _ breakForSwitch:("break" _ ";" {return new Break(location())})* { return
      stmt:( _ "default" _ ":" _ stmt:Stmt{return stmt})? _ "}" { return new Switch(exp, cases, stmt, location()) }
    / Bloque:Bloque { return Bloque }
    / "if" _ "(" _ cond:Expresion _ ")" _ stmtTrue:Stmt
      stmtFalse:(
        _ "else" _ stmtFalse:Stmt { return stmtFalse }
      )? { return new If(cond, stmtTrue, stmtFalse) }

    / "while" _ "(" _ cond:Expresion _ ")" _ stmt:Stmt { return new While(cond, stmt, location()) }
      / "for" _ "(" _ init:ForInit _ cond:Expresion _ ";" _ inc:Incremento _ ")" _ stmt:Stmt {
      return new For(init, cond, inc, stmt, location())
    }
    / "break" _ ";" { return new Break(location()) }
    / "continue" _ ";" { return new Continue(location()) }
    / "return" _ exp:Expresion? _ ";" { return new ReturnDcl(exp, location()) }
    / exp:Expresion _ ";" { return new ExpressionStatement(exp, location()) }
```

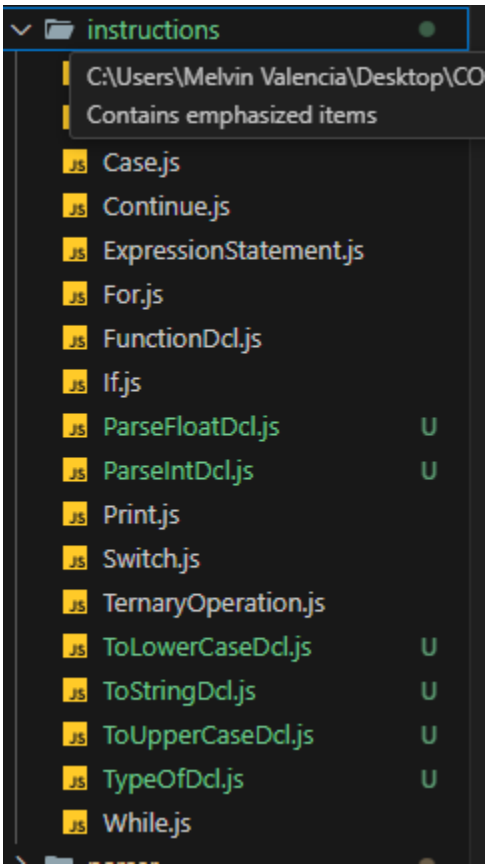
Tenemos los statements como los for, while, etc. Estos están acá porque su sintaxis es diferente a la de una expresión.

```
Expresion = Asignacion

Asignacion = id:Identificador _ "=" _ asgn:Asignacion { return new Assignment(id, asgn, location()) }
           / id:Identificador _ "+" _ "=" _ asgn:Asignacion { return new Assignment(id, new BinaryOperation(new ReferenceVariable(id, location()), asgn, "+", location()), location()) }
           / id:Identificador _ "-" _ "=" _ asgn:Asignacion { return new Assignment(id, new BinaryOperation(new ReferenceVariable(id, location()), asgn, "-", location()), location()) }
           / exp1:AndOr _ "?" _ exp2:AndOr _ ":" _ exp3:AndOr { return new TernaryOperation(exp1, exp2, exp3, location()) }
           / "parseInt" _ "(" _ exp:Expresion _ ")" { return new ParseIntDcl(exp, location()) }
         / "parseFloat" _ "(" _ exp:Expresion _ ")" { return new ParseFloatDcl(exp, location()) }
         / "toString" _ "(" _ exp:Expresion _ ")" { return new ToStringDcl(exp, location()) }
         / "toLowerCase" _ "(" _ exp:Expresion _ ")"  { return new ToLowerCaseDcl(exp, location()) }
         / "toUpperCase" _ "(" _ exp:Expresion _ ")"  { return new ToUpperCaseDcl(exp, location()) }
         / "typeof" _ "(" _ exp:Expresion _ ")"  { return new TypeOfDcl(exp, location()) }
           / AndOr

AndOr = izq:Comparacion expansion:(
  _ op:("&&" / "||") _ der:Comparacion { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (operacionAnterior, operacionActual) => {
      const { tipo, der } = operacionActual;
      return new BinaryOperation(operacionAnterior, der, tipo, location());
    },
    izq
  );
}

Comparacion = izq:MayorMenor expansion:(
  _ op:("<=" / ">=") _ der:MayorMenor { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (operacionAnterior, operacionActual) => {
      const { tipo, der } = operacionActual;
      return new BinaryOperation(operacionAnterior, der, tipo, location());
    },
    izq
  );
```

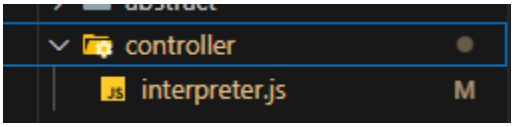
Para manejar precedencias usamos expansiones y el de mayor precedencia lo ponemos hasta abajo.

```
Argumentos = arg:Expresion _ args:("," _ exp:Expresion { return exp })* { return [arg, ...args] }

Numero = "(" _ exp:AndOr _ ")" { return new Agrupation(exp, location()) }
  / "[" _ exp:Expresion _ "]" { return new Agrupation(exp, location()) }
  / [0-9]+( "." [0-9]+ )? { return new Number(parseFloat(text(), 10), location()) }
  / "true" { return new Boolean(true, location()) }
  / "false" { return new Boolean(false, location()) }
  / "\""  txt:StringTxt "\"" { return new String(txt, location()) }
  / "'"  txt:normalChar "'" { return new String(txt, location()) }
  / id:Identificador { return new ReferenceVariable(id, location()) }

StringTxt = [^"]* { return text() }
normalChar = [^'] { return text() }
_ = ([ \t\n\r] / Comentarios)*

Comentarios = "//" (![\n] .)*
            / "/*" (!("*/") .)* "*/"
```

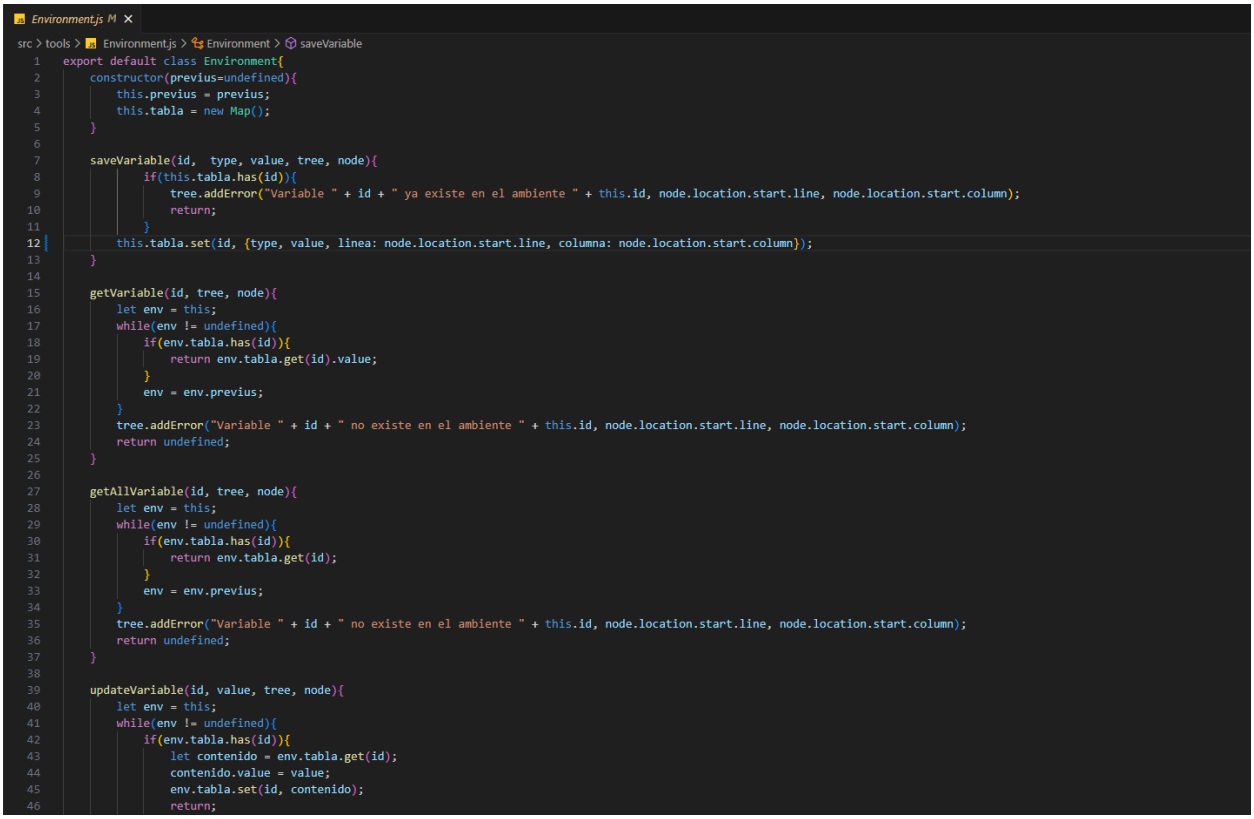
De esta manera organizamos algunas expresiones comunes en nuestro lenguaje.



Nuestras clases instrucciones representan las instrucciones en el lenguaje.



Tenemos la clase interpreter que es donde están nuestros métodos que visitan a nuestras clases.



Nuestra clase environment que es aquí donde esta nuestra tabla de símbolos.

```
import Interpreter from "./controller/Interpreter.js";
import {parse} from "./parser/parser.js";

require.config({ paths: { 'vs': 'https://cdnjs.cloudflare.com/ajax/libs/monaco-editor/0.34.0/min/vs' } });
var errorReport = [];
var tabla = new Map();
require(['vs/editor/editor.main'], () => {
    const editor = monaco.editor.create(document.getElementById('container'), {
        language: 'java',
        theme: 'vs-dark'
    });

    document.getElementById('run-button').addEventListener('click', () => {
        const code = editor.getValue();
        const sentencias = parse(code);
        console.log(sentencias);
        const interprete = new Interpreter();
        sentencias.forEach(sentencia => sentencia.accept(interprete));
        console.log(interprete.environment.tabla);
        console.log("Errores: ");
        console.log(interprete.errors);
        errorReport = interprete.errors;
        tabla = interprete.environment.tabla;
        document.getElementById('output').innerText = interprete.console;
    });

    document.getElementById('open-file-button').addEventListener('click', () => {
        const input = document.createElement('input');
        input.type = 'file';
        input.accept = '.oak';
        input.onchange = e => {
            const file = e.target.files[0];
            const reader = new FileReader();
            reader.onload = event => {
                editor.setValue(event.target.result);
            };
            reader.readAsText(file);
        };
        input.click();
    });

    document.getElementById('save-file-button').addEventListener('click', () => {
        const code = editor.getValue();
        const blob = new Blob([code], { type: 'text/plain' });
        const url = window.URL.createObjectURL(blob);
        const a = document.createElement('a');
        a.href = url;
```

En el script manejamos el botón run y los botones consecuentes del editor.