

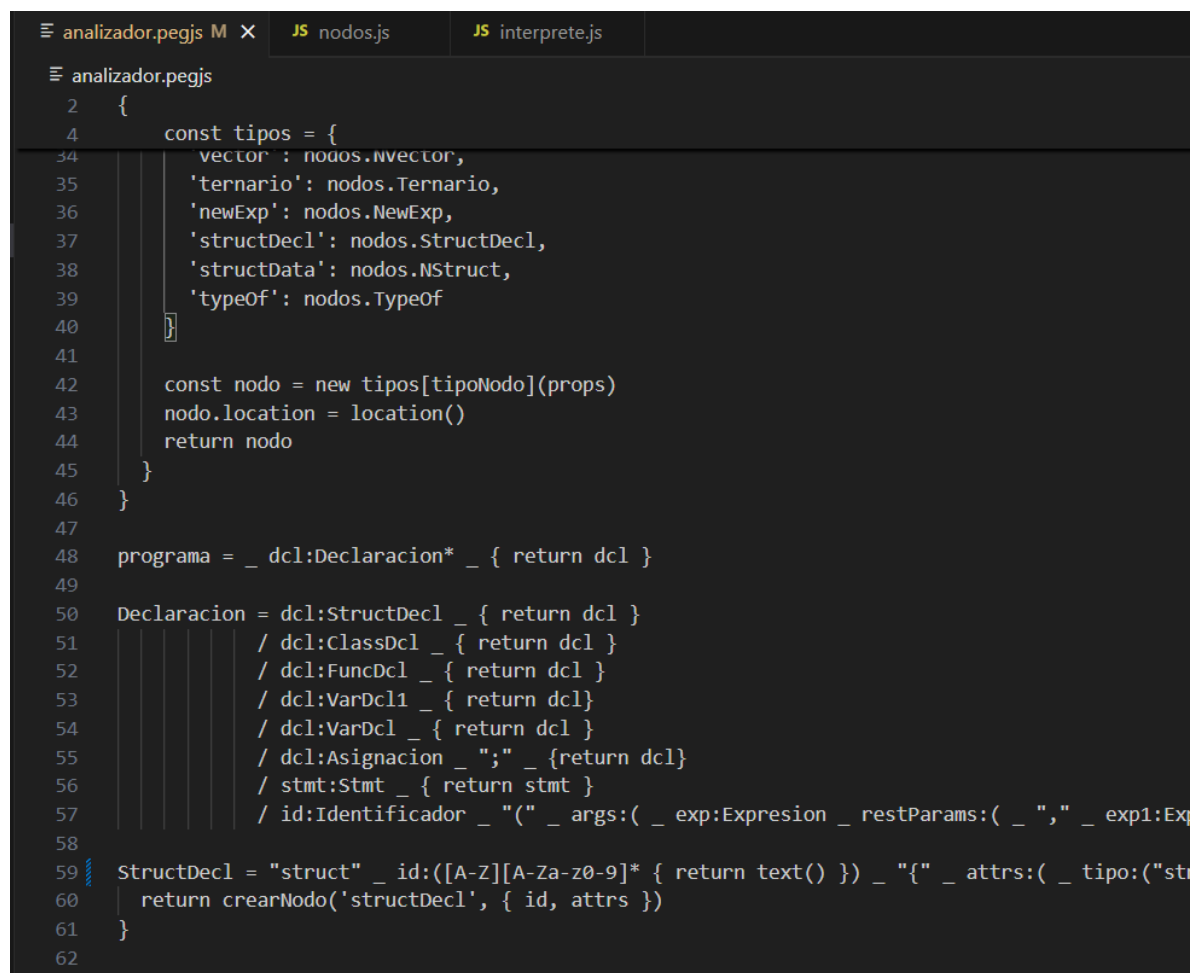
# Manual Técnico

## Características:

De frontend, utiliza meramente html, y vanilla javascript. Como analizador sintáctico, utiliza Peggyjs. Y se utiliza el patrón visitor del lado de semántica.

A continuación, se describe las partes del desarrollo de esta aplicación:

## analizador.pegjs



```
analizador.pegjs M X JS nodos.js JS interprete.js
analizador.pegjs
2  {
4    const tipos = {
34      vector : nodos.Nvector,
35      'ternario': nodos.Ternario,
36      'newExp': nodos.NewExp,
37      'structDecl': nodos.StructDecl,
38      'structData': nodos.Nstruct,
39      'typeof': nodos.TypeOf
40    }
41
42    const nodo = new tipos[tipoNodo](props)
43    nodo.location = location()
44    return nodo
45  }
46 }
47
48 programa = _ dcl:Declaracion* _ { return dcl }
49
50 Declaracion = dcl:StructDecl _ { return dcl }
51             / dcl:ClassDcl _ { return dcl }
52             / dcl:FuncDcl _ { return dcl }
53             / dcl:VarDcl1 _ { return dcl }
54             / dcl:VarDcl _ { return dcl }
55             / dcl:Asignacion _ ";" _ {return dcl}
56             / stmt:Stmt _ { return stmt }
57             / id:Identificador _ "(" _ args:( _ exp:Expresion _ restParams:( _ "," _ exp1:Exp
58
59 StructDecl = "struct" _ id:([A-Z][A-Za-z0-9]*) { return text() } _ "{" _ attrs:( _ tipo:("st
60             return crearNodo('structDecl', { id, attrs })
61 }
62
```

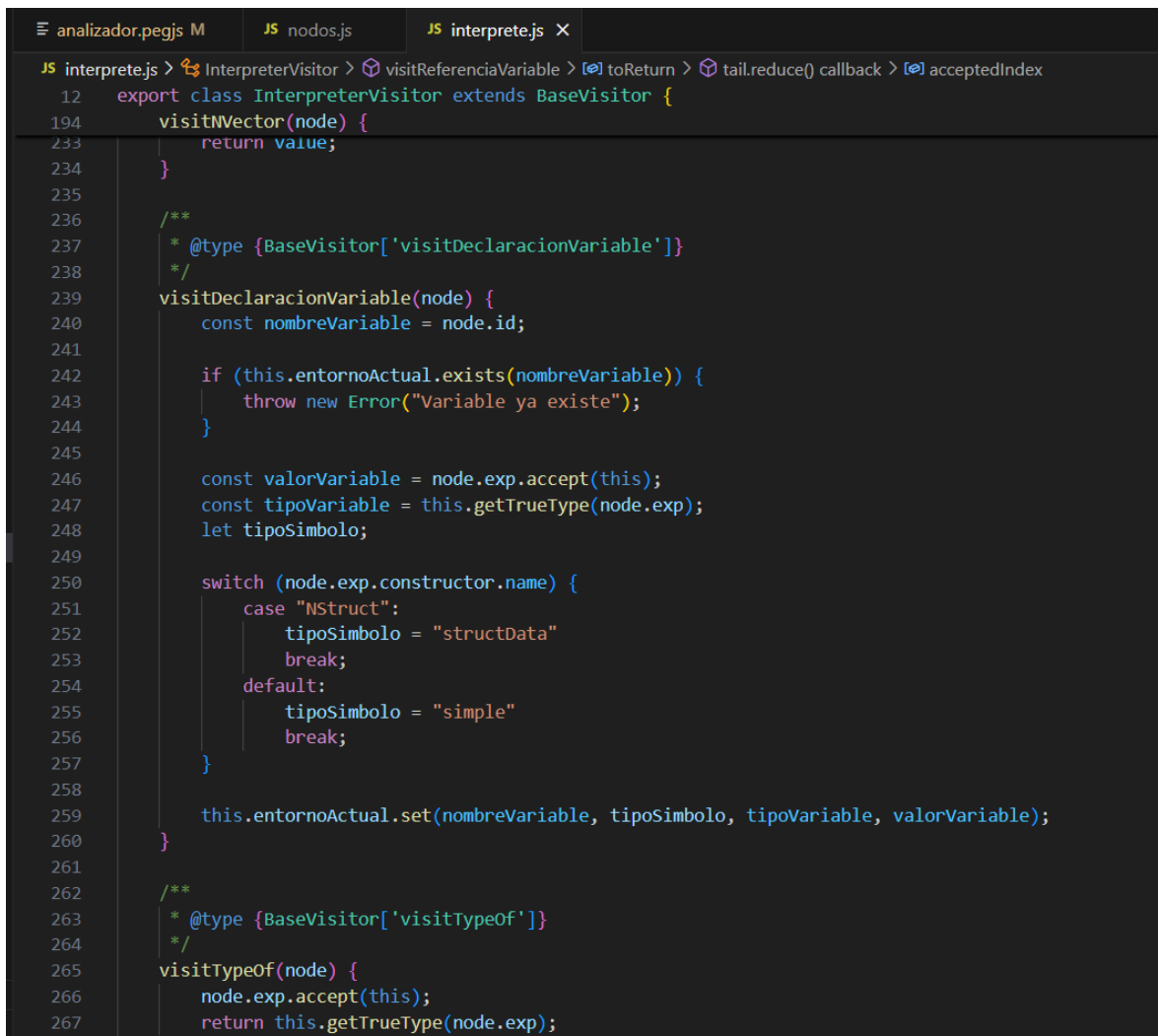
Es donde se define la gramática con lenguaje peggyjs. Esta luego se puede compilar si en la raíz del proyecto se escribe en el CMD “npm run compile”. Éste produce un archivo “analizador.js”, el cual provee una función “parse”, que ya utilizamos luego para analizar un texto.

## nodos.js

```
analizador.pegjs M x JS nodos.js x JS interprete.js
JS nodos.js > [?] default > NVector
126 export class Ternario extends Expression {
166   accept(visitor) {
167     return visitor.visitTernario(this);
168   }
169 }
170
171 export class Agrupacion extends Expression {
172
173   /**
174    * @param {Object} options
175    * @param {Expression} options.exp Expression agrupada
176    */
177   constructor({ exp }) {
178     super();
179
180     /**
181      * Expression agrupada
182      * @type {Expression}
183      */
184     this.exp = exp;
185
186     this.tipo = null;
187   }
188
189   /**
190    * @param {BaseVisitor} visitor
191    */
192   accept(visitor) {
193     return visitor.visitAgrupacion(this);
194   }
195 }
196
```

En el patrón visitor, por cada producción reconocida, se genera un “nodo”. Estos nodos guardan los atributos necesarios para procesarlos (también son los que caracterizan cada nodo individualmente). Notar que cada uno tiene un método `accept`, el cual siempre devuelve un `visit_____`. Todos estos métodos están almacenados en un archivo, el “`interprete.js`”, el cual se describe a continuación.

## interprete.js



```
JS interprete.js > InterpreterVisitor > visitReferenciaVariable > toReturn > tail.reduce() callback > acceptedIndex
12  export class InterpreterVisitor extends BaseVisitor {
194  visitNVector(node) {
233      return value;
234  }
235
236  /**
237   * @type {BaseVisitor['visitDeclaracionVariable']}
238   */
239  visitDeclaracionVariable(node) {
240      const nombreVariable = node.id;
241
242      if (this.entornoActual.exists(nombreVariable)) {
243          throw new Error("Variable ya existe");
244      }
245
246      const valorVariable = node.exp.accept(this);
247      const tipoVariable = this.getTrueType(node.exp);
248      let tipoSimbolo;
249
250      switch (node.exp.constructor.name) {
251          case "NStruct":
252              tipoSimbolo = "structData"
253              break;
254          default:
255              tipoSimbolo = "simple"
256              break;
257      }
258
259      this.entornoActual.set(nombreVariable, tipoSimbolo, tipoVariable, valorVariable);
260  }
261
262  /**
263   * @type {BaseVisitor['visitTypeOf']}
264   */
265  visitTypeOf(node) {
266      node.exp.accept(this);
267      return this.getTrueType(node.exp);
```

Aquí se encuentra cada visit. Al ser un análisis ascendente, cuando se forma el AST, desde el más bajo hacia arriba, se analiza cada nodo. Para hacer eso, se ejecuta su accept, que envía el nodo a su función visit correspondiente, y analiza dependiendo como fue definida, y así sucesivamente.

## entorno.js

```
export class Entorno {  
  /**  
   * @param {Entorno} padre  
   */  
  constructor(padre = undefined) {  
    /**  
     * Identificador de la variable  
     * @type {[val]: { tipoSimbolo: string, tipoVariable: string, dimension: number, valor: any }}  
     */  
    this.valores = {};  
    this.padre = padre;  
  }  
  
  /**  
   * @param {string} nombre  
   * @param {any} tipoSimbolo  
   * @param {any} tipoVariable  
   * @param {number} dimension  
   * @param {any} valor  
   */  
  set(nombre, tipoSimbolo, tipoVariable, valor) {  
    // TODO: si algo ya está definido, lanzar error  
  
    this.valores[nombre] = {  
      tipoSimbolo,  
      tipoVariable,  
      valor  
    };  
  }  
}
```

Este es la clase entorno, la cual básicamente nos provee la tabla de símbolos. Un entorno puede estar dentro de otro entorno. Nótese el método set. Requiere 4 parámetros, los cuales son, el nombre y los datos a guardar que son:

- tipoSimbolo: simple, vector, structDecl; representa el tipo de estructura del valor
- tipoVariable: int, string, etc. Representa el tipo nativo del valor
- valor: es el valor principal a guardar

## my-index.js

```
JS my-index.js X
JS my-index.js > addEventListener("click") callback
1 import { parse } from './analizador.js'
2 import { InterpreterVisitor } from './interprete.js'
3
4 document.getElementById("btn-ejecute").addEventListener("click", () => {
5     const output = document.getElementById("output");
6     const tablinks = document.getElementsByClassName("tablinks");
7     const interprete = new InterpreterVisitor();
8
9     let sentencias;
10
11     for (let i = 0; i < tablinks.length; i++) {
12         if (tablinks[i].classList.contains("active")) {
13             sentencias = parse(document.getElementById(tablinks[i].id.slice(4)).childNodes[1].value)
14             break;
15         }
16     }
17
18     sentencias.forEach(sentencia => sentencia.accept(interprete))
19     console.log({sentencias})
20
21     while (output.firstChild) {
22         output.removeChild(output.firstChild);
23     }
24
25     const newP1 = document.createElement("p");
26     newP1.textContent = " ";
27     output.appendChild(newP1);
28
29     for (let i = 0; i < interprete.salida.split("\n").length - 1; i++) {
30         const newP = document.createElement("p");
31
32         newP.textContent = "> " + interprete.salida.split("\n")[i];
33         output.appendChild(newP);
34     }
35 })
```

Contiene básicamente el onclick del botón ejecutar. Podemos observar que se genera una instancia del intérprete, y que se guarda en “sentencias” un simple parseo del texto. Luego, por cada sentencia, se le provee la instancia de “interprete” para que pueda usarla, y finalmente, en “interprete.salida”, se guarda en forma de string[], cada resultado.