

Manual técnico

Este programa fue desarrollado con Node JS y React para facilitar la comunicación cliente-servidor. Aplicando paradigma de programación orientada a objetos (POO). Se realizó una aplicación web intuitiva para analizar una entrada de texto en un lenguaje intermedio de programación para poder analizarlo a través de una gramática realizada a mano.

Métodos principales:

El programa se dividió en frontend y backend. El trabajo más importante se encuentra en el backend, lo cual tiene como objetivo analizar un instrucciones de un lenguaje de programación y mostrar el resultado.

El backend se dividió en:

- Controladores: aquí está todo el funcionamiento y análisis de lo que se obtiene con la gramática jison
- Endpoints: las conexiones con el frontend

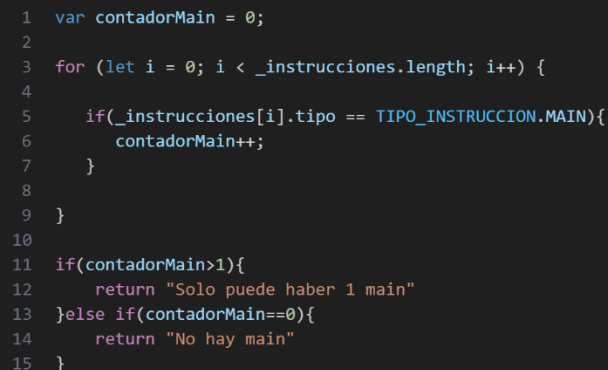
Los controladores se dividen en:

- Ámbito: encierra las estructuras principales, así como clases importantes para el funcionamiento del programa.
- AST: guarda las gráficas generadas
- Enums: engloba todos los tipos de datos que se utilizaran
- Instrucción: aquí se encuentran las instrucciones que tienen una estructura principal
- Operaciones: aquí se encuentran operaciones comunes, así como instrucciones que no necesitan una estructura específica

Los endpoints: mandan a llamar al analizador.js que genera el jison a través de un parser y esto se introduce en la madre de todas las estructuras, la cual es Global.js.

Global

Evalua si viene un main



```
1  var contadorMain = 0;
2
3  for (let i = 0; i < _instrucciones.length; i++) {
4
5      if(_instrucciones[i].tipo == TIPO_INSTRUCCION.MAIN){
6          contadorMain++;
7      }
8
9  }
10
11  if(contadorMain>1){
12      return "Solo puede haber 1 main"
13  }else if(contadorMain==0){
14      return "No hay main"
15  }
```

Posterior a ello evalúa si hay declaraciones de variables, funciones, métodos o asignaciones

```
1  for (let i = 0; i < _instrucciones.length; i++) {
2
3      if(_instrucciones[i].tipo===TIPO_INSTRUCCION.DECLARACION){
4          var mensaje = Declaracion(_instrucciones[i], _ambito)
5          if(mensaje != null){ cadena += mensaje+ "\n" }
6
7      }else if(_instrucciones[i].tipo===TIPO_INSTRUCCION.ASIGNACION){
8          var mensaje = Asignacion(_instrucciones[i], _ambito)
9          if(mensaje != null){ cadena += mensaje+ "\n" }
10
11      }else if (_instrucciones[i].tipo === TIPO_INSTRUCCION.DEC_METODO) {
12          var mensaje = DecMetodo(_instrucciones[i], _ambito)
13          if (mensaje != null) { cadena += mensaje + "\n" }
14      }else if(_instrucciones[i].tipo === TIPO_INSTRUCCION.DEC_FUNCION){
15          var mensaje = DecFuncion(_instrucciones[i], _ambito)
16          if (mensaje != null) { cadena += mensaje + "\n" }
17      }
18  }
19  }
20
```

Ejecuta el método que se encuentra en el Main como primer paso después de a ver evaluado si todo lo que venía era una instrucción válida

```
1  for (let i = 0; i < _instrucciones.length; i++) {
2
3      if(_instrucciones[i].tipo === TIPO_INSTRUCCION.MAIN){
4          var mensaje = Main(_instrucciones[i], _ambito)
5          if(mensaje != null){ cadena += mensaje+ "\n" }
6          break
7      }
8
9  }
10  return cadena
```

Main

Procede a dirigirse al main. Crear un nuevo ámbito y posterior a ello evalúa si el método main llama a un método con o sin parámetros. Busca el método y llama a la clase Bloque para analizar todas las instrucciones del método.

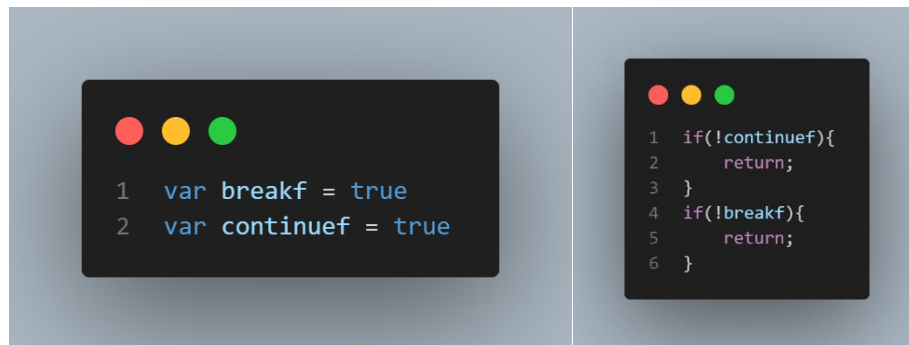
```
1 function Main(instruccion, _ambito) {
2   var metodoEjecutar = _ambito.getMetodo(instruccion.nombre)
3
4   var cadena = ""
5   if (metodoEjecutar != null) {
6     var nuevoAmbito = new Ambito(_ambito, "Main")
7     //console.log(metodoEjecutar)
8     if (metodoEjecutar.lista_parametro != null) {
9       if (_instruccion.lista_valores != null && metodoEjecutar.lista_parametro.length == _instruccion.lista_valores.length) {
10        //console.log("entro")
11        var error = false;
12        for (let i = 0; i < metodoEjecutar.lista_parametro.length; i++) {
13          var declaracionAsignacion = Instruccion.nuevaDeclaracion(metodoEjecutar.lista_parametro[i].id, _instruccion.lista_valores[i], metodoEjecutar.lista_parametro[i].tipo_datos, _instruccion.linea, _instruccion.columna)
14          //console.log(declaracionAsignacion)
15          var mensaje = DecParametro(declaracionAsignacion, nuevoAmbito)
16
17          if (mensaje != null) {
18            error = true
19            cadena += mensaje + "\n"
20          }
21        }
22        if (error) {
23          return cadena
24        }
25        var ejec = Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
26        var mensaje = ejec.cadena
27
28        return mensaje
29      }
30    } else {
31      return 'error: faltan valores para el metodo ${_instruccion.nombre}... Linea: ${_instruccion.linea} Columna: ${_instruccion.columna}'
32    }
33  } else {
34    var ejec = Bloque(metodoEjecutar.instrucciones, nuevoAmbito)
35    var mensaje = ejec.cadena
36    return mensaje
37  }
38 }
39
40 return 'error: el metodo ${_instruccion.nombre} no existe... Linea: ${_instruccion.linea} columna: ${_instruccion.columna}'
41 }
42 }
```

Bloque

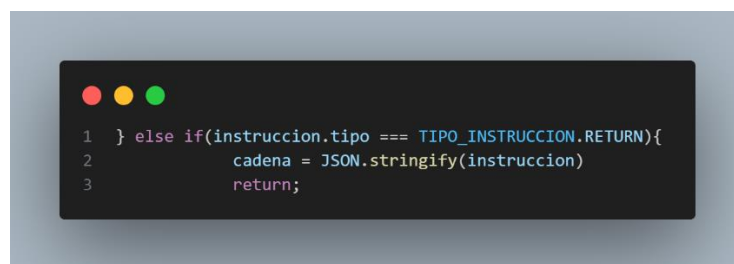
En el método bloque analiza todas las instrucciones que pueden venir. Ya sea un print, declaración, asignación, if, ifelse, ifelseif, ++ o --, lista, for, switch, while, dowhile, ejecutar método y las instrucciones de escape como lo son: Return, Break y continue. A continuación un ejemplo de cómo evalúa cada instrucción y llama a su respectiva clase cuando encuentre una coincidencia.

```
1 if(instruccion.tipo===TIPO_INSTRUCCION.PRINT){
2   cadena += Print(instruccion,_ambito) + "\n"
3
4 }else if (instruccion.tipo === TIPO_INSTRUCCION.DECLARACION) {
5   var mensaje = Declaracion(instruccion, _ambito)
6   if (mensaje != null) { cadena += mensaje }
7
8 }else if (instruccion.tipo === TIPO_INSTRUCCION.ASIGNACION) {
9   var mensaje = Asignacion(instruccion, _ambito)
10  if (mensaje != null) { cadena += mensaje }
11
12 }
```

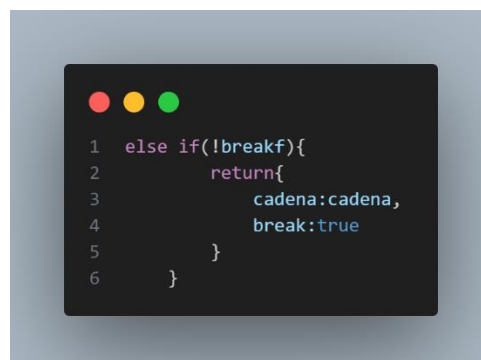
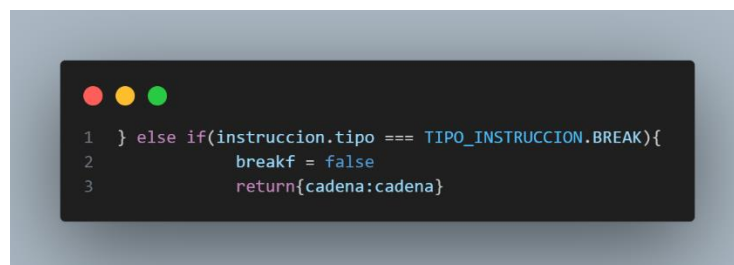
Las instrucciones de escape: break, continue y return se manejan de la siguiente manera. Se declaran afuera de la clase unas variables globales breakf (significa no esta break), continuef (no esta continue). Para ser evaluadas en cada instrucción



Si encuentra un return, devuelve únicamente la instrucción que se encuentra en el return. Debido a que en funciones se devuelve la operación que esta realizando el return. Para esto se convierte a un string toda la instrucción y se retorna como cadena para ser evaluada por individual y que ya no siga el ciclo con las demás instrucciones.



Si encuentra un break, lo que debe hacer es salirse de la función o ciclo. Para esto si se encuentra retorna la cadena de una vez y se devuelve una cadena con un parámetro break= true



Si encuentra un continue, lo que debe de hacer es omitir todo lo que esta debajo y seguir con la siguiente iteración. Sin embargo, se debe evaluar que esté dentro de un ciclo. Para esto se realiza un ciclo evaluando los ámbitos anteriores

```
1 else if(instruccion.tipo === TIPO_INSTRUCCION.CONTINUE){
2     var nambito = _ambito
3     while(nambito.nombre !== "Main"){
4         if(nambito.nombre === "INSTRUCCIONFOR" || nambito.nombre === "WHILE" || nambito.nombre === "DOWHILE"){
5             break;
6         }
7         nambito = nambito.anterior
8     }
9     if(nambito.nombre !== "Main"){
10        continuef = false
11    }
12    else{
13        cadena = `Error: La instruccion ${instruccion.tipo} debe estar dentro de un ciclo... Línea: ${instruccion.linea} Columna: ${instruccion.columna}`
14    }
15 }
```

Lo que se realizó en el ciclo es ir evaluando las instrucciones anteriores. Si encuentra más de algún ciclo definido se detiene y lo devuelve. De no ser así, llegara hasta main y significará que no habrá un ciclo. Al igual que con el break, se devuelve un parámetro extra que indica que encontró un continue=true

```
1 else if(!continuef){
2     return{
3         cadena:cadena,
4         continue: true
5     }
6 }
```

Estas son las principales funciones, y así va funcionando el proyecto en general. Al igual que con las operaciones, estas se mandan a llamar dentro de las instrucciones que llama bloque y devuelve la cadena debida.

Instrucción

Este es un archivo muy importante debido a que se conecta directamente con la ejecución del jison. Sus funciones se van llamando en el jison y se va creando y retornando los valores, instanciando un tipo, valor, línea y columna generalmente para poder ser analizado en los archivos global, main, bloque, operación entre otros.

```

1  },nuevoMain: function (_nombre, _lista_valores, _linea, _columna) {
2      return{
3          tipo: TIPO_INSTRUCCION.MAIN,
4          nombre: _nombre,
5          lista_valores: _lista_valores,
6          linea: _linea,
7          columna: _columna
8      }
9  }
10 },nuevoMetodo: function (_nombre, _lista_parametros, _instrucciones, _linea, _columna) {
11     return {
12         tipo: TIPO_INSTRUCCION.DEC_METODO,
13         nombre: _nombre,
14         lista_parametros: _lista_parametros,
15         instrucciones: _instrucciones,
16         linea: _linea,
17         columna: _columna
18     }

```

Operación

Funciona como la clase bloque que analiza instrucciones. Lo único que esta función solo analiza operaciones. Puede analizar si solo viene datos, aritméticas, lógicas, relacionales, ternarias, casteo, función y operaciones nativas. Estas se llaman dentro de las instrucciones principales.

```

1  function Operacion(_expresion, _ambito){
2      if(_expresion.tipo === TIPO_VALOR.DECIMAL || _expresion.tipo === TIPO_VALOR.BOOL || _expresion.tipo === TIPO_VALOR.ENTERO ||
3         _expresion.tipo === TIPO_VALOR.CADENA || _expresion.tipo === TIPO_VALOR.IDENTIFICADOR || _expresion.tipo === TIPO_VALOR.CHAR){
4          return ValorExpresion(_expresion, _ambito);
5      }
6      // OPERACIONES ARITMETICAS
7      else if(_expresion.tipo === TIPO_OPERACION.SUMA || _expresion.tipo === TIPO_OPERACION.RESTA || _expresion.tipo === TIPO_OPERACION.MULTIPLICACION ||
8         _expresion.tipo === TIPO_OPERACION.DIVISION || _expresion.tipo === TIPO_OPERACION.POTENCIA || _expresion.tipo === TIPO_OPERACION.MODULO ||
9         _expresion.tipo === TIPO_OPERACION.UNARIA){
10         return Aritmetica(_expresion, _ambito);
11     }
12     // OPERACIONES RELACIONALES
13     else if(_expresion.tipo === TIPO_OPERACION.IGUAL || _expresion.tipo === TIPO_OPERACION.DIFERENTE ||
14         _expresion.tipo === TIPO_OPERACION.MENOR || _expresion.tipo === TIPO_OPERACION.MAYOR || _expresion.tipo === TIPO_OPERACION.MAYORIGUAL ||
15         _expresion.tipo === TIPO_OPERACION.MENORIGUAL){
16         return Relacional(_expresion, _ambito);
17     }
18     // OPERACIONES LOGICAS
19     else if(_expresion.tipo === TIPO_OPERACION.AND || _expresion.tipo === TIPO_OPERACION.OR || _expresion.tipo === TIPO_OPERACION.NOT){
20         return Logica(_expresion, _ambito);
21     }
22     // OPERADOR TERNARIO
23     else if(_expresion.tipo === TIPO_OPERACION.TERNARIO){
24         return Ternario(_expresion, _ambito);
25     }
26     // CASTEO
27     else if(_expresion.tipo === TIPO_INSTRUCCION.CASTEO){
28         return Casteo(_expresion, _ambito);
29     }
30     // EJECUTAR FUNCION
31     else if(_expresion.tipo === TIPO_INSTRUCCION.EJE_FUNCION){
32         return FuncionOp(_expresion, _ambito);
33     }
34     // OPERACIONES NATIVAS
35     else if(_expresion.tipo === TIPO_OPERACION.TOLOWER || _expresion.tipo === TIPO_OPERACION.Toupper || _expresion.tipo === TIPO_OPERACION.LENGTH ||
36         _expresion.tipo === TIPO_OPERACION.TRUNCATE || _expresion.tipo === TIPO_OPERACION.ROUND || _expresion.tipo === TIPO_OPERACION.TYPEOF ||
37         _expresion.tipo === TIPO_OPERACION.TOSTRING || _expresion.tipo === TIPO_OPERACION.TOCHARARRAY){
38         return Nativa(_expresion, _ambito);
39     }
40 }

```