



SOBRE ESTE MANUAL

Manual sobre como funciona la pagina creadada para simular el lenguaje de oakland. Brevemente resumido

Jonathan Alexander Sanchez Barrios

Tabla de contenido

Nivel de carpetas:	2
Css:	2
Pruebas:	2
Src:	2
Absstract:	3
Enviroment:	5
Peggy	5
Tables	5
Structures:	6
Invocable:	7
Funciones Embebidas:	8
Gramática	10
Observaciones:	15

Nivel de carpetas:

Css:

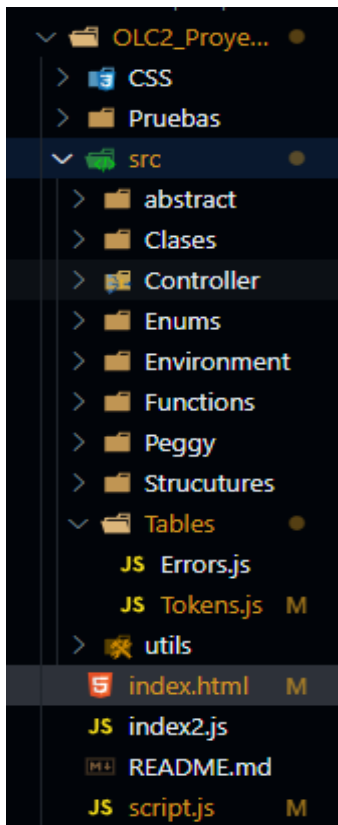
Estilo de la pagina web

Pruebas:

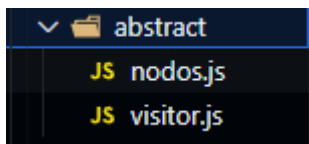
Archivos de entrada de prueba.

Src:

Se encuentra las carpetas y código fuente (backend) del interprete.



Absstract:



En el archivo nodos.js se encuentra la clase expresión de la cual extenderán el diversas de las clases que se utilizaran las cuales heredaran método accpet el cual es donde se realizara la función de interpretar cada nodo.

Expresión:

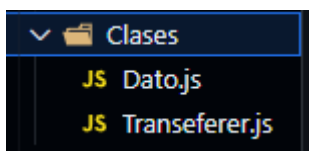
```
Codiumate: Options | Test this class
export class Expression {

    /** ...
    constructor() {

    /**
     * Ubicacion del nodo en el codigo fuente
     * @type {Location|null}
     */
    this.location = null;

    /**
     * @param {BaseVisitor} visitor
     */
    Codiumate: Options | Test this method
    accept(visitor) {
        return visitor.visitExpresion(this);
    }
}
```

Clases:



Dato: La clase Dato nos servirá para el manejo de todo tipo de datos como primitivos

funciones, la cual tiene como atributos:

```
1  export class Dato {
2    /**
3     *
4     * @param {string} type
5     * @param {any} value
6     * @param {location} location
7     */
8    constructor(type, value, location) {
9      this.type = type;
10     this.value = value;
11     this.location = location;
12   }
13 }
```

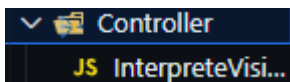
Transferer:

Son las sentencias de transferencias las cuales son extends de errors y se obtienen mediante un try catch:

```
1  export class BreakException extends Error{
2    constructor(){
3      super('BreakException');
4    }
5  }
6
7  export class ContinueException extends Error{
8    constructor(){
9      super('ContinueException');
10   }
11 }
12
13 export class ReturnException extends Error{
14   constructor(value){
15     super('ReturnException');
16     this.value = value;
17   }
18 }
19
```

Controler:

Carpeta donde se trabajara nuestro Interprete Visitor el cual es donde haremos los accept de cada nodo.



Controller

JS InterpreteVisi...

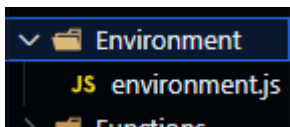
```

1  export class InterpreteVisitor extends BaseVisitor {
2
3      constructor() {
4          super();
5          //su entorno//(undefined )ya que no debe existir un anterior al global
6          this.environment = new Environment(undefined);
7          this.outPut = '';
8          ErrorsArr.splice(0,ErrorsArr.length)
9          //funciones embebidas:
10         Object.entries(embebedas).forEach(([nombre,funcion]) => {
11             this.environment.assignVariable(nombre,funcion,'function')
12         });
13         /**
14          * @type {Expresion|null}
15          */
16         this.preVContinue = null; // manejo de nivel anterior como ciclos, switch, funciones
17     }

```

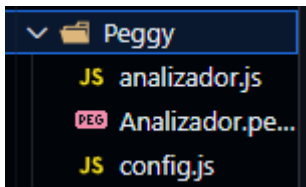
Enviroment:

Clase utilizada como entorno en el cual se agregan variables, modifican y se obtiene mediante un diccionario, recibiendo como parámetros una clase Dato.js



Peggy

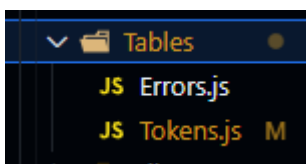
Sera nuestra carpeta con nuestra gramática y el archivo config para generar nuestro Analizador.js de la manera correcta y necesaria para esto:



Para crear el analizador.js es requerido el siguiente comando:
 npx peggy -c .\config.js

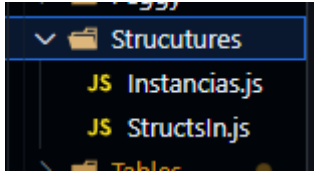
Tables

Manejamos tablas de símbolos y errores.



Structures:

Se harán uso de 2 clases las cuales tienen como objetivo el manejo de structs e instancias de la misma:



```
1  invocar(interprete,args){
2      const struct = new Instances(this);
3      Object.entries(this.properties).forEach(
4          ([key,value]) => {
5              struct.set(key,value);
6          }
7      );
8      args.forEach(
9          (arg) => {
10             const valueAsg = arg.asgn.accept(interprete);
11             struct.set(arg.id,valueAsg);
12          }
13      );
14  }
```

Se crean las propiedades y al invocar si tiene argumentos se setean los valores en la clase Instancia.

Seteando valores de la Instancia y corroborando mismo tipo.

```
1  set(name,value){
2    console.log(`Setting: ${name} → ${value.value}`)
3    const prop = this.properties[name]
4    if (prop) {
5      if (prop.type === value.type) {
6        console.log('same prop type and value type')
7        console.log(`${prop.type} = ${value.type}`)
8        this.properties[name] = value;
9        this.properties[name] = value;
10       return;
11     }else{
12       let errStr = `${name} ${prop.type} ≠ ${value.type}`
13       let line = value.location.start.line
14       let column = value.location.start.column
15       let type = 'Semantico'
16       let err = new ErrorClass(ErrorCounts,errStr,line,column,type)
17       ErrorsArr.push(err)
18     }
19   }else{
20     this.properties[name] = value;
21   }
22 }
```

Invocable:

Aridad es la cantidad de parámetros requeridos para poder invocar una clase invocable.


```
1  export class Invocable {
2
3      aridad(){
4          throw new Error('Aridad aint implemented')
5      }
6
7
8      /**
9       *
10      * @param interprete {InterpreteVisitor}
11      * @param args {any[]}
12      */
13
14      invocar(interprete,args){
15          throw new Error('Invocar aint implemented')
16      }
17
18  }
```

Funciones Embebidas:

Son la funciones nativas del lenguaje las cuales guardamos en el entorno.

```

1 class NativeFunction extends Invocable{
2   constructor(aridad,func) {
3     super()
4     this.aridad = aridad
5     this.invocar = func
6   }
7 }
8 export const embebedas = {
9   'time': new NativeFunction(() => 0, () => {
10     console.log('HolaMundo TIMEEEEEEEEE')
11     return new Dato('string', '04/09/2024',null)
12   }),
13   'parseInt': new NativeFunction(() => 1,(a,b) => {
14     const value = Math.floor(parseFloat(b[0].value))
15     if (isNaN(value)) {
16       const errStr = 'Cant to parseInt -> ${b[0].value} `
17       const line = b[0].location.start.line
18       const column = b[0].location.start.column;
19       const errToSave = new ErrorClass(ErrorCounts,errStr,line,column,"semantic");
20       ErrorsArr.push(errToSave);
21       return new Dato('null',null,b[0].location)
22     }
23     return new Dato('int',value,null)
24   }),
25   'parseFloat': new NativeFunction(() => 1,(a,b) => {
26     console.log(parseFloat(b[0].value))
27     const value = parseFloat(b[0].value)
28     if (isNaN(value)) {
29       const errStr = 'Cant to parseFloat ${b[0].value} `
30       const line = b[0].location.start.line
31       const column = b[0].location.start.column;
32       const errToSave = new ErrorClass(ErrorCounts,errStr,line,column,"semantic");
33       ErrorsArr.push(errToSave);
34       return new Dato('null',null,b[0].location)
35     }
36     return new Dato('float',parseFloat(b[0].value),b[0].location)
37   }),
38   'toString': new NativeFunction(() => 1,(a,b) => {
39     return new Dato('string',b[0].value.toString(),b[0].location)
40   }),
41   //toLowerCase
42   'toLowerCase': new NativeFunction(() => 1,(a,b) => {
43     if (b[0].type === 'string') {
44       return new Dato('string',b[0].value.toLowerCase(),b[0].location)
45     }
46     const errStr = 'Cant to lower case ${b[0].value} `
47     const line = b[0].location.start.line
48     const column = b[0].location.start.column;
49     const errToSave = new ErrorClass(ErrorCounts,errStr,line,column,"semantic");
50     ErrorsArr.push(errToSave);
51     return new Dato('null',null,b[0].location)
52   }),
53   //toUpperCase
54   'toUpperCase': new NativeFunction(() => 1,(a,b) => {
55     //validate Error:
56     if (b[0].type === 'string') {
57       return new Dato('string',b[0].value.toUpperCase(),b[0].location)
58     }
59     const errStr = 'Cant to touppercase -> ${b[0].value} `
60     const line = b[0].location.start.line
61     const column = b[0].location.start.column;
62     const errToSave = new ErrorClass(ErrorCounts,errStr,line,column,"semantic");
63     ErrorsArr.push(errToSave);
64     return new Dato('null',null,b[0].location)
65   }),
66   //typeof
67   'typeof': new NativeFunction(() => 1,(a,b) => {
68     console.log('b: ', b[0].type)
69     const value = b[0].value;
70     console.log(b[0].type)
71     return new Dato('string',b[0].type,b[0].location)
72   }),
73   //object.entries
74   'entries': new NativeFunction(() => 1,(a,b) => {
75     const value = b[0].value;
76   })
77 }
78 )
79
80 };

```

Gramática

Se crea una función que cree los nodos para ser utilizados en el Interpretador Visitor. Para poder usarlo se debe importar desde la carpeta config.js

```
{
  const crearNodo = (tipoNodo, props) =>{
    const tipos = {
      'numero': nodos.Numero,
      'primitive': nodos.Primitive,
      'agrupacion': nodos.Agrupacion,
      'binaria': nodos.OperacionBinaria,
      'logica': nodos.OpLogica,
      'unaria': nodos.OperacionUnaria,
      'declaracionVariable': nodos.DeclaracionVariable,
      'referenciaVariable': nodos.ReferenciaVariable,
      'print': nodos.Print,
      'sout': nodos.Sout,
      'expresionStmt': nodos.ExpresionStmt,
      'asignacion': nodos.Asignacion,
      'bloque': nodos.Bloque,
      'if': nodos.If,
      'while': nodos.While,
      'for': nodos.For,
      'switch': nodos.Switch,
      'break': nodos.Break,
      'continue': nodos.Continue,
      'return': nodos.Return,
      'llamada': nodos.Llamada,
      'DclFunc': nodos.DclFunc,
      'ternario': nodos.tern,
      'DclStruct': nodos.DclStruct,
      'instClass': nodos.instClass,
      'getStruct': nodos.getStruct,
      'setStruct': nodos.setStruct,
      'entries': nodos.entries,
    }

    const nodo = new tipos[tipoNodo](props)
    nodo.location = location()
    return nodo
  }
}
```

En operaciones binarias:

Se hace uso de un reduce para juntar la gramática por la izquierda y así obtener mejores con el lenguaje.

```
1 Multiplicacion = izq:Unaria expansion:(
2   _ op:("*" / "/" / "%") _ der:Unaria { return { tipo: op, der } }
3 )* {
4   return expansion.reduce(
5     (operacionAnterior, operacionActual) => {
6       const { tipo, der } = operacionActual
7       return crearNodo('binaria', { op:tipo, izq: operacionAnterior, der })
8     },
9     izq
10  )
11 }
```

Empezamos la gramática:

```
1 programa = _ dcl:Declaracion* _ { return dcl }
2 Declaracion = dcl:DclStruct _ { return dcl }
3 // dcl:VarDcl _ { return dcl }
4 // dcl:DclFunc _ { return dcl }
5 // stmt:Stmt _ { return stmt }
6
7 //VarDcl = tipo:TypesValues _ id:Identificador _ "=" _ exp:Expresion _ ";" { return crearNodo('declaracionVariable', { id:id, exp:exp,typeD:tipo }) }
8 VarDcl = tipo:TypesValues _ id:Identificador _ asigna:("=" _ exp:Expresion{return exp})? _ ";" { return crearNodo('declaracionVariable', { id:id, exp:asigna,typeD:tipo }) }
9
10 DclFunc = tipo:TypesValues _ id:Identificador "(" _ params:Parametros? _ ")" _ bloque:Bloque {
11   console.log('DclFunc')
12   return crearNodo('DclFunc',{ type:tipo,id,params:params || [],bloque})
13 }
14 DclStruct = "struct" _ id:Identificador _ "{" _ bodyC:DclsStruct* _ "}" _ ptcoma:(";")? {
15   console.log('DeclClass : ', id , ' body: ', bodyC)
16   return crearNodo('DclStruct',{ id,properties:bodyC })
17 }
18
```

```

1  DclsStruct = dcl:VarDcl _ {return dcl}
2    /dcl:DclFunc _ {return dcl}
3
4  //Parametros = id:Identificador _ params:("," ids:Identificador{return ids})* { return [id, ...params] }
5  Parametros = tipo:TypesValues _ id:Identificador _ params:("," _ tipo1:TypesValues _ ids:Identificador{
6    return { id:ids,typeD:tipo1 }
7  })* {
8    let abc = { id:id,typeD:tipo };
9    return [abc, ...params]
10  }
11
12
13  TypesValues = "int" { return "int"}
14    /"float"{ return "float"}
15    /"string" {return "string"}
16    /"char" { return "char"}
17    /"bool"{ return "bool"}
18    /"var" {return "var"}
19    /"void"{ return "void"}
20    /id:Identificador{return id}
21
22
23
24  Stmt = "print(" _ exp:Expresion _ ")" _ ";" { return crearNodo('print', { exp }) }

```

```

1  FortBeginning = dcl:VarDcl {return dcl}
2    / exp:Expresion _ ";" {return exp}
3    /";" {return null }
4
5  Bloque = "{" _ dcls:Declaracion* _ "}" { return crearNodo('bloque', { dcls }) }
6
7  MultipleCases = "case" _ exp:Expresion _ ":" _ stmt:Declaracion* _ cases:( _ "case" _ exp1:Expresion _ ":" _ stmt1:Declaracion*
8    { return [exp:exp1,stmt:stmt1] })* _ { return [ {exp,stmt:stmt}, ...cases] }
9  DefaultExp = "default" _ ":" _ stmt:Declaracion*{
10    return stmt
11  }
12

```

```

1  Identificador = !ReservedWords [a-zA-Z][a-zA-Z0-9]* { return text() }
2
3  Expresion = Asignation
4
5  PrintComa = exp:Expresion _ params:("," _ exp1:Expresion
6    { return exp1 })* { return [exp, ...params] }
7
8
9
10 Asignation = asgndVlalue:Called _ type:asgnTypes _ asgn:Asignation
11 {
12   console.log('asignation: ', type)
13   console.log({asgndVlalue});
14   console.log({asgn})
15   if (asgndVlalue instanceof nodos.ReferenciaVariable) {
16     console.log('Referenciaaaa De Variableeeeeeeeeeeee');
17     return crearNodo('asignacion', { id:asgndVlalue.id, asgn:asgn,op:type })
18   }
19   if (!(asgndVlalue instanceof nodos.getStruct)) {
20     throw new Error('Solo se pueden asignar valores a propiedades de objetos')
21   }
22   console.log('asgnVlalue')
23   console.log({asgndVlalue});
24   console.log('tipoSetStruccccccccccc')
25   console.log({asgn})
26   return crearNodo('setStruct', { id:asgndVlalue.id, propertie:asgndVlalue, value:asgn })
27 }

```

```

1  Asignation = asgndVlalue:Called _ type:assignTypes _ asgn:Asignation
2  {
3    console.log('asignation: ', type)
4    console.log({asgndVlalue});
5    console.log({asgn})
6    if (asgndVlalue instanceof nodos.ReferenciaVariable) {
7      console.log('Referencias De Variables');
8      return crearNodo('asignacion', { id:asgndVlalue.id, asgn:asgn,op:type })
9    }
10   if (!(asgndVlalue instanceof nodos.getStruct)) {
11     throw new Error('Solo se pueden asignar valores a propiedades de objetos')
12   }
13   console.log('asgnVlalue')
14   console.log({asgndVlalue});
15   console.log('tipoSetStruc')
16   console.log({asgn})
17   return crearNodo('setStruct', { id:asgndVlalue.id, propertie:asgndVlalue, value:asgn })
18 }
19 /Ternario
20 /Logical
21
22 assignTypes = ("="|"+="|"-="|"*="|"/="|<="|>="|<|">|"<="|>="|<|">") {return text()}
23
24 Ternario = cond:Logical _ "?" _ stmtTrue:Expresion _ ":" _ stmtFalse:Expresion _ {
25   // console.log('Ternario', cond, stmtTrue, stmtFalse);
26   return crearNodo('ternario', { cond, stmtTrue, stmtFalse })
27 }
28

```

Por precendencia la gramática “números” es donde van nuestros tipos de datos que recibiremos al igual que las expresiones entre paréntesis para poder tenerle mas prioridad.

```

1  llamada = callee:Numero _ params:("(" args:Argumentos? ")" { return args })* {
2      return params.reduce(
3          (callee, args) => {
4              return crearNodo('llamada', { callee, args: args || [] })
5          },
6          callee
7      )
8  }
9
10 Called = callee:Numero operaciones:(
11     "(" _ args:Argumentos? _ ")" {
12         return { args, tipo: 'llamada' }
13     } / "." _ id:Identificador _ {
14         return { args: id, tipo: 'getStruct' }
15     }
16 )*
17 {
18     console.log('Called', callee, operaciones)
19     const call = operaciones.reduce(
20         (callee, args) => {
21             console.log(`Calle ${callee} , args ${args}`)
22             const { tipo, id, args: argumentos } = args
23             if (tipo === 'llamada') {
24                 return crearNodo('llamada', { callee, args: argumentos || [] })
25             } else if (tipo === 'getStruct') {
26                 return crearNodo('getStruct', { id: callee, propertie: args })
27             }
28         },
29         callee
30     )
31     console.log('llamada', { call }, { text: text() });
32     return call
33 }
34
35
36
37
38 Argumentos = arg:Expresion _ args:("(" _ exp:Expresion {
39     return exp })* { return [arg, ...args] }
40
41 Numero = [0-9]+( "." [0-9]+ )+ { return crearNodo('primitive', { typeD: 'float', value: Number(text(), 0) }) }
42 / [0-9]+ { return crearNodo('primitive', { typeD: 'int', value: Number(text(), 0) }) }
43 / "'" [^"]* '"' { return crearNodo('primitive', { typeD: 'string', value: text().slice(1, -1) }) }
44 / '"' [^']* '"' { return crearNodo('primitive', { typeD: 'char', value: text().slice(1, -1) }) }
45 / "true" {
46     console.log('true peggy')
47     return crearNodo('primitive', { typeD: 'bool', value: true })
48 } / "false" { return crearNodo('primitive', { typeD: 'bool', value: false }) }
49 / "null" { return crearNodo('primitive', { typeD: 'null', value: null }) }
50 / "object.keys(" _ exp:Expresion _ )" {
51     return crearNodo('entries', { value: exp })
52 }
53
54 / "(" _ exp:Expresion _ ")" { return crearNodo('agrupacion', { exp }) }
55 / id:Identificador _ "{" _ argsI:Argumentos? _ "}" _ ptcoma:(",")? {
56     console.log('Instancia ', id, argsI)
57     return crearNodo('instClass', { id, args: argsI })
58 }
59 / id:Identificador { return crearNodo('referenciaVariable', { id }) }
60
61
62 _ = ([ \t\n\r] / Comments)*
63

```

Observaciones:

Entre mas abajo se encuentre nuestra regla gramática mayor prioridad tendrá pero se debe tomar en cuenta que dentro de las reglas sus subreglas las de arriba tendrán mayor precedencia:

Ejemplo

Como pueden observar las palabras reservadas “true, false, null” los definimos antes de la regex id para que no la tome como id .

```
1 / "true" {
2   console.log('true peggy' )
3   return crearNodo('primitive', { typeD:'bool', value:true }) }
4 / "false" {return crearNodo('primitive', { typeD:'bool', value:false }) }
5 / "null" {return crearNodo('primitive', { typeD:'null', value:null }) }
6 /object.keys("_ exp:Expresion _") {
7   return crearNodo('entries', {value:exp })
8 }
9 / "(" _ exp:Expresion _ ")" { return crearNodo('agrupacion', { exp }) }
10 / id:Identificador _ "{" _ argsI:Argumentos? _ "}" _ ptcoma(":")? { return crearNodo('instClass',{ id,args:argsI }) }
11 }
12 / id:Identificador { return crearNodo('referenciaVariable', { id }) }
13
```