Natalia Mariel Calderón Echeverría
202200007
Organización de Lenguajes y Compiladores 2
Segundo semestre 2024

Manual de Tecnico

Requerimientos básicos:

- **Lenguaje programado:** Javascript
    - Utilizado para realizar el intérprete y todo lo relacionado a el
    - Restricción: Únicamente javascript vanilla
- **Gramática:**
    - **Peggy**

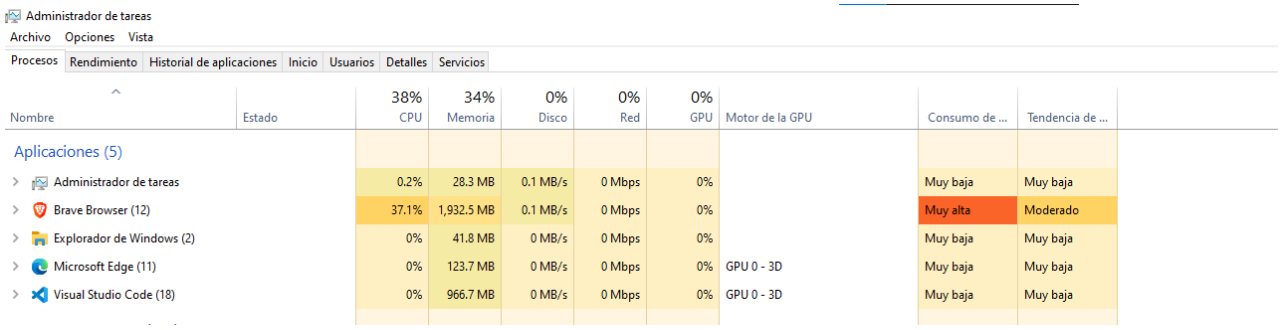
      Utiliza un enfoque basado en PEG (Parsing Expression Grammar), un tipo de gramática que especifica cómo se deben analizar las estructuras del texto.
- **Otros:**
    - **HTML y CSS**

      Utilizado para desarrollar la interfaz gráfica del intérprete del lenguaje Oakland

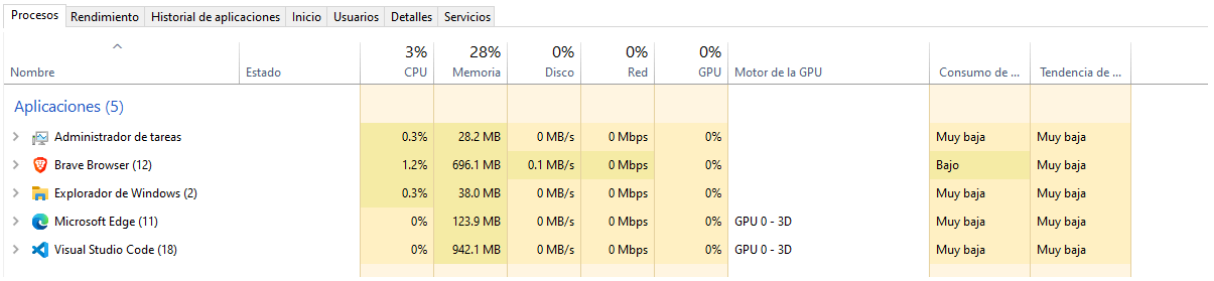
Requerimientos del equipo:

- EQUIPO :
    - 24.0 GB (23.8 GB utilizable)
    - Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz 2.90 GHz
    - Windows 10

- **Recursividad, fibonacci 10**



- **Operaciones y funciones simples, recursividad moderada**



Funcionalidades importantes del programa

- **Operaciones**

  Las operaciones en el lenguaje Oakland siguen el estándar establecido sobre la precedencia y jerarquía de los operadores. Esta jerarquía fue tomada en cuenta e implementada de manera que se respete y facilite el correcto reconocimiento de los tokens enviados, para esto se tomó en cuenta la naturaleza de la herramienta peggy que trabaja con gramáticas PEGS, en donde el operador de elección del PEG está ordenado, es decir si la primera alternativa tiene éxito, la segunda alternativa se ignora. Por tanto, la elección ordenada no es conmutativa, a diferencia de la elección no ordenada como en las gramáticas libres de contexto.

  La precedencia utilizada coincide con el siguiente cuadro:

  La precedencia se implementó en peggy a través de una gramática en donde la mayor precedencia se encuentra al final de la produccion expresion. Esto se debe a la naturaleza de las PEGS.

Asociatividad	Operadores	Categoría
N/A	clone new	clone y new
Izquierda	[	array()
N/A	++ —	Incremento/decremento
N/A	~ - (int) (float) (string) (array) (object) (bool) @	Tipos
N/A	instanceof	Tipos
Derecha	!	Lógico
Izquierda	* / %	Aritmético
Izquierda	+ - .	Aritmético y cadena
Izquierda	<< >>	Bit
N/A	< <= > >= <>	Comparación
N/A	== != === !==	Comparación
Izquierda	&	Bit y referencias
Izquierda	^	Bit
Izquierda	|	Bit
Izquierda	&&	Lógico
Izquierda	||	Lógico
Izquierda	? :	Ternario
Derecha	= += -= *= /= .= %= &= |= ^= <<= >>=	Asignación

  La asignación, al ser considerada con menor precedencia ocupa la primera opción de la producción Expresion. Esta regla permite la asignación de valores a variables o propiedades de objetos. El operador de asignación puede ser "=", "+=", o "-=". En el caso de una asignación a una propiedad de

un objeto o una llamada a una función, se verifica si la variable es una referencia (RefVar) o un atributo de una estructura (GetterStruct) esto para poder evaluar posteriormente las asignaciones como corresponde. Luego se encuentra “AsignarNewValue”. La asignación puede involucrar la creación de arreglos multidimensionales con la palabra clave “new” seguida del tipo y tamaño del arreglo, o simplemente asignar una expresión evaluada.

```
Assign = id:ID  exp:NestedSize _ op:("="/"+="/"-=") _ assign:Expresion { return crearNodo('assignIndiceArreglo', { id, exp, assign, op}) }
        /toAssign:Call _ "=" _ valorAttr: AsignarNewValue {
        if (toAssign instanceof nodos.RefVar) {return crearNodo('assign', { id:toAssign.id, assign:valorAttr, op:"="})}
        if (!(toAssign instanceof nodos.GetterStruct)) {console.log('Error: Para asignarle valor esta debe ser una propiedad de un objeto');}
        return crearNodo('setterStruct', { structObj: toAssign.callObj, attribute: toAssign.attribute, valorAttr })}
        /id:ID _ op:("+="/"-=") _ assign:AsignarNewValue { return crearNodo('assign', { id, assign, op}) }
        / Ternario

AsignarNewValue =  "{" _ elements:NestedArrayElements _ "}" {return elements;}
                /"new" _ tipoNew:TiposVar exp:NestedSize { return crearNodo('asignacionArregloNew', { tipoNew, exp, dimNew:exp.length }) }
                /  exp:Expresion  { return exp }
```

```
Ternario = condition:logicoOr _ IntTernario _ TrueB:Expresion _ ":" _ FalseB:Expresi
/ logicoOr


logicoOr = izq:logicoAnd expansion:(
  _ op:("||") _ der:logicoAnd { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (previousOp, actualOp) => {
      const { tipo, der } = actualOp
      return crearNodo('operacion', { op:tipo, izq: previousOp, der  })
    },
    izq
  )
}


logicoAnd = izq:DesIgualdad expansion:(
  _ op:("&&") _ der:DesIgualdad { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (previousOp, actualOp) => {
      const { tipo, der } = actualOp
      return crearNodo('operacion', { op:tipo, izq: previousOp, der  })
    },
    izq
  )
}

DesIgualdad = izq:Relacional expansion:(
  _ op:("!=" / "==") _ der:Relacional { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (previousOp, actualOp) => {
      const { tipo, der } = actualOp
      return crearNodo('operacion', { op:tipo, izq: previousOp, der  })
    },
    izq
  )
}

Relacional = izq:Suma expansion:(
  _ op:(">=" / "<="/">" / "<") _ der:Suma { return { tipo: op, der } }
)* {
  return expansion.reduce(
    (previousOp, actualOp) => {
      const { tipo, der } = actualOp
      return crearNodo('operacion', { op:tipo, izq: previousOp, der  })
    },
    izq
  )
}
```

Luego de la asignación se encuentran la operación ternaria y las operaciones lógicas. En el caso de las expresiones ternarias (que seleccionan entre dos expresiones basadas en una condición), se maneja primero la condición lógica, y según sea verdadera o falsa, se selecciona la expresión correspondiente. Las operaciones lógicas están organizadas por precedencia, desde los operadores lógicos || e && hasta las comparaciones (==, !=, >, <, <=, >=) . La precedencia de los operadores se ve reflejada en en la gramatica, esta evalua las siguiente producciones en el siguiente orden: Ternario, Logico Or, Logico And, desigualdad (==/!=), relacional (>=, <=,<,>)

Posteriormente se evalúan las posibles expresiones aritméticas comunes como lo son la multiplicación, división, suma y resta. Entre estos mismos operadores existe una jerarquía conocida es por eso que se dividen en

dos producciones, de primero existe SumSub, al ser la suma y la resta las que tienen menor precedencia en comparación con la multiplicacion, division y módulo.

```
SumaSub = izq:MulDivMod expansion:(
    _ op:("+" / "-") _ der:MulDivMod{ return { tipo: op, der } }
)* {
  return expansion.reduce(
    (previousOp, actualOp) => {
      const { tipo, der } = actualOp
      return crearNodo('operacion', { op:tipo, izq: previousOp, der  })
    },
    izq
  )
}

MulDivMod = izq:Unaria expansion:(
    _ op:("*" / "/" /"%") _ der:Unaria { return { tipo: op, der } }
)* {
    return expansion.reduce(
      (previousOp, actualOp) => {
        const { tipo, der } = actualOp
        return crearNodo('operacion', { op:tipo, izq: previousOp, der })
      },
      izq
    )
}
```

Este simple orden implementado evitar errores posteriores en los cálculos y hace posible los cálculos correctos. Por ejemplo: 5+4*4, en este caso al ser la multiplicación de mayor precedencia se resolverá antes la multiplicación (16) y luego se resolverá la suma con el resultado obtenido. Por último se procesan las operaciones Unarias como lo es el - y el signo !, estas al tener más precedencia que las operaciones aritméticas se encuentran después de ellas.

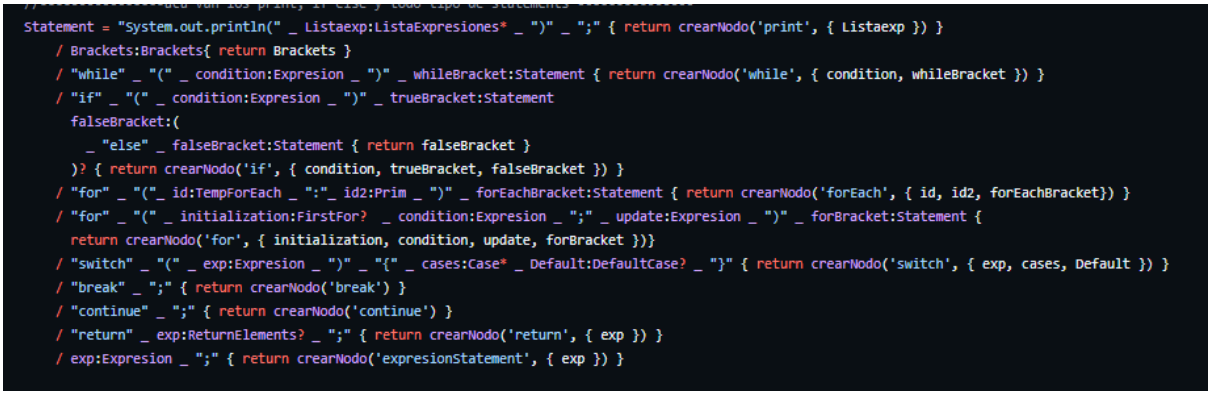
```
Unaria = ope:("-"/"!") _ num:Unaria { return crearNodo('unaria', { op: ope, exp: num }) }
/ Call

Call = callee:(laDELobj/TypeOF) _
Op:( "(" argumentos:Argz? ")" { return {argumentos, tipo: 'CallFuncion'} }
  / ("." _ id:ID _ { return { id, tipo: 'get' } }))*
  { const OpTipo = Op.reduce(
    (callObj, argumentos) => { const { tipo, id, argumentos:argg } = argumentos
      if (tipo === 'CallFuncion') {
        return crearNodo('call', { callee: callObj, argumentos: argg || [] })
      } else if(tipo === 'get'){
        return crearNodo('getterStruct', { callObj, attribute:id })
      }},
    callee)
  return OpTipo
}
```

Las llamadas a funciones tanto nativas como foráneas se encuentran en el mismo nivel de precedencia según la producción Call, y al igual que las operaciones aritméticas estas se asocian por la izquierda. Se establece una diferencia entra la llamada a funciones y la llamada a elementos de los structs, si bien ambas tienen la misma precedencias su manejo e interpretación es distinto.

Por último se evalúan las expresiones primarias o nativas, que son los elementos con mas precedencia que existen en el lenguaje ya que en base a esos se construye e interpreta todo. Esta producción captura los valores literales como enteros, flotantes, cadenas, caracteres, valores booleanos y referencias a variables (refVar). También permiten agrupaciones de expresiones entre paréntesis para forzar una evaluación más inmediata de las operaciones en comparación con otras operaciones con mayor precedencia.

- **Statements**

Los statements son las unidades fundamentales de control del flujo del programa, y pueden variar en función de las operaciones que realizan, como asignaciones, llamadas a funciones, control de flujo, o manipulaciones de estructuras de datos. Existen los statements declarativos, de los cuales se hablará luego en el manual, las expresiones las cuales se describieron anteriormente, y también existe los statements de flujo de control y condicionales, estos statements determinan cómo el programa debe avanzar o repetir ciertos bloques de código. La producción de statement también funciona como el pilar de la manera en la que se manejan los entornos, ya que la expresión de brackets corresponde a la creación de bloques. Al agrupar los brackets se simplificó tanto el manejo de la gramática como el manejo de los entornos.



Los estamentos que se trataron en este lenguaje y se ven implementados en la gramática son:

- **System.out.println():**

Un statement que imprime en la consola el valor de una o más expresiones. Este statement toma una lista de expresiones y las muestra como salida.

```
"System.out.println(" _ Listaexp:ListaExpresiones* _ ")" _ ";" { return crearNodo('print', { Listaexp }) }
```

- **while:**

Un bucle que ejecuta un bloque/brackets de código repetidamente mientras una condición sea verdadera.

```
/ "while" _ "(" _ condition:Expresion _ ")" _ whileBracket:Statement { return crearNodo('while', { condition, whileBracket }) }
```

- **if-else:**

Sentencia condicional que ejecuta un bloque de código si una condición es verdadera (if). Puede existir también un bloque alternativo (else) si la condición es falsa.

```
/ "if" _ "(" _ condition:Expresion _ ")" _ trueBracket:Statement
  falseBracket:(
    _ "else" _ falseBracket:Statement { return falseBracket }
```

- **forEach:**

Este corresponde a un for especializado que itera sobre un arreglo.

```
/ "for" _ "(" _ id:TempForEach _ ":" _ id2:Prim _ ")" _ forEachBracket:Statement { return crearNodo('forEach', { id, id2, forEachBracket})
```

- **for:**

Un bucle for tradicional, que incluye 3 partes: una inicialización, una condición de finalización, y una expresión de actualización (esto para poder salir eventualmente del bucle).

```
/ "for" _ "(" _ initialization:FirstFor?  _ condition:Expresion _ ";" _ update:Expresion _ ")" _ forBracket:Statement {
  return crearNodo('for', { initialization, condition, update, forBracket })}
```

- **switch-case:**

Una estructura de control que selecciona entre múltiples bloques de código basados en el valor de una expresión. Cada bloque está asociado a un case que compara un valor específico con el valor de la expresión.

- **Break, return, continue**

break:
Detiene la ejecución de un bucle o de un switch-case y transfiere el control fuera de esa estructura.
continue:
Interrumpe la iteración actual de un for o while y pasa a la siguiente iteración.
return:
Finaliza la ejecución de una función y devuelve un valor a la llamada de función, este valor puede o no existir

```
/ "break" _ ";" { return crearNodo('break') }
/ "continue" _ ";" { return crearNodo('continue') }
/ "return" _ exp:ReturnElements? _ ";" { return crearNodo('return', { exp }) }
```

- **expresionStatement:**

Representa una sentencia de expresión, esta corresponde a las expresiones abordadas en el primer punto como lo es la asignación, operaciones lógicas, operaciones unarias, llamadas, primitivos, entre otros.

- **Declaración de variables**

La estructura general declarada permite declarar variables, arreglos y structs. Estas tres divisiones posteriormente se dividen en mas debido a la cantidad de formas que existen en el lenguaje Oakland de declarar una variable.

```
//>MAIN
Declaracion = decl:DecFunction _ { return decl }
            / decl:DecVariable _ { return decl }
            / decl:DecStructz _ { return decl }
            / statement:Statement _ { return statement }
```

- Declaración de variables y arreglos

```
//---------------declaracion de variables, clases y funciones
DecVariable = "var" _ id:ID _ "=" _ exp:Expresion _ ";" { return crearNodo('declaracionVar', { id, exp}) }
            / DeclaracionArray
            / tipoz:TiposVar _ id:ID _
            TipadoValor:(
              "=" _ TipadoValor:Expresion _  { return  TipadoValor  }
            )?_";" { return crearNodo('declaracionVarTipo', { id, exp:TipadoValor, tipoz }) }
```

  - Se utiliza la palabra clave `var` para declarar una variable, luego especifica el nombre de la variable (`id`), seguido por una expresión de asignación y una expresión.
  - Existe la declaración de variables indicando únicamente el tipo y el nombre, este tipo de variables asigna el valor default de null.
  - DeclaracionArray
    Esta producción cubre la declaración de arreglos o matrices. Utiliza la regla DeclaracionArray, que permite definir un arreglo de un tipo específico, con una expresión que indica su tamaño o su contenido.

```
DeclaracionArray = tipoz:TiposVar "[]" dims:Corchetes* _ id:ID _ "=" _ exp:MainArrayElements _ ";" {
  return crearNodo('declaracionArreglo', {id,exp, tipoz, dimension: dims.length+1});
}


Corchetes ="[]" { return text() }
```

El contenido de los arreglos puede variar es por eso que este la producción “MainArrayElements”, en donde se declaran las 3 diferentes formas de declarara un array:

- explícitamente declarando los valores. Por ejemplo : {1,2,3,4}
- Detallando la creacion de un arreglo vario, unicamente indicando el numero de elementos. Por ejemplo: new int[4]
- Realizando una copia de un arreglo ya existente

```
MainArrayElements = "{" _ elements:NestedArrayElements _ "}" {return elements;}
                  /"new" _ tipoNew:TiposVar exp:NestedSize { return crearNodo('asignacionArregloNew', { tipoNew, exp, dimNew:exp.length }) }
                  / id:ID { return crearNodo('refVar', { id }) }

NestedSize = "[" _ exp:Expresion _ "]" _ tail:("[" _ expTail:Expresion _ "]")* {
  return [exp].concat(tail.map(function(d) { return d[2]; }));
}

//lleno de expresiones o mas arreglos(matriz)- recursivo
NestedArrayElements = head:(Expresion / MainArrayElements) tail:(_ "," _ (Expresion / MainArrayElements))* {
  return [head].concat(tail.map(function(element) { return element[3]; }));
}
```

- Declaración de structs

Esta producción incluye la declaración del struct como tal y la creación de una instancia de dicho struct.

La declaración de un struct consiste en la palabra reservada “struct”, un id que lo identifique y una serie de declaraciones sobre los atributos del mismo. La producción StructAttrbtz se encarga de definir qué puede ser un atributo de un struct (que puede ser una variable o una instancia de otro struct).

```
DecStructz = "struct" _ id:ID _ "{" _ decl:StructAttrbtz* _ "}" _ ";" { return crearNodo('declaracionStructz', { id, declaraciones: decl }) }
           /idStruct:ID _ id:ID _"=" _ exp:Expresion _ ";" { return crearNodo('creacionInstanceStruct', { idStruct, id, exp }) }

//----STRCUTS UNICAMENTE ATRIBUTOS
StructAttrbtz= decl:DecVariable _ { return decl }
           /idStruct:ID _ id:ID _ ";"{ return crearNodo('creacionInstanceStruct', { idStruct, id, exp: null }) }
```

La creación de una instancia de un struct se caracteriza por tener el id del struct a instancias, seguido de el id que será el nombre de dicha instancia y una expresión. Esta expresión hace referencia a la producción expresión en donde también se encuentra definida la declaración de atributos.

```
attStructs = first:attItem rest:("," _ attItem)* {
    return [first, ...rest.map(item => item[2])];
}

attItem = id:ID _ ":" _ prim:Prim {
    return { id, valor: prim.valor, tipo: prim.tipo };
}
```

Gramática

```
programa = _ decl:Declaracion* _

//>MAIN
Declaracion = decl:DecFunction _
            / decl:DecVariable _
            / decl:DecStructz _
            / statement:Statement _


DecFunction = tipoFunc:"void" _ id:ID _ "(" _ Parameters:Parametros? _ ")" _ brackets:Brackets
            /tipoFunc:TiposVar res:Corchetes* _ id:ID _ "(" _ Parameters:Parametros? _ ")" _ brackets:Brackets
//---------------declaracion de variables, clases y funciones
DecVariable = "var" _ id:ID _ "=" _ exp:Expresion _ ";"
            / DeclaracionArray
            / tipoz:TiposVar _ id:ID _  TipadoValor:("=" _ TipadoValor:Expresion _)?_";"

//>>>>>>>>>>>>>>>>>>>>>>ARREGLOS>>>>>>>>>>>>>>>>>>>>>>>>>>
//dimension de acuerdo a los corchetes (array o matrix )
DeclaracionArray = tipoz:TiposVar "[]" dims:Corchetes* _ id:ID _ "=" _ exp:MainArrayElements _ ";"
Corchetes ="[]"

MainArrayElements = "{" _ elements:NestedArrayElements _ "}"
                  /"new" _ tipoNew:TiposVar exp:NestedSize
                  / id:ID

NestedSize = "[" _ exp:Expresion _ "]" _ tail:("[" _ expTail:Expresion _ "]")*
//lleno de expresiones o mas arreglos(matriz)- recursivo
NestedArrayElements = head:(Expresion / MainArrayElements) tail:(_ "," _ (Expresion / MainArrayElements))*

//---------------DECLARACION STRUCTZZZZZZZ
DecStructz = "struct" _ id:ID _ "{" _ decl:StructAttrbtz* _ "}" _ ";"
           /idStruct:ID _ id:ID _"=" _ exp:Expresion _ ";"

//----STRCUTS UNICAMENTE ATRIBUTOS
StructAttrbtz= decl:DecVariable _
           /idStruct:ID _ id:ID _ ";"


//----------------aca van los print, if else y todo tipo de statements ---------------
Statement = "System.out.println(" _ Listaexp:ListaExpresiones* _ ")" _ ";"
    / Brackets:Brackets
    / "while" _ "(" _ condition:Expresion _ ")" _ whileBracket:Statement
    / "if" _ "(" _ condition:Expresion _ ")" _ trueBracket:Statement falseBracket:( _ "else" _ falseBracket:Statement)?
    / "for" _ "(" _ id:TempForEach _ ":" _ id2:Prim _ ")" _ forEachBracket:Statement
    / "for" _ "(" _ initialization:FirstFor?  _ condition:Expresion _ ";" _ update:Expresion _ ")" _ forBracket:Statement
    / "switch" _ "(" _ exp:Expresion _ ")" _ "{" _ cases:Case* _ Default:DefaultCase? _ "}"
    / "break" _ ";"
    / "continue" _ ";"
    / "return" _ exp:ReturnElements? _ ";"
    / exp:Expresion _ ";"

 //SEPARARA PARA BRACKETS
 Brackets="{" _ decl:Declaracion* _ "}"

 TempForEach = tipoz:TiposVar _ id:ID

 //----------------PARAMETROS DE FUNCIONES------------------

 Parametros = first:Parametro _ rest:("," _ param:Parametro
 Parametro = tipo:TiposVar _ id:ID
 // SEPARAR PRIMERA CONDICION DE FOR

 ListaExpresiones = "," _ exp:Expresion _
                 /exp: Expresion _

 // para arreglar que no vengan dos ;; y de error
 FirstFor = decl:DecVariable
         / exp:Expresion _ ";"
         / ";"

 Case = "case" _ valorCase:Expresion _ ":" _ caseBracket:Declaracion*

 DefaultCase = "default" _ ":" _ defaultBracket:Declaracion*

 //>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

 //restriccion de NO empezar con un numero
 ID = [a-zA-Z_][a-zA-Z0-9_]*
 String = "\"" ( "\\" "\"" / [^\"] )* "\""
 Charr = "\'" [^\']* "\'"
 IntTernario = "?"
 ''
```

```
Expresion = Assign

Assign = id:ID  exp:NestedSize _ op:("="/"+="/"-=") _ assign:Expresion
        /toAssign:Call _ "=" _ valorAttr: AsignarNewValue
        /id:ID _ op:("+="/"-=") _ assign:AsignarNewValue
        / Ternario

AsignarNewValue =  "{" _ elements:NestedArrayElements _ "}"
                /"new" _ tipoNew:TiposVar exp:NestedSize
                /  exp:Expresion

Ternario = condition:logicoOr _ IntTernario _ TrueB:Expresion _ ":" _ FalseB:Expresion
/ logicoOr

logicoOr = izq:logicoAnd expansion:(_ op:("||") _ der:logicoAnd)*

logicoAnd = izq:DesIgualdad expansion:(_ op:("&&") _ der:DesIgualdad )*

DesIgualdad = izq:Relacional expansion:( _ op:("!=" / "==") _ der:Relacional )*

Relacional = izq:SumaSub expansion:(_ op:(">=" / "<="/">" / "<") _ der:SumaSub)*


 SumaSub = izq:MulDivMod expansion:(_ op:("+" / "-") _ der:MulDivMod)*

 MulDivMod = izq:Unaria expansion:( _ op:("*" / "/" /"%") _ der:Unaria )*

 Unaria = ope:("-"/"!") _ num:Unaria
 / Call

 Call = callee:(laDELobj/TypeOF) _
 Op:( "(" argumentos:Argz? ")"
   / ("." _ id:ID _ ))*

 laDELobj= id:"Object.keys"

 TypeOF = "typeof"  _ argumentos:TypeOF _
          / Prim

 Argz = argum:ReturnElements _ argumentos:("," _ exp:ReturnElements

 ReturnElements = "{" _ elements:NestedArrayElements _ "}"
                 /"new" _ tipoNew:TiposVar exp:NestedSize
                 /  exp:Expresion

 tipoFloat =[0-9]+( "." [0-9]+ )
 tipoInt = [0-9]+

 Prim =  floatN:tipoFloat
   / intN:tipoInt
   / str:String
   / char:Charr
   /"true"
   /"false"
   / "(" _ exp:Expresion _ ")"
   / "[" _ exp:Expresion _ "]"
   / "null"
   / id:ID "{" _ attributes:attStructs? _ "}"
   /id:ID "."funcion:("length"/"join"/"indexOf") exp:("(" exp: Argz? ")"
   / id:ID exp:NestedSize
   / id:ID

attStructs = first:attItem rest:("," _ attItem)*
attItem = id:ID _ ":" _ prim:Prim

TiposVar = "int"
/ "float"
/ "boolean"
/ "char"
/ "string"

_ = ([ \t\n\r] / Comentarios)*

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