

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Organización de Lenguajes y Compiladores 2

Manual de Técnico:
Proyecto 1

Hecho por:
Walter Daniel Jiménez Hernández
201901108
Fecha: 17/09/2024

Contenido

1. Introducción
2. Objetivos
3. Conceptos Previos
4. Especificación Técnica
 - a. Requisitos de Hardware
 - b. Requisitos de Software
 - i. Sistema Operativo
 - ii. Lenguaje de Programación
 - iii. Tecnologías usadas
5. Lógica del Programa
 - a. Analizadores
 - i. Sintáctico
 - ii. Gramática
 - b. Clases Utilizadas
 - i. Instruction
 - ii. Errores
 - iii. Tree
 - iv. SymbolsTable
6. Link del Repositorio

Introducción

El programa cumple con la función de poder reconocer el lenguaje de programación oak que es una copia del lenguaje Java pero con una sintaxis más simplificada.

Objetivo

El objetivo es poder aplicar el conocimiento sobre análisis léxico, análisis sintáctico y análisis semántico.

- Implementar análisis léxico.
- Implementar análisis sintáctico.
- Implementar análisis semántico.

Conocimientos Previos

- Programación en Java
- Programación en JavaScript
- Programación Orientada a Objetos
- Estructuras lineales
- HTML
- CSS
- GITHUB
- PEGGY JS

Especificaciones Técnicas

Requisitos de Hardware

- Procesador Dual Core o Superior
- Mínimo 2 GB de RAM

Requisitos de Software

- Sistema Operativo
 - Windows 7
 - Windows 8
 - Windows 10
 - Windows 11
 - Distribuciones de Linux
 - MAC OS
- Navegador
 - Chrome
 - Edge
 - Mozilla Firefox
 - Safari
 - Opera
 - Brave
- Lenguaje de Programación
 - JavaScript
- Tecnologías Usadas
 - Visual Studio Code
 - Git
 - Github

Lógica del Programa

Analizadores

Léxico: Es utilizado para leer cada uno de los tokens de la entrada que se estará utilizando al igual que las expresiones regulares que componen al mismo.

Sintáctico: En este archivo se escribe la gramática con sintaxis de PEGGY JS

```
S = INSTRUCTIONS

INSTRUCTIONS = ins:INSTRUCTION arr:INSTRUCTIONS { arr.unshift(ins); return arr }
              / ins:INSTRUCTION { return new Array(ins) }

INSTRUCTION = PRINT
              / BREAK
              / CONTINUE
              / RETURN
              / IF
              / WHILE
              / SWITCH
              / FOR
              / FOR2
              / STRUCT
              / SSTRUCT
              / STATEMENT
              / @ASSIGNMENT - ";" -
              / @EXPRESSIONS - ";" -
```

Como se puede ver en la imagen no se cuenta con un analizador léxico explícito y en la misma sintaxis se van agregando las producciones junto a las expresiones regulares necesarias para representar los tokens

```
S = INSTRUCTIONS

INSTRUCTIONS = ins:INSTRUCTION arr:INSTRUCTIONS { arr.unshift(ins); return arr }
              / ins:INSTRUCTION { return new Array(ins) }

INSTRUCTION = PRINT
              / BREAK
              / CONTINUE
              / RETURN
              / IF
              / WHILE
              / SWITCH
              / FOR
              / FOR2
              / STRUCT
              / SSTRUCT
              / STATEMENT
              / @ASSIGNMENT - ";" -
              / @EXPRESSIONS - ";" -
```

Al ser una gramática recursiva por la derecha se debe realizar algo diferente ya que se debe ir reduciendo las producciones para poder operar de izquierda a derecha como normalmente se hace en las expresiones aritméticas, lógicas y relacionales.

```
PREC7 = op1:PREC6 _ expanded:("(" _ PREC6)* { const {start:{line, column}} = location(); const res = expanded.reduce(Log
res.setLine(line); res.setColumn(column); return res}

PREC6 = op1:PREC5 _ expanded:("&&" _ PREC5)* { const {start:{line, column}} = location(); const res = expanded.reduce(Log
res.setLine(line); res.setColumn(column); return res}

PREC5 = op1:PREC4 _ expanded:("==" / "!=" _ PREC4)* { const {start:{line, column}} = location(); const res = expanded.red
res.setLine(line); res.setColumn(column); return res}

PREC4 = op1:PREC3 _ expanded:(">=" / ">" / "<=" / "<" _ PREC3)* { const {start:{line, column}} = location(); const res =
res.setLine(line); res.setColumn(column); return res}

PREC3 = op1:PREC2 _ expanded:("+ " / "- " _ PREC2)* { const {start:{line, column}} = location(); const res = expanded.reduc
res.setLine(line); res.setColumn(column); return res}

PREC2 = op1:PREC1 _ expanded:("*" / "/" / "%" _ PREC1)* { const {start:{line, column}} = location(); const res = expanded
res.setLine(line); res.setColumn(column); return res}

PREC1 = @NATIVOS _

NATIVOS = "(" _ @EXPRESSIONS _ ")"
/("-") _ op1:PREC2 { const {start:{line, column}} = location(); const res = new Arithmetics(null, op1, Operators.DE
res.setLine(line); res.setColumn(column); return res}
/("!") _ op1:PREC2 { const {start:{line, column}} = location(); const res = new Logical(null, op1, LogOperator.NOT,
res.setLine(line); res.setColumn(column); return res}
/ STRING
/ CHAR
/ DOUBLE
/ INTEGER
/ BOOLEAN
/ "parseInt" _ "(" _ exp:EXPRESSIONS _ ")" { const {start:{line, column}} = location(); return new ParseInt(exp,
/ "parseFloat" _ "(" _ exp:EXPRESSIONS _ ")" { const {start:{line, column}} = location(); return new ParseFloat(exp
```

Se debe colocar los espacios en blanco y los comentarios en una producción para poder reconocerlos al momento de estar leyendo el código.

```

STRING = "\"" chars:STRINGS* "\"" { const {start:{line, column}} = location();
return new Native(new Type(dtype.STRING), chars.join(""), line, column) }

STRINGS = [^"]

CHAR = "'" char:CHARS "'" { const {start:{line, column}} = location();
return new Native(new Type(dtype.CHAR), char[0], line, column) }

CHARS = [^']

INTEGER = int:[0-9]+ { const {start:{line, column}} = location();
return new Native(new Type(dtype.INTEGER), parseInt(int.join(""), 10), line, column) }

DOUBLE = int:[0-9]+ "." double:[0-9]+ { const {start:{line, column}} = location();
return new Native(new Type(dtype.DOUBLE), parseFloat(int.join("") + "." + double.join(""), 10), line, column) }

BOOLEAN = bool:("true"/"false") { const {start:{line, column}} = location()
return new Native(new Type(dtype.BOOLEAN), bool == "true", line, column) }

ID = [a-zA-Z_][a-zA-Z0-9_]* {return text() }

_ "whitespace" = ([ \t\n\r] / COMENTARIOS)*

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

```

Gramática: Es la gramática limpia utilizada en el analizador sintáctico. Para poder leerla completa esta se encontrará en la carpeta Manuales.

<s> ::= <instructions>

<instructions> ::= <instruction> <instructions>

| <instruction>

<instruction> ::= <print>

| <break>

| <continue>

| <return>

| <if>

| <while>

| <switch>

| <for>

| <foreach>

| <struct>

| <sstruct>

| <statement>

| <assignment>

| <expression>

<print> ::= "System.out.println(" <expressionPrint> ")"

<statement> ::= <type> <id> "=" <expression>

| "var" <id> "=" <expression>

| <type> <id> "[]" + <id> "{" <array2> "}"

<params> ::= <type> <id> ", " <params>

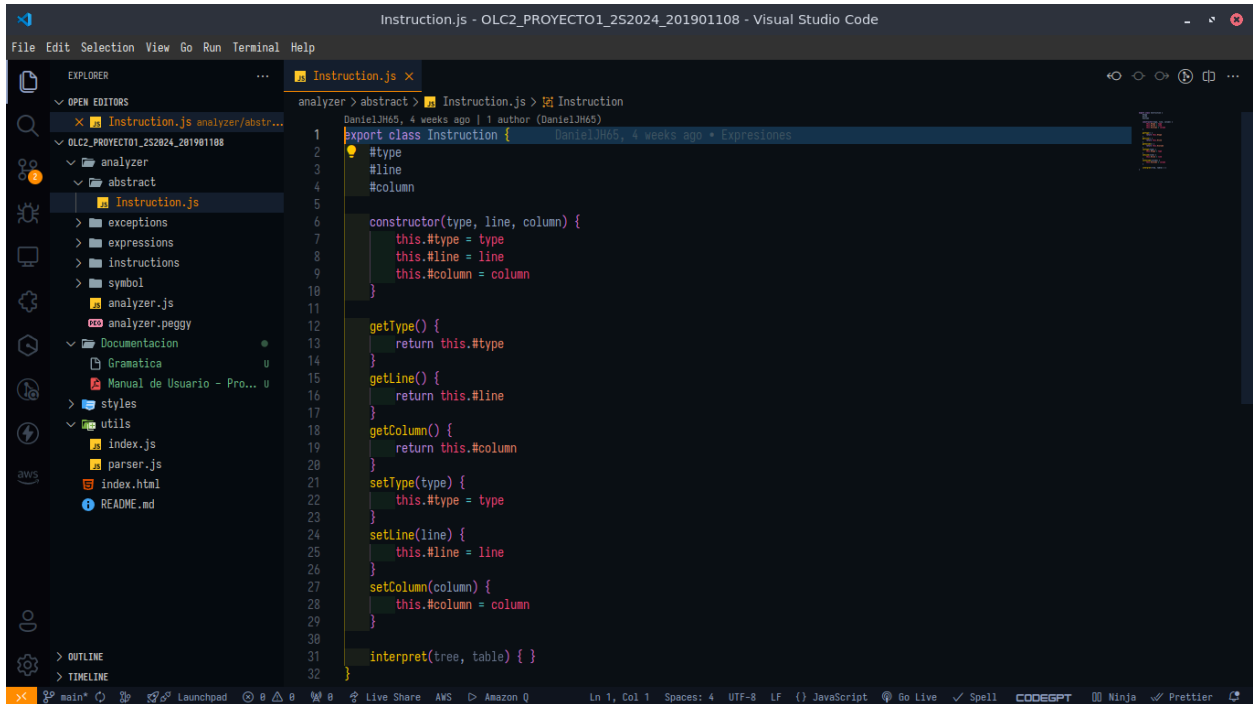
| <type> <id>

<array2> ::= "{" <values> "}" ", " <array2>

| "{" <values> "}"

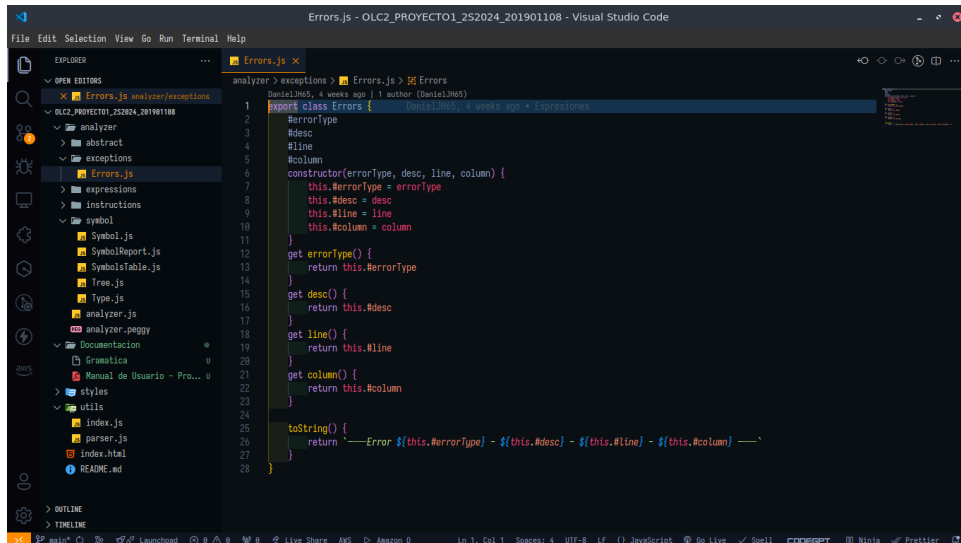
Clases Utilizadas

Instruction: Esta clase se utiliza para poder manejar todo relacionado a las instrucciones y expresiones que vienen en el lenguaje para poder utilizar el patrón interprete:



```
1 export class Instruction {
2   #type
3   #line
4   #column
5
6   constructor(type, line, column) {
7     this.#type = type
8     this.#line = line
9     this.#column = column
10  }
11
12  getType() {
13    return this.#type
14  }
15  getLine() {
16    return this.#line
17  }
18  getColumn() {
19    return this.#column
20  }
21  setType(type) {
22    this.#type = type
23  }
24  setLine(line) {
25    this.#line = line
26  }
27  setColumn(column) {
28    this.#column = column
29  }
30
31  interpret(tree, table) {}
32 }
```

Errores: Esta clase se utiliza para poder representar todos los errores que se detecten durante la ejecución del programa. Esta clase cuenta con todos los atributos necesarios para poder almacenar cada detalle.



```
1 export class Errors {
2   #errorType
3   #desc
4   #line
5   #column
6
7   constructor(errorType, desc, line, column) {
8     this.#errorType = errorType
9     this.#desc = desc
10    this.#line = line
11    this.#column = column
12  }
13
14  get errorType() {
15    return this.#errorType
16  }
17  get desc() {
18    return this.#desc
19  }
20  get line() {
21    return this.#line
22  }
23  get column() {
24    return this.#column
25  }
26
27  toString() {
28    return `Error ${this.#errorType} - ${this.#desc} - ${this.#line} - ${this.#column}`
29  }
30 }
```


Tree: Esta clase se utiliza para poder manejar todas las instrucciones ya que se guardan en una lista y se puede recorrer cada una en orden como si fuera el árbol de análisis de sintaxis.

```
export class Tree {
  #instructions
  #console
  #globalTable
  #errors
  constructor(instructions) {
    this.#instructions = instructions
    this.#console = ""
    this.#globalTable = new SymbolsTable(null)
    this.#errors = new Array()
    this.structs = new Array()
    this.SymbolsReport = new Array()
  }
}
```

You, 3 hours ago • Uncommitted changes

Type: Esta clase se utiliza para poder manejar los tipos durante la ejecución del programa.

```
export class Type {
  #type
  constructor(type) {
    this.#type = type
  }
  setType(type) {
    this.#type = type
  }
  getType() {
    return this.#type
  }
}

export const dataType = Object.freeze({
  "INTEGER": "int",
  "DOUBLE": "float",
  "CHAR": "char",
  "STRING": "string",
  "BOOLEAN": "boolean",
  "VOID": "void",
  "STRUCT": "struct",
  "NULL": "null"
})
```

You, 7 hours ago

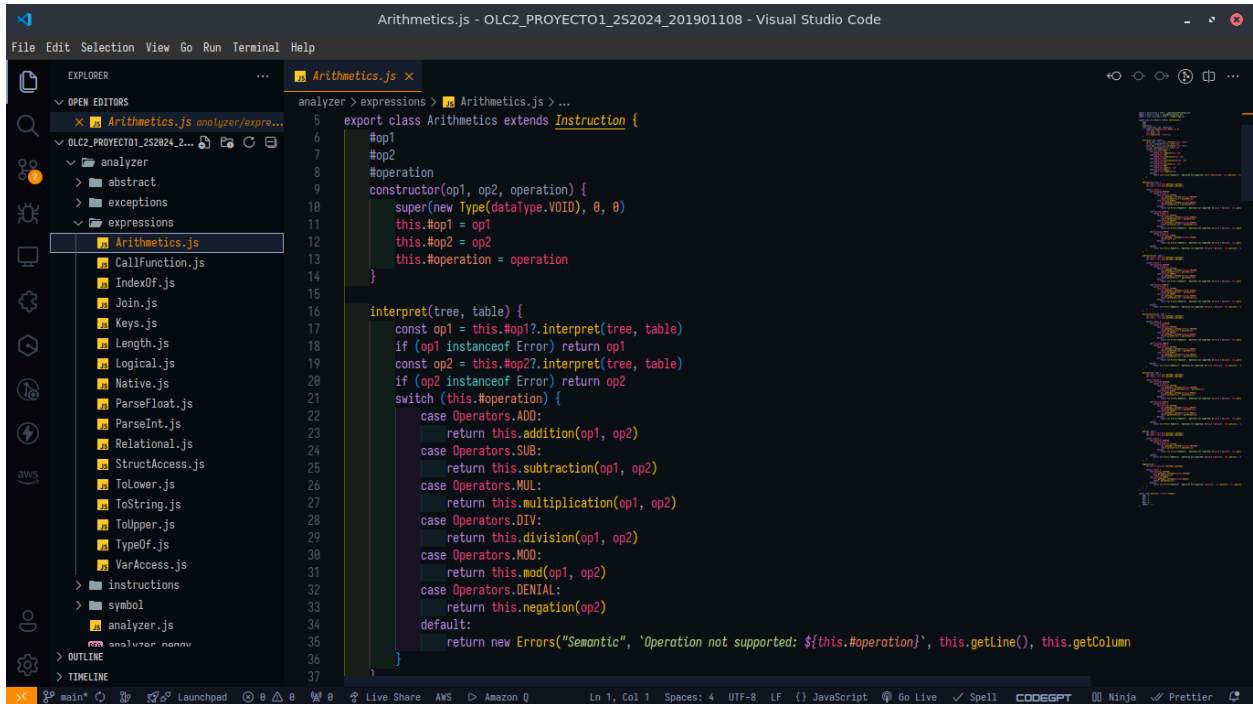
Symbol: Esta clase se utiliza para poder estructurar los datos de la tabla de símbolos y asegurarse que contenga todos los atributos necesarios.

```
export class Symbol {  
    #type  
    #id  
    #value  
    constructor(type, id, value) {  
        this.#type = type  
        this.#id = id  
        this.#value = value  
    }  
}
```

SymbolsTable: Esta clase se utiliza para poder almacenar las tablas de símbolos y los entornos que se manejan.

```
export class SymbolsTable {  
    #previousTable  
    #table  
    #name  
  
    constructor(previousTable) {  
        this.previousTable = previousTable  
        this.table = new Map()  
        this.functions = new Map()  
        this.name = ""  
    }  
  
    getPreviousTable() {  
        return this.previousTable  
    }  
    getTable() {  
        return this.table  
    }  
    getName() {  
        return this.name  
    }  
    getFunctions() {  
        return this.functions  
    }  
    setPreviousTable(previousTable) {  
        this.previousTable = previousTable  
    }  
    setTable(table) {  
        this.table = table  
    }  
    setName(name) {  
        this.name = name  
    }  
}
```

Arithmetics: En esta clase se manejan todas las operaciones aritméticas



```
5 export class Arithmetics extends Instruction {
6   #op1
7   #op2
8   #operation
9   constructor(op1, op2, operation) {
10     super(new Type(dataType.VOID), 0, 0)
11     this.#op1 = op1
12     this.#op2 = op2
13     this.#operation = operation
14   }
15
16   interpret(tree, table) {
17     const op1 = this.#op1?.interpret(tree, table)
18     if (op1 instanceof Error) return op1
19     const op2 = this.#op2?.interpret(tree, table)
20     if (op2 instanceof Error) return op2
21     switch (this.#operation) {
22       case Operators.ADD:
23         return this.addition(op1, op2)
24       case Operators.SUB:
25         return this.subtraction(op1, op2)
26       case Operators.MUL:
27         return this.multiplication(op1, op2)
28       case Operators.DIV:
29         return this.division(op1, op2)
30       case Operators.MOD:
31         return this.mod(op1, op2)
32       case Operators.DENIAL:
33         return this.negation(op2)
34       default:
35         return new Errors("Semantic", `Operation not supported: ${this.#operation}`, this.getLine(), this.getColumn)
36     }
37   }
38 }
```

Print: Esta clase recibe las expresiones y las muestra en la consola.

```

import { Instruction } from "../abstract/Instruction.js"
import { Errors } from "../exceptions/Errors.js"
import { Type, dataType } from "../symbol/Type.js"

You, 2 hours ago | 2 authors (You and one other)
export class Print extends Instruction {
  constructor(expressions, line, column) {
    super(new Type(dataType.VOID), line, column)
    this.expressions = expressions
  }

  interpret(tree, table) {
    for (let expression of this.expressions) {
      let value = expression.interpret(tree, table)
      if (value instanceof Errors) {
        tree.setConsole(tree.getConsole() + "null\n")
        return value
      }
      if (expression.getType().getType() == dataType.STRING) {
        value = value.toString()
        value = value.replaceAll("\\'", "'");
        value = value.replaceAll("\\\"", "\"");
        value = value.replaceAll("\\t", "\t");
        value = value.replaceAll("\\n", "\n");
        value = value.replaceAll("\\\\", "\\");
      }
      tree.setConsole(tree.getConsole() + value?.toString() + " ")
    }
    tree.setConsole(tree.getConsole() + "\n")
    return null
  }
}

```

VarStatement: Se utiliza para poder manejar las declaraciones de variable, verifica que el id no exista y la expresión corresponda al tipo de la variable.

```

export class VarStatement extends Instruction {
  constructor(type, id, expression, line, column) {
    super(type, line, column)
    this.id = id
    this.expression = expression
  }

  interpret(tree, table) {
    if (table.variableExists(this.id)) return new Errors("Semantic", `Variable ${this.id} already exists`, this.getLine())
    let value = this.expression?.interpret(tree, table)
    if (value instanceof Errors) {
      const sym = new Symbol(this.getType(), this.id, null)
      table.setVariable(sym)
      tree.SymbolsReport.push(new SymbolReport(this.id, "Variable", this.getType().getType(), value, table.getName(), this.getLine(), this.getColumn()))
      return value
    }
    if (value == undefined) {
      this.expression = new Native(this.getType(), null, this.getLine(), this.getColumn())
      value = this.expression?.interpret(tree, table)
    }
    if (this.getType().getType() == dataType.VOID) this.getType().setType(this.expression.getType().getType())
    if (this.getType().getType() == dataType.DOUBLE && this.expression?.getType().getType() == dataType.INTEGER) this.getType().setType(dataType.DOUBLE)
    if (this.getType().getType() != this.expression?.getType().getType()) return new Errors("Semantic", `Type mismatch expected ${this.getType().getType()} but got ${this.expression.getType().getType()}`, this.getLine(), this.getColumn())
    const sym = new Symbol(this.getType(), this.id, value)
    table.setVariable(sym)
    tree.SymbolsReport.push(new SymbolReport(this.id, "Variable", this.getType().getType(), value, table.getName(), this.getLine(), this.getColumn()))
    return null
  }
}

```

VarAssignment: Esta clase maneja la asignación de variables.

```

export class VarAssignment extends Instruction {
  constructor(id, expression, row, col) {
    super(new Type(dataType.VOID), row, col)
    this.id = id
    this.expression = expression
  }

  interpret(tree, table) {
    const variable = table.getVariable(this.id)
    if (variable == null) return new Errors("Semantic", `Variable ${this.id} does not exists`, this.getLine(), this.getColumn())
    this.getType().setType(variable.getType().getType())
    const value = this.expression?.interpret(tree, table)

    if (value instanceof Error) return value
    if (this.expression.getType().getType() != this.getType().getType()) {
      return new Errors("Semantic", `Type mismatch, expected ${this.getType().getType()} but got ${this.expression.getType().getType()}`, this.getLine(), this.getColumn())
    }

    variable.setValue(value)
    return null
  }
}

```

Link del Repositorio

https://github.com/DanielJH65/OLC2_PROYECTO1_2S2024_201901108