



MANUAL TECNICO

T-Swift

Juan Pedro Valle Lema
202101648

Encabezado

Desarrollador : Juan Pedro Valle Lema

Carnet: 202101648

Curso: Organización de Lenguajes y Compiladores 2

Sección: B

Lenguaje usado: Golang

IDE Utilizada: Visual Studio Code

Nombre del Sistema: Windows 10

Programa: T-Swift

Backend: localhost:3002

Frontend: localhost:3000

Librerías o Frameworks Utilizados:

Desarrollo Backend:

Golang

ANTLR4

Fiber

Desarrollo Frontend:

React

Principio, técnica o paradigma aplicado de programación

Se utilizó el paradigma de programación orientada a objetos en este caso por medio de structs e interfaces junto a el patrón interprete esto ya que de esta forma el desarrollo del programa se volvió mucho más fácil al manejar clases abstractas como lo eran la clase abstracta expresión y la clase abstracta instrucción las cuales eran usadas para generar otras como la clase de instrucción Print o la clase de Expresión Operación. También se utilizó una arquitectura de tipo cliente servidor en el que el usuario del programa realiza peticiones de tipo post a nuestro servidor backend desde la interfaz gráfica es decir el frontend.

Archivo

Es importante saber que la única extensión de archivo aceptada es la extensión swift. Este es un archivo de texto plano que contiene la información que usara nuestro programa para realizar la interpretación de este lenguaje de programación.

```
1 print("-----")
2 print("----ARCHIVO BASICO----")
3 print("-----")
4
5 var bol = false
6 var bol2 = !bol
7 var cad1 = "imprimir"
8 var cad2 = "cadena valida"
9
10 var val1 = 7 - (5 + 10 * (2 + 4 * (5 + 2 * 3)) - 8 * 3 * 3) + 50 * (6 * 2)
11 var val2 = (2 * 2 * 2 * 2) - 9 - (8 - 6 + (3 * 3 - 6 * 5 - 7 - (9 + 7 * 7 * 7) + 10) - 5) + 8 - (6 - 5 * (2 * 3))
12 var val3 = val1 + ((2 + val2 * 3) + 1 - ((2 * 2 * 2) - 2) * 2) - 2
13
14 print("El valor de val1 es:", val1)
15 print("El valor de val2 es:", val2)
16 print("El valor de val3 es:", val3)
17 print("El resultado de la operación es:", val3)
18 print("El valor de bol es:", bol)
19 print("El valor de cad1 es:", cad1)
20 print("El valor de cad2 es:", cad2)
21 print("El valor de bol2:", bol2)
22
23 var a = 100
24 var b = 100
25 var c = 7
26 var f = true
27 var j: Float = 10.0
28 var k: Float = 10.0
29
30 print("")
31 print("")
32
33 if a > b || b < c {
34     print(">>>>>> Esto no debería de imprimirse")
35 } else {
36     print(">>>>>> Esto debería de imprimirse")
37 }
38
39 if (a == b && j == k) || 14 != c {
40     print(">>>>>> Esto debería de imprimirse")
41 } else {
42     print(">>>>>> Esto no debería de imprimirse")
43 }
44
45 var val = 5
46 var resp = 5
47 var valorVerdadero = 100
48
49 if (valorVerdadero == (50 + 50 + (val - val))) && (!(true)) {
50     print(">>>>>> En este lugar debería de entrar :)")
51     valorVerdadero = 50
52 } else if (f || (valorVerdadero > 50)) && ((resp != 100) && !(f)) {
53     print(">>>>>> Aca no debería de entrar :ccc")
54     valorVerdadero = 70
55 } else {
56     print(">>>>>> Aca no debería de entrar :cccc")
57 }
58
59 var x1 = 15
60
61 if x1 % 2 == 0 {
62     print(">>>>>> numeroPar ingreso a if verdadero,", x1, "es par")
63 } else {
64     print(">>>>>> numeroPar ingreso a if falso,", x1, "no es par")
65 }
66
67 /*
68 ----ARCHIVO BASICO----
69 ----
70
71 El valor de val1 es: 214
72 El valor de val2 es: 412
73 El valor de val3 es: 1439
74 El resultado de la operación es: 1439
75 El valor de bol es: false
```

Normal text file | length: 2,232 | lines: 85 | Ln: 37 | Col: 2 | Pos: 1,008 | Windows (CR LF) | UTF-8 | INS

Estos archivos cuentan de 1 sola parte que puede verse dividida por entornos, es decir tenemos un entorno global en el cual podemos tener instrucciones como la declaración de variables, la declaración de funciones o la declaración de vectores y matrices, además de poder realizar ciclos condicionales o loops. Luego esta el entorno de las funciones que se creen donde se pueden tener las mismas instrucciones y características del entorno global. También se debe mencionar que tanto en el entorno de funciones como el entorno de ciclos condicionales o loops es posible el uso de sentencias de transferencia.

Interfaces

Instrucción

Esta clase abstracta es utilizada como base para todas las clases que sean de tipo instrucción esto quiere decir clases que interactúan con el programa pero que no necesariamente retornan algún tipo de valor o resultado en este caso se retorna un valor en forma de símbolo del sistema aunque este en la mayoría de casos no tiene mayor uso.

```
nterfaces > Instruction.go > Instruction
1 package interfaces
2
3 import "Proyecto1_OLC2_2S2023_202101648/Environment"
4
5 type Instruction interface {
6     Ejecutar(ast *environment.AST, env interface{}) environment.Symbol
7 }
```

Expresión

Esta clase abstracta es utilizada como base para todas las clases que sean de tipo Expresión, estas deben retornar un valor en forma de símbolo el cual nos permite obtener datos como el tipo de nuestra expresión o su valor entre otros.

```
package interfaces

import "Proyecto1_OLC2_2S2023_202101648/Environment"

type Expression interface {
    Ejecutar(ast *environment.AST, env interface{}) environment.Symbol
}
```

Environment

Esta clase es utilizada para generar los diferentes entornos en los que se ejecuta el programa iniciando por el entorno global y luego continuando a otros como el entorno local dentro de una función o el entorno dentro de una sentencia if, en esta clase también se tienen funciones como SaveVariable lo que nos permite guardar una variable en su entorno para su posterior uso en donde sea posible por su alcance, otras funciones que tenemos para el uso de las variables son GetVariable y SetVariable lo que nos permite obtener el valor de una variable en el caso de la primera función o darle un nuevo valor a la variable en caso de la segunda función . Otra función importante es SaveFuncion la cual nos permite guardar las diversas funciones o métodos que se crearan durante el programa, esta junto a la función GetFunción nos permite el manejo de tanto la declaración de una función como su posterior obtención. Por ultimo tenemos las funciones que nos permiten el manejo de structs las cuales son SaveStruct para poder guardar un struct y sus datos y GetStruct para poder obtener el struct y los datos que este debe tener.

```
Environment > -o Environment.go > NewEnvironment
1  package environment
2
3  import (
4      "fmt"
5      "strconv"
6  )
7
8  type Environment struct {
9      Anterior interface{}
10     Tabla    map[string]Symbol
11     Structs map[string]Symbol
12     Functions map[string]FunctionSymbol
13     Id       string
14 }
15
16 func NewEnvironment(anterior interface{},id string) Environment{
17     return Environment{
18         Anterior: anterior,
19         Tabla:    make(map[string]Symbol),
20         Structs:  make(map[string]Symbol),
21         Functions: make(map[string]FunctionSymbol),
22         Id: id,
23     }
24 }
```

```

func (env Environment) SaveVariable(id string,value Symbol,ast *AST) {
    var tipo = ""
    linea := strconv.Itoa(value.Lin)
    columna := strconv.Itoa(value.Col)
    if variable, ok := env.Tabla[id]; ok {
        ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Variable ya declarada " + id, Ambito: env.Id})
        fmt.Println("Variable ya declarada: ",variable)
        return
    }
    if value.Tipo == INTEGER {
        tipo = "Int"
    } else if value.Tipo == FLOAT {
        tipo = "Float"
    } else if value.Tipo == STRING {
        tipo = "String"
    } else if value.Tipo == BOOLEAN {
        tipo = "Bool"
    } else if value.Tipo == VECTOR {
        tipo = "Vector"
    } else if value.Tipo == STRUCT {
        tipo = "Struct"
    }
    ast.SetTablaSimbolos(SimbolTabla{Lin: linea, Col: columna, TipoSimbolo: "Variable", TipoDato: tipo, Ambito: env.Id,Id: id})
    env.Tabla[id] = value
}

```

```

func (env Environment) GetVariable(id string,ast *AST,linea string, columna string) Symbol {
    var tmpEnv Environment
    tmpEnv = env
    for {
        if variable, ok := tmpEnv.Tabla[id]; ok {
            return variable
        }
        if tmpEnv.Anterior == nil {
            break
        } else {
            tmpEnv = tmpEnv.Anterior.(Environment)
        }
    }
    fmt.Println("Variable no declarada: ",id)
    ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Variable no declarada " + id, Ambito: env.Id})
    return Symbol{Lin: 0, Col: 0, Tipo: NULL, Valor: 0}
}

```

```

func (env Environment) SetVariable(id string, value Symbol,ast *AST) Symbol {
    var tmpEnv Environment
    tmpEnv = env
    for {
        if variable, ok := tmpEnv.Tabla[id]; ok {
            if tmpEnv.Tabla[id].Mutable == true{
                if tmpEnv.Tabla[id].Tipo == value.Tipo {
                    tmpEnv.Tabla[id] = value
                    return variable
                } else {
                    fmt.Println("Tipo de dato incorrecto: ")
                    linea := strconv.Itoa(value.Lin)
                    columna := strconv.Itoa(value.Col)
                    ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Tipo de dato incorrecto" , Ambito: env.Id})
                    return Symbol{Lin: 0, Col: 0, Tipo: NULL, Valor: 0}
                }
            } else {
                fmt.Println("Variable no mutable: " , tmpEnv.Tabla[id].Valor)
                linea := strconv.Itoa(value.Lin)
                columna := strconv.Itoa(value.Col)
                ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Variable no mutable" , Ambito: env.Id})
                return Symbol{Lin: 0, Col: 0, Tipo: NULL, Valor: 0}
            }
        }
    }
}

```

```

3     }
4     if tmpEnv.Anterior == nil {
5         break
6     } else {
7         tmpEnv = tmpEnv.Anterior.(Environment)
8     }
9 }
10 fmt.Println("Variable no declarada: ",id)
11 linea := strconv.Itoa(value.Lin)
12 columna := strconv.Itoa(value.Col)
13 ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Variable no declarada" , Ambito: env.Id})
14 return Symbol{Lin: 0, Col: 0, Tipo: NULL, Valor: 0}
15 }

```

```

107 func (env Environment) SaveFunction(id string, value FunctionSymbol,ast *AST) {
108     var tipo = ""
109     linea := strconv.Itoa(value.Lin)
110     columna := strconv.Itoa(value.Col)
111     if variable, ok := env.Functions[id]; ok {
112         ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Funcion ya existe " + id, Ambito: env.Id})
113         fmt.Println("La funcion " + variable.Id + " ya existe")
114         return
115     }
116     if value.TipoRetorno == INTEGER {
117         tipo = "Int"
118     } else if value.TipoRetorno == FLOAT {
119         tipo = "Float"
120     } else if value.TipoRetorno == STRING {
121         tipo = "String"
122     } else if value.TipoRetorno == BOOLEAN {
123         tipo = "Bool"
124     } else if value.TipoRetorno == VECTOR {
125         tipo = "Vector"
126     }
127     ast.SetTablaSimbolos(SimbolTabla{Lin: linea, Col: columna, TipoSimbolo: "Funcion", TipoDato: tipo, Ambito: env.Id,Id: id})
128     env.Functions[id] = value
129 }

```

```

131 func (env Environment) GetFunction(id string, ast *AST, linea string, columna string) FunctionSymbol {
132     var tmpEnv Environment
133     tmpEnv = env
134     for {
135         if variable, ok := tmpEnv.Functions[id]; ok {
136             return variable
137         }
138         if tmpEnv.Anterior == nil {
139             break
140         } else {
141             tmpEnv = tmpEnv.Anterior.(Environment)
142         }
143     }
144     ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Funcion no existe " + id, Ambito: env.Id})
145     fmt.Println("La funcion ", id, " no existe")
146     return FunctionSymbol{TipoRetorno: NULL}
147 }
148
149 func (env Environment) SaveStruct(id string, list []interface{}), ast *AST, linea string, columna string) {
150     if _, ok := env.Structs[id]; ok {
151         ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Struct ya existe " + id, Ambito: env.Id})
152         //fmt.Println("El struct " + id + " ya existe")
153         return
154     }
155     env.Structs[id] = Symbol{Lin: 0, Col: 0, Tipo: STRUCT, Valor: list}
156     ast.SetTablaSimbolos(SimbolTabla{Lin: linea, Col: columna, TipoSimbolo: "Struct", TipoDato: "STRUCT", Ambito: env.Id, Id: id})
157 }
158
159 func (env Environment) GetStruct(id string, ast *AST, linea string, columna string) Symbol {
160
161     var tmpEnv Environment
162     tmpEnv = env
163
164     for {
165         if tmpStruct, ok := tmpEnv.Structs[id]; ok {
166             return tmpStruct
167         }
168         if tmpEnv.Anterior == nil {
169             break
170         } else {
171             tmpEnv = tmpEnv.Anterior.(Environment)
172         }
173     }
174
175     ast.SetErrors(ErrorS{Lin: linea, Col: columna, Descripcion: "Struct no existe " + id, Ambito: env.Id})
176     fmt.Println("El struct ", id, " no existe")
177     return Symbol{Lin: 0, Col: 0, Tipo: NULL, Valor: 0}
178 }

```

Symbol

Esta clase nos permite formar símbolos es decir un struct que tiene un valor un tipo, una línea y columna, si este es mutable, etc.

```

1  package environment
2
3  type Symbol struct {
4      Lin int
5      Col int
6      Tipo TipoExpresion
7      Valor interface{}
8      Mutable bool
9      BreakFlag bool
10     ContinueFlag bool
11     ReturnFlag bool
12     ArrayTipo TipoExpresion
13     VectorTipo TipoExpresion
14 }

```


TipoExpresion

Esta constante nos permite obtener el tipo que puede tener una variable o valor.

```
1 package environment
2
3 type TipoExpresion int
4
5 const (
6     INTEGER TipoExpresion = iota //0
7     FLOAT                       //1
8     STRING                     //2
9     BOOLEAN                    //3
10    NULL                       //4
11    ARRAY                      //5
12    DEPENDIENTE                //6
13    VECTOR                     //7
14    STRUCT                     //8
15 )
```

StructContent

Este struct nos permite guardar el contenido o parámetros de un struct.

```
Environment > ContenidoStruct.go > NewStructContent
1 package environment
2
3 type StructContent struct {
4     Id string
5     Exp interface{}
6 }
7
8 func NewStructContent(id string, ex interface{}) StructContent {
9     exp := StructContent{Id: id, Exp: ex}
10    return exp
11 }
```

TipoStruct

Este nos permite almacenar el tipo de un struct.

```
Environment > TipoStruct.go > NewStructType
1 package environment
2
3 type StructType struct {
4     Id string
5     Tipo TipoExpresion
6     StructID string
7 }
8
9 func NewStructType(id string, tip TipoExpresion, id2 string) StructType {
10    exp := StructType{id, tip, id2}
11    return exp
12 }
```

TipoArray

Struct usado para almacenar el tipo de dato que se almacenara en una matriz

```
Environment > TipoArray.go > TipoArray
1  package environment
2
3  type TipoArray struct {
4      Tipo TipoExpresion
5  }
6
7  func NewTipoArray(tipo TipoExpresion) TipoArray {
8      return TipoArray{tipo}
9  }
```

SimbolTabla

Este struct nos permite el manejo de los datos necesarios que se deben guardar en la tabla de símbolos.

```
Environment > TablaSimbol.go > SimbolTabla
1  package environment
2
3  type SimbolTabla struct {
4      Lin      string
5      Col      string
6      TipoSimbolo string
7      TipoDato  string
8      Ambito    string
9      Id        string
10 }
```

FunctionSymbol

Este struct nos permite almacenar los datos necesarios para manejar correctamente una función como por ejemplo la lista de parámetros, su bloque de instrucciones o su tipo de retorno.

```

Environment > FunctionSymbol.go > FunctionSymbol
1  package environment
2
3  type FunctionSymbol struct {
4      Lin      int
5      Col      int
6      Id       string
7      ListaParametros []interface{}
8      Bloque    []interface{}
9      TipoRetorno TipoExpresion
10 }

```

ErrorS

Este struct nos permite el manejo de los datos necesarios para el reporte de errores

```

Environment > ErrorS.go > ErrorS
1  package environment
2
3  type ErrorS struct {
4      Lin      string
5      Col      string
6      Descripcion string
7      Ambito   string
8  }

```

Structs que usan la interfaz de Expresión

Array

Este struct permitía la creación de una matriz ya sea de una o múltiples dimensiones.

```

expressions > Array.go > Array
1  package expressions
2
3  import(
4      "Proyecto1_OLC2_252023_202101648/Environment"
5      "Proyecto1_OLC2_252023_202101648/interfaces"
6  )
7
8  type Array struct {
9      Lin      int
10     Col      int
11     ListExp   []interface{}
12 }
13
14 func NewArray(lin int, col int, list []interface{}) Array {
15     exp := Array{lin,col,list}
16     return exp
17 }
18

```

```

18
19 func (p Array) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
20
21     var tempExp []interface{}
22
23     for _, s := range p.ListExp {
24         tempExp = append(tempExp, s.(interfaces.Expression).Ejecutar(ast, env))
25     }
26
27     return environment.Symbol{
28         Lin: p.Lin,
29         Col: p.Col,
30         Tipo: environment.ARRAY,
31         Valor: tempExp,
32         Mutable: true,
33     }
34 }

```

ArrayAccess

Este struct nos permite acceder a una posición específica ya sea de una matriz o de un vector de la forma ID[0]

```

1  package expressions
2
3  import(
4      "Proyecto1_OLC2_2S2023_202101648/Environment"
5      "Proyecto1_OLC2_2S2023_202101648/interfaces"
6      //"fmt"
7      "strconv"
8  )
9
10 type ArrayAccess struct {
11     Lin      int
12     Col      int
13     Array    interfaces.Expression
14     Index    interfaces.Expression
15 }
16
17 func NewArrayAccess(lin int, col int, array interfaces.Expression, index interfaces.Expression) ArrayAccess {
18     exp := ArrayAccess{lin, col, array, index}
19     return exp
20 }
21
22 func (p ArrayAccess) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
23
24     var tempArray environment.Symbol
25     tempArray = p.Array.Ejecutar(ast, env)
26
27     if tempArray.Tipo == environment.ARRAY || tempArray.Tipo == environment.VECTOR {
28         var tempIndex environment.Symbol
29         tempIndex = p.Index.Ejecutar(ast, env)
30         var tempValor interface{}
31         tempValor = tempArray.Valor
32
33         if tempIndex.Valor.(int) >= 0 && tempIndex.Valor.(int) < len(tempValor.([]interface{})) {
34             valorRetorno := tempValor.([]interface{})[tempIndex.Valor.(int)].(environment.Symbol)
35             return valorRetorno
36         } else {
37             linea := strconv.Itoa(p.Lin)
38             columna := strconv.Itoa(p.Col)
39             ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "El índice no es valido", Ambito: env.(environment.Environment).Id})
40             //fmt.Println("Indice: ", tempIndex.Valor.(int))
41             //fmt.Println("Tamaño: ", len(tempValor.([]interface{})))
42         }
43     } else {
44         linea := strconv.Itoa(p.Lin)
45         columna := strconv.Itoa(p.Col)
46         ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "La variable no es un arreglo", Ambito: env.(environment.Environment).Id})
47     }
48     return environment.Symbol{
49         Lin: p.Lin,
50         Col: p.Col,
51         Tipo: environment.NULL,
52         Valor: 0,
53         Mutable: true,
54     }
55 }

```

LlamadoFuncion

Este struct se encargaba de permitirnos realizar el llamado a una función que hubiera sido creada previamente para poder ejecutar las instrucciones que esta tiene, por lo que necesitamos la lista de parámetros si es que tuviera.

```
1 package expressions
2
3 import (
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "Proyecto1_OLC2_252023_202101648/instructions"
7     "strconv"
8 )
9
10 type LlamadoFuncion struct {
11     Lin      int
12     Col      int
13     Id       string
14     Parametros []interface{}
15 }
16
17 func NewLlamadoFuncion(lin int, col int, id string, parametros []interface{}) LlamadoFuncion {
18     return LlamadoFuncion{lin, col, id, parametros}
19 }
20
21 func (lf LlamadoFuncion) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
22     var resultado environment.Symbol
23     var function environment.FunctionSymbol
24     linea := strconv.Itoa(lf.Lin)
25     columna := strconv.Itoa(lf.Col)
26     function = env.(environment.Environment).GetFunction(lf.Id, ast, linea, columna)
27     var envFuncion environment.Environment
28     envFuncion = environment.NewEnvironment(lf.GetGlobal(env), lf.Id+"(Funcion)")
29
30     if len(lf.Parametros) == len(function.ListaParametros) {
31         for i:=0; i < len(lf.Parametros); i++ {
32             var param environment.Symbol
33             param = lf.Parametros[i].(interfaces.Expression).Ejecutar(ast, env)
34             if param.Tipo == function.ListaParametros[i].(instructions.DeclaracionParametros).Tipo {
35                 envFuncion.SaveVariable(function.ListaParametros[i].(instructions.DeclaracionParametros).Id, param, ast)
36             } else {
37                 ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "El tipo de dato del parámetro no coincide", Ambito: envFuncion.Id})
38                 return resultado
39             }
40         }
41     } else {
42         ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "La cantidad de parámetros no coincide", Ambito: envFuncion.Id})
43     }
44
45     for _, inst := range function.Bloque {
46         val := inst.(interfaces.Instruction).Ejecutar(ast, envFuncion)
47         if val.ReturnFlag == true {
48             if function.TipoRetorno == val.Tipo {
49                 return val
50             } else {
51                 ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "El tipo de dato del retorno no coincide", Ambito: envFuncion.Id})
52                 return environment.Symbol{Lin: lf.Lin, Col: lf.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
53             }
54         } else if val.BreakFlag == true || val.Valor == "break" {
55             return val
56         } else if val.ContinueFlag == true || val.Valor == "continue" {
57             return val
58         }
59     }
60     return resultado
61 }
62
63 func (lf LlamadoFuncion) GetGlobal(env interface{}) environment.Environment {
64     var tmpGlobal environment.Environment
65     tmpGlobal = env.(environment.Environment)
66     for {
67         if tmpGlobal.Id == "GLOBAL" {
68             return tmpGlobal
69         }
70         if tmpGlobal.Anterior == nil {
71             break
72         } else {
73             tmpGlobal = tmpGlobal.Anterior.(environment.Environment)
74         }
75     }
76     return tmpGlobal
77 }
```

Count

Este struct es utilizado para el manejo de la función propia de los struct count la cual se encarga de devolvernos como un entero la cantidad de datos dentro de un vector.

```
1 package expressions
2
3 import (
4     "Proyecto1_OLC2_2S2023_202101648/Environment"
5     //"fmt"
6     "strconv"
7 )
8
9 type Count struct {
10     Lin    int
11     Col    int
12     Id     string
13 }
14
15 func NewCount(lin int, col int, id string) Count {
16     exp := Count{lin,col,id}
17     return exp
18 }
19
20 func (p Count) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
21     var temporal environment.Symbol
22     linea := strconv.Itoa(p.Lin)
23     columna := strconv.Itoa(p.Col)
24     temporal = env.(environment.Environment).GetVariable(p.Id,ast,linea,columna)
25     if temporal.Tipo == environment.VECTOR {
26         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: len(temporal.Valor.([]interface{})), Mutable: true}
27     } else {
28         ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "La variable no es un vector", Ambito: env.(environment.Environment).Id})
29         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: 0, Mutable: true}
30     }
31 }
```

ForRange

Este struct se encarga de construir el rango que se utiliza en el ciclo ForIn cuando este es de la forma for i in 0...9 ; esto quiere decir que este struct se encarga de realizar un array que tenga los números del 0 al 9 para poder realizar los ciclos completos.

```
expressions > ForRange.go > (ForRange).Ejecutar
1 package expressions
2
3 import (
4     "Proyecto1_OLC2_2S2023_202101648/Environment"
5     "Proyecto1_OLC2_2S2023_202101648/interfaces"
6     "strconv"
7     //"fmt"
8 )
9
10 type ForRange struct {
11     Lin    int
12     Col    int
13     range1 interfaces.Expression
14     range2 interfaces.Expression
15 }
16
17 func NewForRange(lin int, col int, r1 interfaces.Expression, r2 interfaces.Expression) ForRange {
18     return ForRange{lin,col,r1,r2}
19 }
20
```

```

21 func (p ForRange) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
22     var rang1,rang2 environment.Symbol
23     var rango []interface{}
24
25     rang1 = p.rang1.Ejecutar(ast,env)
26     rang2 = p.rang2.Ejecutar(ast,env)
27
28     if (rang1.Valor.(int) < rang2.Valor.(int)+1) && (rang1.Tipo == environment.INTEGER) && (rang2.Tipo == environment.INTEGER) {
29         var tmpVal environment.Symbol
30         index := rang1.Valor.(int)
31         tmpVal = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: index, Mutable: true}
32         rango = append(rango,tmpVal)
33         for {
34             index++
35             if index < rang2.Valor.(int)+1 {
36                 tmpVal = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: index, Mutable: true}
37                 rango = append(rango,tmpVal)
38             } else {
39                 break
40             }
41         }
42     } else {
43         linea := strconv.Itoa(p.Lin)
44         columna := strconv.Itoa(p.Col)
45         ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "El rango no es valido", Ambito: "FOR"})
46     }
47     return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.ARRAY, Valor: rango, Mutable: true}
48 }

```

IsEmpty

Este struct se encarga de manejar la función propia de vectores isEmpty la cual por medio de un valor booleano ya sea true o false nos permite saber si un vector está vacío o no.

```

expressions > IsEmpty.go > (IsEmpty).Ejecutar
1 package expressions
2
3 import (
4     "Proyecto1_0LC2_252023_202101648/Environment"
5     //"fmt"
6     "strconv"
7 )
8
9 type IsEmpty struct {
10     Lin      int
11     Col      int
12     Id       string
13 }
14
15 func NewIsEmpty(lin int, col int, id string) IsEmpty {
16     exp := IsEmpty(lin,col,id)
17     return exp
18 }
19
20 func (p IsEmpty) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
21     var temporal environment.Symbol
22     linea := strconv.Itoa(p.Lin)
23     columna := strconv.Itoa(p.Col)
24     temporal = env.(environment.Environment).GetVariable(p.Id,ast,linea,columna)
25     if temporal.Tipo == environment.VECTOR {
26         arr:=temporal.Valor.([]interface{})
27         if len(arr)>0{
28             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.BOOLEAN, Valor: false, Mutable: true}
29         } else {
30             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.BOOLEAN, Valor: true, Mutable: true}
31         }
32     } else {
33         ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "La variable no es un vector", Ambito: env.(environment.Environment).Id})
34         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.BOOLEAN, Valor: false, Mutable: true}
35     }
36 }

```

LlamadoVar

Este struct se encarga de brindarnos la variable y su valor si es que en algún momento la llamamos para hacer una comparación o para imprimir su valor, por ejemplo.

```
expressions > LlamadoVar.go > (LlamadoVar).Ejecutar
1  package expressions
2
3  import(
4      "Proyecto1_OLC2_2S2023_202101648/Environment"
5      "strconv"
6  )
7
8  type LlamadoVar struct {
9      Lin      int
10     Col      int
11     Id       string
12 }
13
14 func NewLlamadoVar(lin int, col int, id string) LlamadoVar {
15     exp := LlamadoVar{lin,col,id}
16     return exp
17 }
18
19 func (p LlamadoVar) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
20     linea := strconv.Itoa(p.Lin)
21     columna := strconv.Itoa(p.Col)
22     resultado := env.(environment.Environment).GetVariable(p.Id,ast,linea,columna)
23     return resultado
24 }
```

Operación

Este struct se encarga del manejo de todas las posibles operaciones Aritméticas o Lógicas dentro de nuestro programa por medio de la obtención de los operadores izquierdo y derecho, también la obtención del operador de esta para saber qué tipo de operación se realizará.

expressions > Operation.go > (Operation).Ejecutar

```
1 package expressions
2
3 import(
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "fmt"
7     "strconv"
8 )
9
10 type Operacion struct {
11     Lin      int
12     Col      int
13     Operador_izq interfaces.Expression
14     Operador  string
15     Operador_der interfaces.Expression
16 }
17
18 func NewOperation(lin int, col int, op1 interfaces.Expression, operador string, op2 interfaces.Expression) Operacion {
19     exp := Operacion{Lin: lin, Col: col, Operador_izq: op1, Operador: operador, Operador_der: op2}
20     return exp
21 }
22
23 func (o Operacion) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
24     var dominante environment.TipoExpresion
25
26     tabla_dominante := [5][5]environment.TipoExpresion{
27         // INT      FLOAT      STRING      BOOLEAN      NULL
28         {environment.INTEGER, environment.FLOAT, environment.NULL, environment.NULL, environment.NULL},
29         //FLOAT
30         {environment.FLOAT, environment.FLOAT, environment.NULL, environment.NULL, environment.NULL},
31         //STRING
32         {environment.NULL, environment.NULL, environment.STRING, environment.NULL, environment.NULL},
33         //BOOLEAN
34         {environment.NULL, environment.NULL, environment.NULL, environment.BOOLEAN, environment.NULL},
35         //NULL
36         {environment.NULL, environment.NULL, environment.NULL, environment.NULL, environment.NULL},
37     }
38 }
```

expressions > Operation.go > (Operation).Ejecutar

```
38
39 var op1, op2 environment.Symbol
40 op1 = o.Operador_izq.Ejecutar(ast, env)
41 if o.Operador_der != nil {
42     op2 = o.Operador_der.Ejecutar(ast, env)
43 }
44 switch o.Operador {
45 case "+":
46     {
47         dominante = tabla_dominante[op1.Tipo][op2.Tipo]
48
49         if dominante == environment.INTEGER {
50             return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: op1.Valor.(int) + op2.Valor.(int), Mutable: true}
51         } else if dominante == environment.FLOAT {
52             val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
53             val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
54             return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: val1 + val2, Mutable: true}
55         } else if dominante == environment.STRING {
56             r1 := fmt.Sprintf("%v", op1.Valor)
57             r2 := fmt.Sprintf("%v", op2.Valor)
58             return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: r1 + r2, Mutable: true}
59         } else {
60             linea := strconv.Itoa(o.Lin)
61             columna := strconv.Itoa(o.Col)
62             ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "Error de tipos en la suma", Ambito: env.(environment.Environment)})
63         }
64     }
65 case "-":
66     {
67         dominante = tabla_dominante[op1.Tipo][op2.Tipo]
68
69         if dominante == environment.INTEGER {
70             return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: op1.Valor.(int) - op2.Valor.(int), Mutable: true}
71         } else if dominante == environment.FLOAT {
72             val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
73             val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
74             return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: val1 - val2, Mutable: true}
75         } else {
76             linea := strconv.Itoa(o.Lin)
77             columna := strconv.Itoa(o.Col)
78             ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "Error de tipos en la resta", Ambito: env.(environment.Environment)})
79         }
80     }
81 }
```

```

expressions > Operacion.go > (Operacion).Ejecutar
columna := strconv.Itoa(o.Col)
ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la resta", Ambito: env.(environment.Environment)
78
79
80
81 case "+":
82 {
83 dominante = tabla_dominante[op1.Tipo][op2.Tipo]
84 if dominante == environment.INTEGER {
85     return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: op1.Valor.(int) * op2.Valor.(int), Mutable: true}
86 } else if dominante == environment.FLOAT {
87     val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
88     val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
89     return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: val1 * val2, Mutable: true}
90 } else {
91     linea := strconv.Itoa(o.Lin)
92     columna := strconv.Itoa(o.Col)
93     ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la multiplicacion", Ambito: env.(environment.Environment)
94 }
95 case "/":
96 {
97 dominante = tabla_dominante[op1.Tipo][op2.Tipo]
98 if dominante == environment.INTEGER {
99     if op2.Valor.(int) != 0 {
100         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: op1.Valor.(int) / op2.Valor.(int), Mutable: true}
101     } else {
102         linea := strconv.Itoa(o.Lin)
103         columna := strconv.Itoa(o.Col)
104         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "No puede dividir entre cero", Ambito: env.(environment.Environment)
105     }
106 } else if dominante == environment.FLOAT {
107     val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
108     val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
109     if val2 != 0 {
110         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: val1 / val2, Mutable: true}
111     } else {
112         linea := strconv.Itoa(o.Lin)
113         columna := strconv.Itoa(o.Col)
114         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "No puede dividir entre cero", Ambito: env.(environment.Environment)
115     }
116 }

```

```

Operacion.go x Primitivo.go StructAcceso.go StructExpresion.go 2 toFloat.go 1 toInt.go 1 toString.go 1 Vector.go
expressions > Operacion.go > (Operacion).Ejecutar
columna := strconv.Itoa(o.Col)
ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "No puede dividir entre cero", Ambito: env.(environment.Environment)
114
115
116 }
117 } else {
118     linea := strconv.Itoa(o.Lin)
119     columna := strconv.Itoa(o.Col)
120     ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la division", Ambito: env.(environment.Environment)
121 }
122 }
123 case "%":
124 {
125 dominante = tabla_dominante[op1.Tipo][op2.Tipo]
126 if dominante == environment.INTEGER {
127     if op2.Valor.(int) != 0 {
128         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: dominante, Valor: op1.Valor.(int) % op2.Valor.(int), Mutable: true}
129     } else {
130         linea := strconv.Itoa(o.Lin)
131         columna := strconv.Itoa(o.Col)
132         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "No puede dividir entre cero", Ambito: env.(environment.Environment)
133     } else {
134         linea := strconv.Itoa(o.Lin)
135         columna := strconv.Itoa(o.Col)
136         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en el modulo", Ambito: env.(environment.Environment)
137     }
138 }
139 case "UNARIO":
140 {
141     if op1.Tipo == environment.INTEGER {
142         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.INTEGER, Valor: -op1.Valor.(int), Mutable: true}
143     } else if op1.Tipo == environment.FLOAT {
144         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
145         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.FLOAT, Valor: -val1, Mutable: true}
146     } else {
147         linea := strconv.Itoa(o.Lin)
148         columna := strconv.Itoa(o.Col)
149         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en el unario", Ambito: env.(environment.Environment)
150     }
151 }
152 case "<":
153 {

```

```

expressions > Operacion.go > (Operacion).Ejecutar
151 }
152 case "<":
153 {
154     dominante = tabla_dominante[op1.Tipo][op2.Tipo]
155     if dominante == environment.INTEGER {
156         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(int) < op2.Valor.(int), Mutable: true)
157     } else if dominante == environment.FLOAT {
158         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
159         val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
160         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: val1 < val2, Mutable: true)
161     } else {
162         linea := strconv.Itoa(o.Lin)
163         columna := strconv.Itoa(o.Col)
164         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion menor que", Ambito: env.(envi
165     }
166 }
167 case ">":
168 {
169     dominante = tabla_dominante[op1.Tipo][op2.Tipo]
170     if dominante == environment.INTEGER {
171         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(int) > op2.Valor.(int), Mutable: true)
172     } else if dominante == environment.FLOAT {
173         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
174         val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
175         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: val1 > val2, Mutable: true)
176     } else {
177         linea := strconv.Itoa(o.Lin)
178         columna := strconv.Itoa(o.Col)
179         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion mayor que", Ambito: env.(envi
180     }
181 }
182 case "<=":
183 {
184     dominante = tabla_dominante[op1.Tipo][op2.Tipo]
185     if dominante == environment.INTEGER {
186         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(int) <= op2.Valor.(int), Mutable: true)
187     } else if dominante == environment.FLOAT {
188         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
189         val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
190         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: val1 <= val2, Mutable: true)
191     } else {
192         linea := strconv.Itoa(o.Lin)
193         columna := strconv.Itoa(o.Col)
194         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion menor igual que", Ambito: env.(envi
195     }
196 }
197 case ">=":
198 {
199     dominante = tabla_dominante[op1.Tipo][op2.Tipo]
200     if dominante == environment.INTEGER {
201         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(int) >= op2.Valor.(int), Mutable: true)
202     } else if dominante == environment.FLOAT {
203         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
204         val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
205         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: val1 >= val2, Mutable: true)
206     } else {
207         linea := strconv.Itoa(o.Lin)
208         columna := strconv.Itoa(o.Col)
209         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion mayor igual que", Ambito: env.(envi
210     }
211 }
212 case "==":
213 {
214     if op1.Tipo == op2.Tipo {
215         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor == op2.Valor, Mutable: true)
216     } else {
217         linea := strconv.Itoa(o.Lin)
218         columna := strconv.Itoa(o.Col)
219         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion igual que", Ambito: env.(envi
220     }
221 }
222 case "!=":
223 {
224     if op1.Tipo == op2.Tipo {
225         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor != op2.Valor, Mutable: true)
226     } else {
227         linea := strconv.Itoa(o.Lin)
228         columna := strconv.Itoa(o.Col)
229         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion diferente que", Ambito: env.(

```

```

expressions > Operacion.go > (Operacion).Ejecutar
191     } else {
192         linea := strconv.Itoa(o.Lin)
193         columna := strconv.Itoa(o.Col)
194         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion menor igual que", Ambito: env
195     }
196 }
197 case ">=":
198 {
199     dominante = tabla_dominante[op1.Tipo][op2.Tipo]
200     if dominante == environment.INTEGER {
201         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(int) >= op2.Valor.(int), Mutable: true)
202     } else if dominante == environment.FLOAT {
203         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", op1.Valor), 64)
204         val2, _ := strconv.ParseFloat(fmt.Sprintf("%v", op2.Valor), 64)
205         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: val1 >= val2, Mutable: true)
206     } else {
207         linea := strconv.Itoa(o.Lin)
208         columna := strconv.Itoa(o.Col)
209         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion mayor igual que", Ambito: env
210     }
211 }
212 case "==":
213 {
214     if op1.Tipo == op2.Tipo {
215         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor == op2.Valor, Mutable: true)
216     } else {
217         linea := strconv.Itoa(o.Lin)
218         columna := strconv.Itoa(o.Col)
219         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion igual que", Ambito: env.(envi
220     }
221 }
222 case "!=":
223 {
224     if op1.Tipo == op2.Tipo {
225         return environment.Symbol(Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor != op2.Valor, Mutable: true)
226     } else {
227         linea := strconv.Itoa(o.Lin)
228         columna := strconv.Itoa(o.Col)
229         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion diferente que", Ambito: env.(

```

```

expressions > Operacion.go > (Operacion).Ejecutar
230 }
231 }
232 case "&&":
233 {
234     if(op1.Tipo == environment.BOOLEAN && op2.Tipo == environment.BOOLEAN){
235         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(bool) && op2.Valor.(bool),Mutable: true}
236     } else {
237         linea := strconv.Itoa(o.Lin)
238         columna := strconv.Itoa(o.Col)
239         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion AND", Ambito: env.(environment)
240     }
241 }
242 case "||":
243 {
244     if(op1.Tipo == environment.BOOLEAN && op2.Tipo == environment.BOOLEAN){
245         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: op1.Valor.(bool) || op2.Valor.(bool),Mutable: true}
246     } else {
247         linea := strconv.Itoa(o.Lin)
248         columna := strconv.Itoa(o.Col)
249         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion OR", Ambito: env.(environment)
250     }
251 }
252 case "!":
253 {
254     if(op1.Tipo == environment.BOOLEAN){
255         return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.BOOLEAN, Valor: !op1.Valor.(bool),Mutable: true}
256     } else {
257         linea := strconv.Itoa(o.Lin)
258         columna := strconv.Itoa(o.Col)
259         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error de tipos en la comparacion NOT", Ambito: env.(environment)
260     }
261 }
262 }
263
264 var result interface{}
265 return environment.Symbol{Lin: o.Lin, Col: o.Col, Tipo: environment.NULL, Valor: result,Mutable: true}
266 }

```

Primitivo

Esta struct se encarga de brindarnos el valor Primitivo esto quiere decir un valor Integer o Float, entre otros.

```

expressions > Primitivo.go > (Primitivo).Ejecutar
1  package expressions
2
3  import(
4      "Proyecto1_OLC2_252023_202101648/Environment"
5  )
6
7  type Primitivo struct {
8      Lin    int
9      Col    int
10     Valor  interface{}
11     Tipo   environment.TipoExpresion
12 }
13
14 func NewPrimitive(lin int, col int, valor interface{}, tipo environment.TipoExpresion) Primitivo {
15     exp := Primitivo{Lin: lin, Col: col, Valor: valor, Tipo: tipo}
16     return exp
17 }
18
19 func (p Primitivo) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
20     return environment.Symbol{
21         Lin: p.Lin,
22         Col: p.Col,
23         Tipo: p.Tipo,
24         Valor: p.Valor,
25         Mutable: true,
26     }
27 }

```

StructAccess

Este struct se encarga de brindarnos acceso a los datos guardados como parámetros en un struct

```
expressions > == StructAccess.go > (StructAccess).Ejecutar
1 package expressions
2
3 import (
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "strconv"
7 )
8
9 type StructAccess struct {
10     Lin    int
11     Col    int
12     Struct interfaces.Expression
13     Id     string
14 }
15
16 func NewStructAccess(lin int, col int, str interfaces.Expression, id string) StructAccess {
17     exp := StructAccess{lin, col, str, id}
18     return exp
19 }
20
21 func (p StructAccess) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
22     var result, tempStruct environment.Symbol
23     tempStruct = p.Struct.Ejecutar(ast, env) //obtengo el struct
24
25     if tempStruct.Tipo == environment.STRUCT {
26         if variable, ok := tempStruct.Valor.(map[string]environment.Symbol)[p.Id]; ok {
27             return variable
28         }
29         linea := strconv.Itoa(p.Lin)
30         columna := strconv.Itoa(p.Col)
31         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no existe en el struct", Ambito: env.(environment.Environment)})
32         return result
33     }
34     linea := strconv.Itoa(p.Lin)
35     columna := strconv.Itoa(p.Col)
36     ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no es un struct", Ambito: env.(environment.Environment).Id})
37     return result
38 }
39
```

StructExpression

Este struct se encarga de brindarnos la posibilidad de realizar el struct como expresión.

```
1 package expressions
2
3 import (
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "strconv"
7 )
8
9 type StructExp struct {
10     Lin    int
11     Col    int
12     Id     string
13     ListExp []interface{}
14 }
15
16 func NewStructExp(lin int, col int, id string, list []interface{}) StructExp {
17     exp := StructExp{lin, col, id, list}
18     return exp
19 }
20
21 func (p StructExp) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
22     var valor map[string]environment.Symbol
23     var result environment.Symbol
24     var tempExp []interface{}
25     //se guarda el listado de valores nuevos
26     for s := range p.ListExp {
27         tempExp = append(tempExp, s)
28     }
29     //se obtiene la estructura guardada
30     linea := strconv.Itoa(p.Lin)
31     columna := strconv.Itoa(p.Col)
32     resultStruct := env.(environment.Environment).GetStruct(p.Id, ast, linea, columna)
33
34     //se valida si existe struct
35     if resultStruct.Tipo == environment.STRUCT {
36         //Validar tamaño y que cada tipo coincida con el struct existente
37         if len(resultStruct.Valor.([]interface{})) == len(p.ListExp) {
38             valor = make(map[string]environment.Symbol)
39         }
40     }
41 }
42
```

```

34 //se valida si existe struct
35 if resultStruct.Tipo == environment.STRUCT {
36     //Validar tamaño y que cada tipo coincida con el struct existente
37     if len(resultStruct.Valor.([]interface{})) == len(p.ListExp) {
38         valor = make(map[string]environment.Symbol)
39         //recorrer el struct almacenado
40         for i := 0; i < len(resultStruct.Valor.([]interface{})); i++ {
41             //validar los identificadores
42             if resultStruct.Valor.([]interface{})[i].(environment.StructType).Id == p.ListExp[i].(environment.StructContent).Id {
43                 tempVal := p.ListExp[i].(environment.StructContent).Exp.(interfaces.Expression).Ejecutar(ast, env)
44                 //validando tipos
45                 if resultStruct.Valor.([]interface{})[i].(environment.StructType).Tipo == tempVal.Tipo {
46                     valor[resultStruct.Valor.([]interface{})[i].(environment.StructType).Id] = tempVal
47                 } else {
48                     ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "El tipo de dato no es valido", Ambito: env.(environment.Environment).Id})
49                 }
50             }
51         }
52         result = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.STRUCT, Valor: valor}
53     } else {
54         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "El tamaño del struct no es valido", Ambito: env.(environment.Environment).Id})
55     }
56 } else {
57     ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no es un struct", Ambito: env.(environment.Environment).Id})
58 }
59 return result
60 }

```

ToFloat

Esta estructura se encarga de pasar el tipo de valores String a Float siempre y cuando tengan el formato correcto.

```

expressions > toFloat.go > (toFloat).Ejecutar
1  package expressions
2
3  import(
4      "Proyecto1_OLC2_252023_202101648/Environment"
5      "Proyecto1_OLC2_252023_202101648/interfaces"
6      "fmt"
7      "strconv"
8      // "math"
9      "strings"
10 )
11
12 type toFloat struct {
13     Lin      int
14     Col      int
15     Expresion interfaces.Expression
16 }
17
18 func NewToFloat(lin int, col int, exp interfaces.Expression) toFloat {
19     return toFloat{lin,col,exp}
20 }
21

```

```

22 func (p toFloat) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
23     var exp environment.Symbol
24     exp = p.Expression.Ejecutar(ast, env)
25
26     if exp.Tipo == environment.STRING {
27         numero := fmt.Sprintf("%v", exp.Value)
28         if strings.Contains(numero, ".") {
29             num, err := strconv.ParseFloat(numero, 64);
30             if err != nil {
31                 fmt.Println(err)
32                 return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
33             }
34             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.FLOAT, Valor: num, Mutable: true}
35         } else if !strings.Contains(numero, ".") {
36             num, err1 := strconv.ParseFloat(numero, 64);
37             num2 := strconv.FormatFloat(num, 'f', 2, 64)
38             num3, err2 := strconv.ParseFloat(num2, 64);
39             //fmt.Printf("num2: %f\n", num3)
40             if err1 != nil && err2 != nil {
41                 fmt.Println(err1)
42                 fmt.Println(err2)
43                 return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
44             }
45             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.FLOAT, Valor: num3, Mutable: true}
46         }
47     } else {
48         linea := strconv.Itoa(p.Lin)
49         columna := strconv.Itoa(p.Col)
50         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no es un string", Ambito: env.(environment.Environment).Id})
51         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
52     }
53     return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
54 }
55

```

ToInt

Esta estructura se encarga de pasar el tipo de valores String a Int siempre y cuando tengan el formato correcto o de Float a String.

```

expressions > ToInt.go > (toInt).Ejecutar
1 package expressions
2
3 import(
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "fmt"
7     "strconv"
8     "math"
9     "strings"
10 )
11
12 type toInt struct {
13     Lin      int
14     Col      int
15     Expresion interfaces.Expression
16 }
17
18 func NewToInt(lin int, col int, exp interfaces.Expression) toInt {
19     return toInt{lin, col, exp}
20 }
21

```

```

22 func (p toInt) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
23     var exp environment.Symbol
24     exp = p.Expression.Ejecutar(ast, env)
25
26     if exp.Tipo == environment.STRING {
27         numero := fmt.Sprintf("%v", exp.Valor)
28         if (strings.Contains(numero, ".")) {
29             num, err := strconv.ParseFloat(numero, 64);
30             if err != nil {
31                 fmt.Println(err)
32             }
33             valor := math.Trunc(num)
34             valorInt := int(valor)
35             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: valorInt, Mutable: true}
36         } else {
37             valor, _ := strconv.Atoi(numero)
38             return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: valor, Mutable: true}
39         }
40     } else if exp.Tipo == environment.FLOAT {
41         val1, _ := strconv.ParseFloat(fmt.Sprintf("%v", exp.Valor), 64)
42         //fmt.Print("valor:", val1)
43         valor := math.Trunc(val1)
44         //fmt.Print("valor:", valor)
45         valorInt := int(valor)
46         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.INTEGER, Valor: valorInt, Mutable: true}
47     } else {
48         linea := strconv.Itoa(p.Lin)
49         columna := strconv.Itoa(p.Col)
50         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no pudo ser convertida a int", Ambito: env.(environment.Environment)})
51         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
52     }
53 }

```

ToString

Esta estructura se encarga de pasar el tipo de valores Int a String, Float a String o Bool a String.

```

expressions > toString.go > (toString).Ejecutar
1 package expressions
2
3 import(
4     "Proyecto1_OLC2_2S2023_202101648/Environment"
5     "Proyecto1_OLC2_2S2023_202101648/interfaces"
6     "fmt"
7     "strconv"
8     // "math"
9     //"strings"
10 )
11
12 type toString struct {
13     Lin      int
14     Col      int
15     Expression interfaces.Expression
16 }
17
18 func NewToString(lin int, col int, exp interfaces.Expression) toString {
19     return toString{lin, col, exp}
20 }
21

```



```

22 func (p toString) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
23     var exp environment.Symbol
24     exp = p.Expression.Ejecutar(ast,env)
25
26     if exp.Tipo == environment.INTEGER{
27         valor := strconv.Itoa(exp.Valor.(int))
28         exp.Valor = valor
29         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.STRING, Valor: exp.Valor, Mutable: true}
30     } else if exp.Tipo == environment.FLOAT {
31         valor := strconv.FormatFloat(exp.Valor.(float64), 'f', 3, 64)
32         exp.Valor = valor
33         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.STRING, Valor: exp.Valor, Mutable: true}
34     } else if exp.Tipo == environment.BOOLEAN {
35         valor := fmt.Sprintf("%v", exp.Valor)
36         exp.Valor = valor
37         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.STRING, Valor: exp.Valor, Mutable: true}
38     } else {
39         linea := strconv.Itoa(p.Lin)
40         columna := strconv.Itoa(p.Col)
41         ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "La variable no puede convertirse a string", Ambito: env.(environment.Environment).Id})
42         return environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.NULL, Valor: nil, Mutable: true}
43     }
44 }

```

Vector

Esta struct se encarga de manejar y crear nuestro vector de datos.

```

expressions > go Vector.go > NewVector
1 package expressions
2
3 import(
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6     "strconv"
7 )
8
9 type Vector struct {
10     Lin      int
11     Col      int
12     ListExp  []interface{}
13 }
14
15 func NewVector(lin int, col int, list []interface{}) Vector {
16     exp := Vector{lin,col,list}
17     return exp
18 }
19
20 func (p Vector) Ejecutar(ast *environment.AST,env interface{}) environment.Symbol {
21     var retorno environment.Symbol
22     var tempExp []interface{}
23
24     if len(p.ListExp) > 0 {
25         tempTipo := p.ListExp[0].(interfaces.Expression).Ejecutar(ast,env).Tipo
26         tempVectorTipo environment.TipoExpresion
27         for _, s := range p.ListExp {
28             simboloTmp := s.(interfaces.Expression).Ejecutar(ast,env)
29             if simboloTmp.Tipo == tempTipo {
30                 tempExp = append(tempExp,simboloTmp)
31                 tempVectorTipo = tempTipo
32             } else {
33                 linea := strconv.Itoa(p.Lin)
34                 columna := strconv.Itoa(p.Col)
35                 ast.SetErrors(environment.Errors{Lin: linea, Col: columna, Descripcion: "Error en tipos", Ambito: env.(environment.Environment).Id})
36             }
37         }
38         retorno = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.VECTOR, Valor: tempExp, Mutable: true, VectorTipo: tempVectorTipo}
39     } else {
40         retorno = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.VECTOR, Valor: tempExp, Mutable: true}
41     }
42     return retorno
43 }
44

```

Clases extendidas de Instrucción

Return

Esta struct nos permite manejar la sentencia return.

```
20 func (p Vector) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
21     var retorno environment.Symbol
22     var tempExp []interface{}
23
24     if len(p.ListExp) > 0 {
25         tempTipo := p.ListExp[0].(interfaces.Expression).Ejecutar(ast, env).Tipo
26         var tempVectorTipo environment.TipoExpresion
27         for _, s := range p.ListExp {
28             simboloTmp := s.(interfaces.Expression).Ejecutar(ast, env)
29             if simboloTmp.Tipo == tempTipo {
30                 tempExp = append(tempExp, simboloTmp)
31                 tempVectorTipo = tempTipo
32             } else {
33                 linea := strconv.Itoa(p.Lin)
34                 columna := strconv.Itoa(p.Col)
35                 ast.SetErrors(environment.ErrorS{Lin: linea, Col: columna, Descripcion: "Error en tipos", Ambito: env.(environment.Environment).Id})
36             }
37         }
38         retorno = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.VECTOR, Valor: tempExp, Mutable: true, VectorTipo: tempVectorTipo}
39     } else {
40         retorno = environment.Symbol{Lin: p.Lin, Col: p.Col, Tipo: environment.VECTOR, Valor: tempExp, Mutable: true}
41     }
42     return retorno
43 }
```

AsignarValor

Este struct nos permitía manejar el asignar un nuevo valor a una variable existente siempre y cuando el tipo de valor a asignar fuera igual al de la variable que se tenía previamente.

```
Instructions > Asignacion.go > (Asignacion).Ejecutar
1 package instructions
2
3 import(
4     "Proyecto1_OLC2_252023_202101648/Environment"
5     "Proyecto1_OLC2_252023_202101648/interfaces"
6 )
7
8 type Asignacion struct {
9     Lin      int
10    Col       int
11    Id        string
12    Expresion interfaces.Expression
13 }
14
15 func NewAsignacion(lin int, col int, id string, val interfaces.Expression) Asignacion{
16     instr := Asignacion{lin, col, id, val}
17     return instr
18 }
19
20 func (va Asignacion) Ejecutar(ast *environment.AST, env interface{}) environment.Symbol {
21     var result environment.Symbol
22     result = va.Expresion.Ejecutar(ast, env)
23     env.(environment.Environment).SetVariable(va.Id, result, ast)
24     return result
25 }
```

Reportes Generados

El ejecutar nuestro programa nos permite obtener 3 reportes diferentes para el usuario:

- Tabla de Símbolos
- Reporte de Errores

Cada reporte se genera en el momento en el que el usuario usa el botón de Ejecutar.

Tabla de Simbolos

Para la tabla de símbolos se utiliza un arreglo al cual se le añaden todas las variables, funciones y métodos que se están guardando por medio de las funciones que guardan dichas dentro del Environment, en ese espacio se envia datos como el tipo, si es una variable función o struct, su entorno es decir es local o es global y también la línea y columna donde se encuentra.

Ejemplo Tabla de Simbolos:

ID	Tipo Simbolo	Tipo Dato	Ambito	Linea	Columna
a	Variable	Int	GLOBAL	1	8
aux	Variable	Int	GLOBAL	19	10
index	Variable	Int	GLOBAL	53	12
i	Variable	Int	FOR	111	9
output	Variable	String	FOR	112	17
_	Variable	Int	FOR	113	13

Tabla de Errores:

Para la creación del reporte de errores se utiliza un struct para manejar errores sintácticos y semánticos.

Ejemplo Reporte Errores

Editor

```
1 var a = 909
2
3 print("=====
4 print("=====IF=====
5 print("=====
6
7 if a > 50 {
8     print("IF CORRECTO")
9 } else if a == 56 {
```

Consola

```
1 Error en la linea 121 y columna 0 El tipo de dato no es
2 =====
3 =====IF=====
4 =====
5 IF CORRECTO
6
7 =====
8 =====IFs ANIDADOS=====
9 =====
```

Abrir

Este metodo se usaban para el manejo del archivo, esto por medio en todos los casos de el buscador de archivos que react brinda.

```
const handleFile = () => {
    const file = document.getElementById("file").files[0];
    if(file){
        const reader = new FileReader();
        reader.onload = function (e) {
            const contents = e.target.result;
            setEditor(contents);
            setCodigoEditor(contents);
        };
        reader.readAsText(file);
    }
}
```

Interfaz Utilizada

La interfaz se creó por medio del uso del framework React, el cual utiliza el lenguaje de programación JavaScript para ir generando diferentes módulos los cuales tienen un parecido bastante alto a HTML que son utilizados para ir generando cada una de las piezas que conformarán finalmente lo que es la interfaz de usuario. La interfaz corre en el localhost:3000



Servidor

El servidor se creó por medio de Golang con Fiber un servidor local que en este caso es localhost:3002 y se tiene un solo endpoint que es localhost:3002/Interpreter lo que permite realizar todo el proceso de análisis y ejecución del programa.

Recibimiento de datos en el Frontend con axios

```
const interpretar = async () => {
  console.log("ejecutando");
  try {
    setConsola("Ejecutando...");
    if (editor === "") {
      setConsola("No hay nada que ejecutar");
      console.log("No hay código a interpretar");
    } else {
      console.log(editor);
      const response = await axios.post('http://localhost:5000/interprete/interpretar', { codigo: editor });
      console.log(response.data);
      const { consola, errores, ast, tablaSimbolos } = response.data;
      console.log(consola);
      console.log("ast", ast);
      console.log("tablaS", tablaSimbolos);
      console.log("errores", errores);
      setDot(ast);
      setDot2(tablaSimbolos);
      setDot3(errores);
      setConsola(consola);
    }
  } catch (error) {
    console.log(error);
    setConsola("Error en Servidor");
  }
}
```