Manual técnico

Cristian Daniel Gomez Escobar
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

ENVIROMENT

Aquí será el lugar donde tendremos las funciones necesarias para manejar la tabla de funciones y valores que manejara el programa.

FUNCION SaveFuncion

```
func (env Environment) SaveFuncion(id string, symb interface{}, Line int,
Column int /*, tipo interfaces.TipoExpresion, isMut bool, nameentorno
string, tipos *arrayList.List*/) {
    if _, ok := env.TablaFunciones[id]; ok {

        NewError("La función '"+id+"' ya declarada en entorno "+env.Nombre,
env.Nombre, Line, Column)
        return
    }
    env.TablaFunciones[id] = symb

    NewSymbol(id, "FUNCIÓN", "Función", env.Nombre, Line, Column)
}

func (env Environment) ExistFunction(id string) bool {

    var tmpEnv Environment
    tmpEnv = env

    for {
        if _, ok := tmpEnv.TablaFunciones[id]; ok {
            return true
        }

        if tmpEnv.father == nil {
            break
        } else {
            tmpEnv = tmpEnv.father.(Environment)
        }
    }
    return false
}
```

Con esta función se podrán guardar las funciones para la tabla de símbolos que se utillizaran para el manejo del programa.

ERRORLIST

Aquí se almacenaran los errores que tendrá el programa

```
func NewError(descripcion string, entorno string, line int, column int) {
    ahora := time.Now()
    //fechastr := fmt.Sprintf("%v", ahora.Day()) + "/" + fmt.Sprintf("%v",
ahora.Month()) + "/" + fmt.Sprintf("%v", ahora.Year())
    fechastr := ahora.Format("02/01/2023 13:01:09")
    err := Error{
        Descripcion: descripcion, Entorno: entorno,
        Line: line, Column: column, Fecha: fechastr,
    }

    ErrorList = append(ErrorList, err)

    printcon := fmt.Sprintf("ERROR: %s en (%d, %d)", err.Descripcion,
err.Line, err.Column) + "\n"
    Console += printcon
}
```

EXPRESIONES

VECTORES

Estos se crearan utilizando arraylist y se evaluar que el tipo del vector sea el correcto

```
    if p.TipoDec == 1 {
        tempType =
p.ListExp.GetValue(0).(interfaces.Expresion).EjecutarValor(env).Tipo

        //fmt.Println("     tempType: ", tempType)
        for _, s := range p.ListExp.ToArray() {
            valsym := s.(interfaces.Expresion).EjecutarValor(env)
            //  fmt.Println("--     valsym: ", valsym)
            if valsym.Tipo == tempType {
                //  fmt.Println("--     valsym.Valor: ", valsym.Valor)
                tempExp.Add(valsym)
                //  fmt.Println("--     tempExp: ", tempExp)
            } else {
                //fmt.Println("Error en el tipo del vector")
                desc := fmt.Sprintf("se esperaba '%v' se tiene '%v'",
interfaces.GetType(tempType), interfaces.GetType(valsym.Tipo))
```

```
                err.NewError("Vector incorrecta "+desc,
env.(environment.Environment).Nombre, p.Line, p.Column)

            }

        }

        /*vector formato tipo;tam*/
    }
```

Este una vez evaluado se podrá agregar a la tabla de símbolos en donde se podrá acceder a ellos al ejecutar la llamada

CADENAFOR

Esta clase permitirá recorrer una cadena para poder imprimir cada una de las letras de esta.

```
    array = arrayList.New()
    val1 = p.Exp_ini.EjecutarValor(env)
    p.TipoDec = val1.Tipo

    //print(val1.Valor.(string))
    if val1.Tipo == interfaces.STRING || val1.Tipo == interfaces.STR {

        var tmpSym interfaces.Symbol
        inival := val1.Valor.(string)
        //i:= len(inival)

        for _, c := range inival {
            car := string(c)
            tmpSym = interfaces.Symbol{
                Line:   p.Line,
                Column: p.Column,
                Id:     "",
                Tipo:   interfaces.STRING,
                Valor:  car,
                IsMut:  true,
            }
            array.Add(tmpSym)
        }
        //  fmt.Println(array)

    }
```

Aquí se podrá detectar si se coincide los tipos para poder recorrer la cadena y poder almacenarla en la tabla de símbolos como un string.

COUNT

```
if retornoExp.Tipo == interfaces.ARRAY || retornoExp.Tipo ==
interfaces.VECTOR {

        //fmt.Println("---      reflect.TypeOf(retornoExp)..Valor",
reflect.TypeOf(retornoExp.Valor))nterfaces.Symbol{Id: "", Tipo:
interfaces.INTEGER, Valor: retornoExp.Valor.(*arrayList.List).Len()}
        return interfaces.Symbol{Id: "", Tipo: interfaces.INTEGER, Valor:
retornoExp.Valor.(*arrayList.List).Len()}

    }
```

Aquí se puede retornar el valor de la cantidad de elementos en la lista en la que se le indique con el comando y se podrá almacenar en la tabla de símbolos.

OPERACIÓN

```
var res_dominante = [8][8]interfaces.TipoExpresion{
    //INTEGER           //FLOAT            //STRING         //STR
//BOOLEAN        //ARRAY              //VOID          //NULL
    //INTEGER
    {interfaces.INTEGER, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //FLOAT
    {interfaces.NULL, interfaces.FLOAT, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //STRING
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.STRING,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //STR
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.STR,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //BOOLEAN
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //ARRAY
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //VOID
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.VOID, interfaces.NULL},
    //NULL
```

```
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
}

var res_dominante_unario = [8][8]interfaces.TipoExpresion{
    //INTEGER           //FLOAT            //STRING         //STR
//BOOLEAN        //ARRAY              //VOID          //NULL
    //INTEGER
    {interfaces.INTEGER, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //FLOAT
    {interfaces.NULL, interfaces.FLOAT, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //STRING
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.STRING,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //STR
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.STR,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //BOOLEAN
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //ARRAY
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.NULL, interfaces.NULL},
    //VOID
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.BOOLEAN, interfaces.NULL, interfaces.VOID, interfaces.NULL},
    //NULL
    {interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL,
interfaces.NULL, interfaces.NULL, interfaces.NULL, interfaces.NULL},
}
```

Con estas matrices se podrán retornar el valor de cada uno de los valores que posee el programa para poder manejarlo en las operaciones que se podrán manejar en la tabla de símbolos.

Operación de resta

```
            if p.Unario {

                if retornoIzq.Tipo == interfaces.INTEGER {
                    return interfaces.Symbol{Id: "", Tipo: retornoIzq.Tipo,
Valor: -1 * retornoIzq.Valor.(int)}
                } else if retornoIzq.Tipo == interfaces.FLOAT {
                    return interfaces.Symbol{Id: "", Tipo: retornoIzq.Tipo,
Valor: -1 * retornoIzq.Valor.(float64)}
                }

            } else {
                dominante = res_dominante[retornoIzq.Tipo][retornoDer.Tipo]

                if dominante == interfaces.INTEGER {

                    return interfaces.Symbol{Id: "", Tipo: dominante, Valor:
retornoIzq.Valor.(int) - retornoDer.Valor.(int)}

                } else if dominante == interfaces.FLOAT {
                    val1, _ := strconv.ParseFloat(fmt.Sprintf("%v",
retornoIzq.Valor), 64)
                    val2, _ := strconv.ParseFloat(fmt.Sprintf("%v",
retornoDer.Valor), 64)
                    return interfaces.Symbol{Id: "", Tipo: dominante, Valor:
val1 - val2}

                } else {
                    desc := fmt.Sprintf("'%v' con '%v'",
interfaces.GetType(retornoIzq.Tipo), interfaces.GetType(retornoDer.Tipo))
                    err.NewError("Tipos incompatibles en resta "+desc,
env.(environment.Environment).Nombre, p.Line, p.Column)
                    //fmt.Print("ERROR: No es posible restar")
                }
            }
```

En este ejemplo se podrá evaluar que tanto como el dominante como el no dominante sean de el mismo tipo y verificar que se traten de elementos numéricos, así poder efectuar la resta con proceder a guardarla en la tabla de símbolos.

RANGE F

Con esta función se podrá recorrer un for indicando desde un numero hasta otro en el in

```
var val1, val2 interfaces.Symbol
    var array *arrayList.List
    array = arrayList.New()
    val1 = p.Exp_ini.EjecutarValor(env)
    val2 = p.Exp_fin.EjecutarValor(env)
    p.Tipo = val1.Tipo

    if (val1.Valor.(int) < val2.Valor.(int)) && (val1.Tipo ==
interfaces.INTEGER && val2.Tipo == interfaces.INTEGER) {
        //  fmt.Println("v1 ", val1.Valor.(int), "v2 ", val2.Valor.(int))
        //  fmt.Println("v1 ", val1.Valor, "v2 ", val2.Valor)
        //  fmt.Println("v1 ", val1, "v2 ", val2)

        var tmpSym interfaces.Symbol
        inival := val1.Valor.(int)

        tmpSym = interfaces.Symbol{
            Line:   p.Line,
            Column: p.Column,
            Id:     "",
            Tipo:   interfaces.INTEGER,
            Valor:  inival,
            IsMut:  true,
        }
        array.Add(tmpSym)

        for {
            inival++
            if inival < val2.Valor.(int) {
                tmpSym = interfaces.Symbol{
                    Line:   p.Line,
                    Column: p.Column,
                    Id:     "",
                    Tipo:   interfaces.INTEGER,
                    Valor:  inival,
                    IsMut:  true,
                }
                array.Add(tmpSym)
                //  fmt.Println(array.ToArray())
            } else {
                break
            }
```

```
        }
        //  fmt.Println(array)

    }
```

Como se puede observar el programa verificara si el valor izquierdo es menor que el derecho para poder ejecutar el for, cumpliendo esto se enviara la lista de valores recorridos a la tabla de símbolos para su posterior ejecución en el programa.

STRUCACCES

```
    if teStruct.Tipo == interfaces.STRUCT {

        if variable, ok :=
teStruct.Valor.(map[string]interfaces.Symbol)[p.Id]; ok {
            return variable
        }
        err.NewError("No existe atributo '"+p.Id+"' en struct",
env.(environment.Environment).Nombre, p.Line, p.Column)
        //return result
    }
```

Con esta función se verificara que se trate de un struct para poder ser buscado en la tabla de símbolos, de no ser encontrado será reportado como error en la lista de error

INSTRUCCIÓN

ASIGNACION

Se verificará si se trata de una lista o de una variable para posteriormente si su asignación se trata de su mismo tipo

```
a_valido := true
    ar_noelementos := 0
    arrType := l_tipo.GetValue(l_tipo.Len() - 1)

    //validar si es nulo
    if arrType.(interfaces.ArrayType).SizeA == nil {
        return true
    }
    res_exp :=
arrType.(interfaces.ArrayType).SizeA.(interfaces.Expresion).EjecutarValor(en
v)
    var arrSize int

    if res_exp.Tipo == interfaces.INTEGER {
        arrSize = res_exp.Valor.(int)
    } else {
        desc := fmt.Sprintf("Se esperaba un '%v' se tiene '%v'", "int",
interfaces.GetType(res_exp.Tipo))
        err.NewError("Error en Size "+desc,
env.(environment.Environment).Nombre, p.Line, p.Column)
        return false
    }
```

De tratarse del mismo tipo entonces será agregado a la tabla de símbolos

ASINACION DE VECTORES

```
    if result_mut.Tipo == interfaces.VECTOR {

        var index interfaces.Symbol
        index = p.index.EjecutarValor(env)

        var result interfaces.Symbol
        result = p.Expresion.EjecutarValor(env)

        if index.Tipo == interfaces.INTEGER {
        } else {
            desc := fmt.Sprintf("se esperaba '%v' se tiene '%v'",
interfaces.GetType(interfaces.INTEGER), interfaces.GetType(index.Tipo))
```

```
            err.NewError("Tipos no coinciden en av "+desc,
env.(environment.Environment).Nombre, p.Line, p.Column)
            return nil
        }

        valvec :=
result_mut.Valor.(interfaces.Symbol).Valor.(*arrayList.List)

        objec := valvec.GetValue(index.Valor.(int)).(interfaces.Symbol)
        //fmt.Println("---objec", objec)

        if objec.Tipo == interfaces.STRUCT {
        } else {
            desc := fmt.Sprintf("se esperaba '%v' se tiene '%v'",
interfaces.GetType(interfaces.STRUCT), interfaces.GetType(objec.Tipo))
            err.NewError("Tipos no coinciden "+desc,
env.(environment.Environment).Nombre, p.Line, p.Column)
            return nil
        }

        env.(environment.Environment).UpdateStructVector(p.ListAccesStruct,
result, p.Line, p.Column, objec.Valor.(map[string]interfaces.Symbol))
```

con esta función se verificará si se trata de un vector al obtener su valor de su tabla de símbolos y poder compararlo con los valores ya establecidos, de tratarse de un vector se obtendrá el valor de la asignación y se le agregará en la tabla de símbolos

DECLARACION

Aquí se enviara cada variable y su tipo para ser comparado y validar que no se trate de un valor nulo, una vez validado eso se procede a guardar la variable en la tabla de símbolos

```
if result.Tipo == interfaces.NULL {
            return nil
        }

        if result.Tipo == p.Tipo {
            env.(environment.Environment).SaveVariable(p.Id, result, p.Tipo,
p.IsMut, p.Line, p.Column, env.(environment.Environment).Nombre, nil, 0)

        } else if p.Tipo == interfaces.NULL {

            if result.Tipo == interfaces.ARRAY || result.Tipo ==
interfaces.VECTOR {
```

```
                env.(environment.Environment).SaveVariable(p.Id, result,
result.Tipo, p.IsMut, p.Line, p.Column,
env.(environment.Environment).Nombre, nil,
result.Valor.(*arrayList.List).Len())
            } else {
                /*no tiene tipo en asignacion, se le asigna el tipo de la
expresion*/
                env.(environment.Environment).SaveVariable(p.Id, result,
result.Tipo, p.IsMut, p.Line, p.Column,
env.(environment.Environment).Nombre, nil, 0)
            }

        }
```

FOR

Aquí se enviaran los elementos de la lista de números o cadena que posee el for en sus instrucciones

```
   for _, s := range arr.ToArray() {
            //crendo entorno
            var loopEnv environment.Environment
            loopEnv = environment.NewEnvironment("Forin",
env.(environment.Environment))
            //agregando variable al entorno
            ////fmt.Println("s: ", s)

            loopEnv.SaveVariable(p.Id, s.(interfaces.Symbol),
interfaces.ARRAY, true, p.Line, p.Column,
env.(environment.Environment).Nombre, nil, 0)

            //instrucciones
            for _, b := range p.Inst.ToArray() {
                rest := b.(interfaces.Instruction).Ejecutar(loopEnv)
                if rest != nil {
                    if reflect.TypeOf(rest) ==
reflect.TypeOf(interfaces.Symbol{}) {
                        if rest.(interfaces.Symbol).TipoRet ==
interfaces.BREAK {
                            Isbreak = true
                            break
                        }
                        if rest.(interfaces.Symbol).TipoRet ==
interfaces.CONTINUE {
                            break
```

```
                        }
                        if rest.(interfaces.Symbol).TipoRet ==
interfaces.RETURN {
                            return result
                        }
                    }
                }
            }
            if Isbreak == true {
                break
            }
        }
```

Una vez en el for se detectara si el break y continue están dentro del forcé para así continuar o retornar el valor dado.

IF

```
    if result.Valor == true {

        var tmpEnv environment.Environment
        tmpEnv = environment.NewEnvironment("if",
env.(environment.Environment))

        var rest interface{}
        for _, s := range i.LB_Principal.ToArray() {
            rest = s.(interfaces.Instruction).Ejecutar(tmpEnv)
            if rest != nil {

                if reflect.TypeOf(rest) ==
reflect.TypeOf(interfaces.Symbol{}) {

                    if rest.(interfaces.Symbol).TipoRet == interfaces.BREAK
|| rest.(interfaces.Symbol).TipoRet == interfaces.CONTINUE ||
rest.(interfaces.Symbol).TipoRet == interfaces.RETURN {
                        return rest
                    }
                }
            }
        }
        return rest

    }
```

Con esto padreemos saber si este valor existe, de ser así este podrá acceder a su valor para poder ser evaluado en el entorno que este tendrá por sí mismo, en el que se almacenaran sus instrucciones en un array para posteriormente ser ejecutado y almacenado en la tabla de símbolos.

PRINT

Aquí se imprimirán los valores que estén en la tabla de símbolos, el algoritmo consiste en tomar los valores de la tabla de símbolos y remarcar mediante la bandera “{}” para detectar el lugar donde ira el valor establecido después de la “,” en el programa

```
 format_str = format_str + "{}"
                format_str = strings.Replace(format_str, "{}",
fmt.Sprintf("%v", expre_print.Valor), 1)
```