Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ingeniería en Ciencias y Sistemas

Organización de Lenguajes y Compiladores 2

Catedráticos: Ing. Luis Espino

Tutores Académico: Estuardo Sebastián Valle Bances

V-Lang Cherry

PROYECTO 1

Programadores:

- Kevin Andres Alvares Herrera
- Marco Pool Chiché Sapón
- Vicente Rocael Matías Osorio

Carne:

- 202203038
- 3357975470901
- 3208209201308

INDICE

Objetivos

Especificos

1) La creación de un intérprete que permita transformar por medio de analizadores, léxicos, sintácticos y semánticos junto con la herramienta de un arbol CST para el lenguaje de programación V-Lang Cherry, un lenguaje inspirado en Vlang.

Generales

1) La creación de un analizador léxico para el reconocimiento de tokens provenientes de diferentes entradas.
2) La creación y adaptación de un analizador sintáctico que permita el reconocimiento del orden proveniente de los tokens para validar cadenas de entrada y proporcionarle dichas cadenas al analizador semántico.
3) La creación de una interfaz gráfica la cual el usuario pueda interactuar de una forma cómoda y agradable.

Especificaciones Software y Hardware

Software necesario para la ejecución del programa:

- Go en la versión 1.24.3
- Antlr4 en la versión 4.13.2
- Linux distribución (mint / archi linux)
- Visual Studio Code 1.97

Hardware recomendado para la ejecución del programa:

- Procesador i5
- Ram 16gb
- Monitor
- Mouse
- Teclado

Carpetas y subcarpetas utilizadas

- V-Lang/
  - backend/
  - interprete.go: Lógica principal del intérprete.
  - analizador/
    - operaciones/: Operaciones aritméticas, booleanas, relacionales y secuencias de escape.
    - parser/: Gramática ANTLR, parser generado y visitantes/listeners base.
    - sentencias/: Implementaciones de sentencias como If.
    - listener/: Listeners personalizados para ANTLR.
    - visitor/: Visitantes para evaluación, manejo de errores y símbolos.
  - frontend/
    - ui.go: Interfaz gráfica con Fyne.
    - cst/: Reporte y visualización del árbol de sintaxis concreta.
    - errors/: Manejo y visualización de errores.
    - symbols/: Implementación de la tabla de símbolos.
- Archivos raíz: main.go, generate.go, generate.sh, go.mod, go.sum, antlr-4.13.2-complete.jar.

Descripción de estructuras y atributos principales

- symbols.Entorno: Representa un ámbito de declaración.
    - Nombre string
    - Padre *Entorno
    - Hijos []*Entorno
    - Simbolos map[string]*Simbolo
- symbols.Simbolo: Elemento de la tabla de símbolos.
    - ID string
    - TipoSimbolo string
    - TipoDato string
    - Ambito string
    - Linea int
    - Columna int
    - Valor interface{}
- symbols.TablaSimbolos: Maneja todos los símbolos del programa.
    - EntornoGlobal *Entorno
    - EntornoActual *Entorno
    - Errores *errors.ErrorTable
    - EntornosFunciones map[string]*Entorno
    - visitor.EvaluationError: Error de evaluación.
    - Message string
    - Line int
    - Column int

- visitor.EvalVisitor: Visitante principal para evaluar el AST.
    - Entorno map[string]interface{}
    - Tabla *symbols.TablaSimbolos
    - Console *strings.Builder
    - StructDef map[string][]StructAtributo

Otros atributos para control de flujo y funciones.

visitor.StructAtributo: Atributo de un struct.

Tipo string

Id string

Métodos clave de la aplicación

- EvalVisitor.Visit: Inicia la evaluación del árbol de sintaxis.

```go
func (v *EvalVisitor) Visit(tree antlr.ParseTree) interface{} {
    fmt.Println("Visit: ", reflect.TypeOf(tree))
    return tree.Accept(v)
}
```

- EvalVisitor.VisitDeclaracionMultiple: Procesa declaraciones múltiples de variables.

```go

func (v *EvalVisitor) VisitDeclaracionMultiple(ctx *parser.DeclaracionMultipleContext) interface{} {
    tipo := ctx.Tipos().GetText()
    valores := obtenerValores(ctx.ListaExpr(), v)
    ids := obtenerIDs(ctx.ListaIDS())
    if len(ids) != len(valores) {
        msg := fmt.Sprintf("Error: cantidad de variables (%d) no coincide con valores (%d)",
            len(ids), len(valores))
        v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
        return nil

    }

    for i, id := range ids {
        valor := valores[i]
        valorTipo := inferirTipo(valor)
        if !tiposCompatibles(tipo, valorTipo) {
            msg := fmt.Sprintf("Error: no se puede asignar %s a variable %s de tipo %s",
                valorTipo, id, tipo)
            v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
            continue
        }

        simbolo := &symbols.Simbolo{
            ID:          id,
            TipoDato:    tipo,
            Valor:       valor,
            Ambito:      v.Tabla.EntornoActual.Nombre,
            TipoSimbolo: "variable",
            Linea:       ctx.GetStart().GetLine(),
            Columna:     ctx.GetStart().GetColumn(),
        }
        v.Tabla.EntornoActual.Simbolos[id] = simbolo
    }
    return nil
}
```

- EvalVisitor.VisitStructDef: Procesa la definición de structs.

```go
func (v *EvalVisitor) VisitStructDef(ctx *parser.StructDefContext) interface{} {
    nombreStruct := ctx.IDENTIFICADOR().GetText()
    var atributos []StructAtributo

    fmt.Printf("[StructDef] Definiendo struct: %s\n", nombreStruct)

    for _, atrCtx := range ctx.AllAtributos() {
        tipo := atrCtx.Tipos().GetText()
        id := atrCtx.IDENTIFICADOR().GetText()
        fmt.Printf("  - Atributo: %s %s\n", tipo, id)
        atributos = append(atributos, StructAtributo{Tipo: tipo, Id: id})
    }

    v.StructDef[nombreStruct] = atributos
    fmt.Printf("[StructDef] Struct '%s' registrado con atributos: %+v\n", nombreStruct, atributos)
    return nil
}

```

- EvalVisitor.VisitAsignacionMultiple: Procesa asignaciones múltiples.

```go
func (v *EvalVisitor) VisitAsignacionMultiple(ctx *parser.AsignacionMultipleContext) interface{} {
    ids := obtenerIDs(ctx.ListaIDS())
    valores := obtenerValores(ctx.ListaExpr(), v)

    if len(ids) != len(valores) {
        v.Tabla.Errores.NewSemanticError(ctx.GetStart(), "Número de variables y valores no coincide en declaración múltiple")
        return nil
    }

    for i, id := range ids {
        valor := valores[i]

        // Copia profunda para evitar aliasing accidental
        switch val := valor.(type) {
        case []interface{}:
            copia := make([]interface{}, len(val))
            copy(copia, val)
            valor = copia
        case [][]interface{}:
            copia := make([][]interface{}, len(val))
            for j, fila := range val {
                if fila == nil {
                    copia[j] = nil
                    continue
                }
                subcopia := make([]interface{}, len(fila))
                copy(subcopia, fila)
                copia[j] = subcopia
            }
            valor = copia
        }

        tipoInferido := inferirTipo(valor)

        // DEBUG: Entorno actual
        fmt.Printf("DEBUG: Buscando símbolo '%s' en entorno '%s'\n", id, v.Tabla.EntornoActual.Nombre)

        simbolo := v.Tabla.EntornoActual.BuscarSimbolo(id)
        if simbolo != nil {
            // Validar compatibilidad de tipos
            if !tiposCompatibles(simbolo.TipoDato, tipoInferido) {
                msg := fmt.Sprintf("No se puede asignar tipo '%s' a '%s' de tipo '%s'", tipoInferido, id, simbolo.TipoDato)
                v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
                continue
            }
            simbolo.Valor = valor
            v.Entorno[id] = valor
            fmt.Printf("Actualización: %s = %v (tipo: %s)\n", id, valor, tipoInferido)
        } else {
            // Declaración implícita
            v.Tabla.DeclararVariableSimple(id, tipoInferido, valor, ctx, v.Tabla.EntornoActual.Nombre)
            v.Entorno[id] = valor
            fmt.Printf("Declaración: %s = %v (tipo: %s)\n", id, valor, tipoInferido)
        }
    }

    return nil
}
```

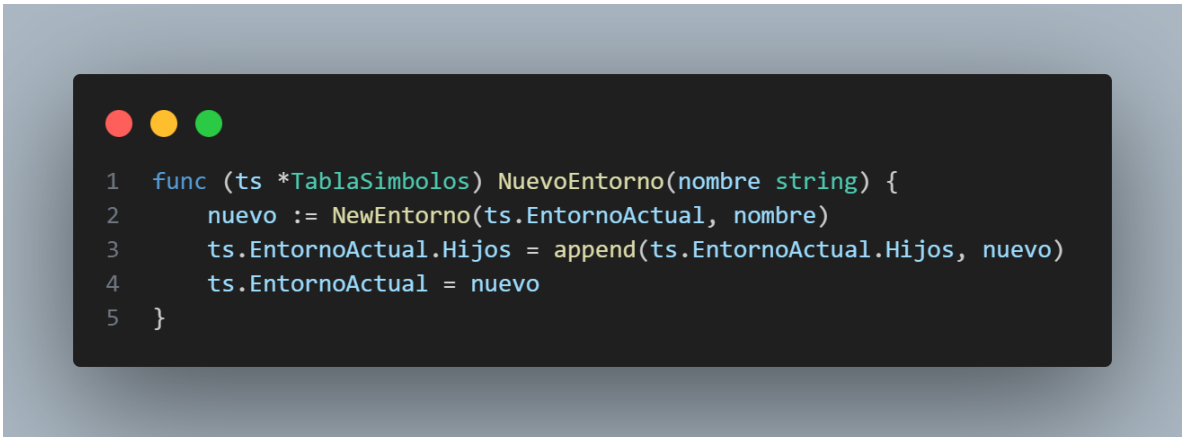
- EvalVisitor.VisitSliceDef: Procesa la definición de slices.

```go
func (v *EvalVisitor) VisitSliceDef(ctx *parser.SliceDefContext) interface{} {
    id := ctx.IDENTIFICADOR().GetText()
    literal := ctx.SliceLiteral()

    // 1. Determinar tipo de slice
    tipo := literal.Tipos().GetText()

    // 2. Unidimensional
    if literal.ListaExpr() != nil {
        exprs := literal.ListaExpr()
        var elementos []interface{}

        for _, exprCtx := range exprs.AllExpresion() {
            valor := v.Visit(exprCtx)

            if !v.checkTipo(valor, tipo) {
                v.Tabla.Errores.NewSemanticError(ctx.GetStart(), "tipo incompatible en slice inválida")
                return nil
            }
            elementos = append(elementos, valor)
        }

        v.Tabla.DeclararVariableSimple(id, "slice("+tipo+")", elementos, ctx, v.Tabla.EntornoActual.Nombre)

    } else if literal.ListaExprList() != nil {
        // 3. Bidimensional
        lista := literal.ListaExprList()
        var matriz [][]interface{}

        for _, filaCtx := range lista.AllListaExpr() {
            var fila []interface{}
            for _, exprCtx := range filaCtx.AllExpresion() {
                valor := v.Visit(exprCtx)
                if !v.checkTipo(valor, tipo) {
                    v.Tabla.Errores.NewSemanticError(ctx.GetStart(), "tipo incompatible en slice bidimensional")
                    return nil
                }
                fila = append(fila, valor)
            }
            matriz = append(matriz, fila)
        }

        v.Tabla.DeclararVariableSimple(id, "slice(slice("+tipo+"))", matriz, ctx, v.Tabla.EntornoActual.Nombre)

    } else {
        v.Tabla.Errores.NewSemanticError(ctx.GetStart(), "Definición de slice inválida")
    }

    return nil
}
```

- TablaSimbolos.DeclararVariable: Declara una variable en la tabla de símbolos.

```go
func (ts *TablaSimbolos) DeclararVariable(id, tipo string, valor interface{}, ctx antlr.ParserRuleContext, nombreEntorno string) {
    for _, s := range ts.EntornoActual.Simbolos {
        if s.ID == id {
            // Ya existe en este entorno → reasignar valor
            s.Valor = valor
            return
        }
    }

    // Si no existe → declarar
    simbolo := &Simbolo{
        ID:          id,
        TipoSimbolo: "variable",
        TipoDato:    tipo,
        Valor:       valor,
        Ambito:      ts.EntornoActual.Nombre,
        Linea:       ctx.GetStart().GetLine(),
        Columna:     ctx.GetStart().GetColumn(),
    }
    ts.Insertar(simbolo)
}
```

- TablaSimbolos.NuevoEntorno: Crea un nuevo ámbito.



```go
func (ts *TablaSimbolos) NuevoEntorno(nombre string) {
    nuevo := NewEntorno(ts.EntornoActual, nombre)
    ts.EntornoActual.Hijos = append(ts.EntornoActual.Hijos, nuevo)
    ts.EntornoActual = nuevo
}
```

- TablaSimbolos.GenerarReporte: Genera un reporte de todos los símbolos.

```go
func (ts *TablaSimbolos) GenerarReporte() []*Simbolo {
    var todosSimbolos []*Simbolo

    var recolectar func(e *Entorno)
    recolectar = func(e *Entorno) {
        for _, simbolo := range e.Simbolos {
            todosSimbolos = append(todosSimbolos, simbolo)
        }

        // Recorrer entornos hijos para recolectar sus símbolos
        for _, hijo := range e.Hijos {
            recolectar(hijo)
        }
    }

    recolectar(ts.EntornoGlobal)

    // Ordenar por Ámbito, Línea y Columna
    sort.Slice(todosSimbolos, func(i, j int) bool {
        if todosSimbolos[i].Ambito == todosSimbolos[j].Ambito {
            if todosSimbolos[i].Linea == todosSimbolos[j].Linea {
                return todosSimbolos[i].Columna < todosSimbolos[j].Columna
            }
            return todosSimbolos[i].Linea < todosSimbolos[j].Linea
        }
        return todosSimbolos[i].Ambito < todosSimbolos[j].Ambito
    })

    return todosSimbolos
}
```

- CstReport: Genera el reporte del árbol de sintaxis concreta (CST). Por medio de una solicitud a antlr.lab

```go
func CstReport(input string) string {
    _, filename, _, _ := runtime.Caller(0)
    path := filepath.Dir(filename)

    gramaticaPath := filepath.Join(path, "..", "..", "backend", "analizador", "parser", "gramatica.g4")

    absPath, _ := filepath.Abs(gramaticaPath)
    fmt.Println("Leyendo gramática en:", absPath)

    gramaticaContent, err := ReadFile(gramaticaPath)
    if err != nil {
        fmt.Println("Error leyendo la gramática:", err)
        return ""
    }

    gramaticaJSON, err := json.Marshal(gramaticaContent)
    if err != nil {
        fmt.Println("Error haciendo marshal de gramática:", err)
        return ""
    }

    jinput, err := json.Marshal(input)
    if err != nil {
        fmt.Println("Error haciendo marshal de input:", err)
        return ""
    }

    payload := []byte(fmt.Sprintf(
        `{
        "grammar": %s,
        "input": %s,
        "lexgrammar": %s,
        "start": "%s"
    }`,
        string(gramaticaJSON),
        string(jinput),
        string(gramaticaJSON),
        "init",
    ))

    req, err := http.NewRequest("POST", "http://lab.antlr.org/parse/", bytes.NewBuffer(payload))
    if err != nil {
        fmt.Println("Error creando request:", err)
        return ""
    }
    req.Header.Set("Content-Type", "application/json")

    client := &http.Client{}
    resp, err := client.Do(req)
    if err != nil {
        fmt.Println("Error enviando request:", err)
        return ""
    }
    defer resp.Body.Close()

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error leyendo respuesta:", err)
        return ""
    }

    var data map[string]interface{}
    err = json.Unmarshal(body, &data)
    if err != nil {
        fmt.Println("Error unmarshalling json:", err)
        return ""
    }

    resultRaw, ok := data["result"]
    if !ok {
        fmt.Println("La respuesta no contiene 'result'")
        return ""
    }

    result, ok := resultRaw.(map[string]interface{})
    if !ok {
        fmt.Println("'result' no es un objeto")
        return ""
    }

    svgtreeRaw, ok := result["svgtree"]
    if !ok {
        fmt.Println("No existe 'svgtree' en 'result'")
        return ""
    }

    svgtree, ok := svgtreeRaw.(string)
    if !ok {
        fmt.Println("'svgtree' no es un string")
        return ""
    }

    return svgtree
}
```

Estructuras de datos utilizadas

- Mapas: Uso extensivo para la tabla de símbolos (map[string]*Simbolo), entornos, definición de structs, etc.
- Listas (slices): Para almacenar listas de atributos, parámetros, valores, errores, etc.
- Estructuras (structs): Para representar variables, funciones, entornos, errores, etc.

Librerías empleadas

- ANTLR4: Generación de analizadores léxicos y sintácticos.
- Fyne: Interfaz gráfica de usuario.
- Go estándar: fmt, strings, sort, os, io, net/http, etc.
- github.com/antlr4-go/antlr/v4: Integración de ANTLR con Go.