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 ASM

PROYECTO 2

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 y luego pasa a traducirse a lenguaje ensamblador aarch64

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 correcta traducción del lenguaje ensamblador aarch64

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
- qemu

Hardware recomendado para la ejecución del programa:

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

Carpetas y subcarpetas utilizadas

- V-Lang/
    - Assembler/
    - backend/
        - analizador/
            - parser/
            - traducciones/
        - visitor/
    - 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.
- Archivos relacionados a ensamblador: decAsig.go, funcionesEmbebidas.go, makeFile, build.sh, program y program.s

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

Métodos clave de la aplicación

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

```go
func (v *EvalVisitor) Visit(tree antlr.ParseTree) interface{} {
    switch val := tree.(type) {
    case *antlr.ErrorNodeImpl:
        log.Fatal(val.GetText())
        return nil
    default:
        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 _, id := range ids {
        if yaDeclaradoEnAmbitoActual(id, tipo, v.Tabla) {
            msg := fmt.Sprintf("Error: variable '%s' ya fue declarada con otro tipo en este ámbito", id)
            v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
            return nil
        }
    }

    if err := traducciones.ProcesarDeclaracionMultiple(ids, valores, tipo, v.Tabla, &v.OutputASM); err != nil {
        v.Tabla.Errores.NewSemanticError(ctx.GetStart(), err.Error())
    }
    return nil
}

```

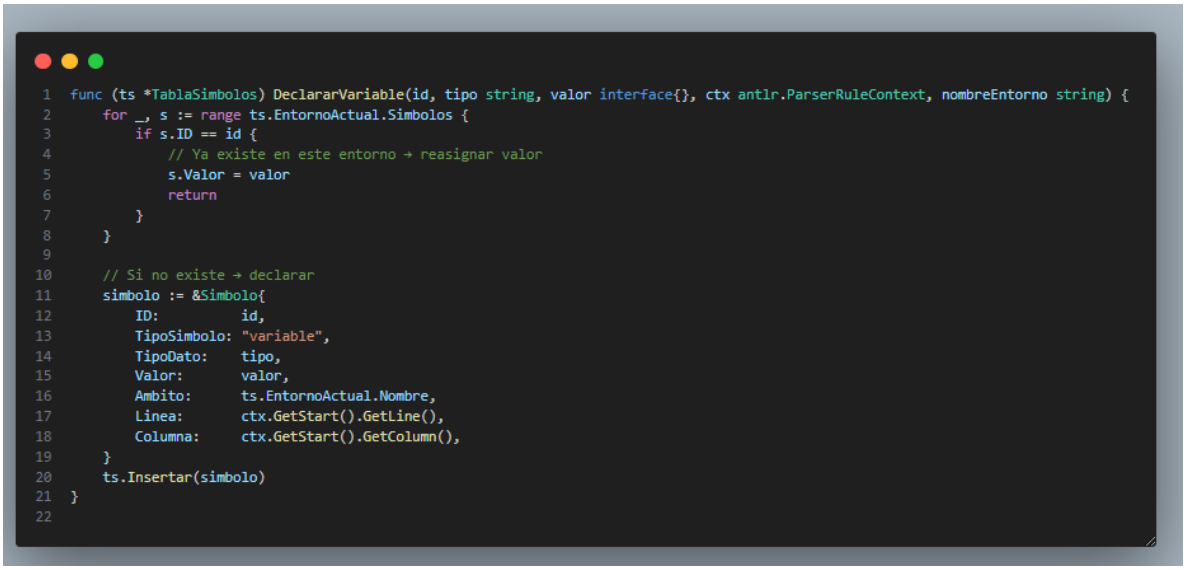
- EvalVisitor.VisitAsignacionMultiple: Procesa asignaciones múltiples.

```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 asignación múltiple")
        return nil
    }

    for i, id := range ids {
        valor := valores[i]
        valorInferido := inferirTipo(valor)

        simbolo := v.Tabla.EntornoActual.BuscarSimbolo(id)
        if simbolo == nil {
            msg := fmt.Sprintf("Error: variable '%s' no ha sido declarada en este ámbito", id)
            v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
            continue
        }

        if !tiposCompatibles(simbolo.TipoDato, valorInferido) {
            msg := fmt.Sprintf("Error: tipo incompatible en asignación. '%s' es de tipo '%s', no '%s'",
                id, simbolo.TipoDato, valorInferido)
            v.Tabla.Errores.NewSemanticError(ctx.GetStart(), msg)
            continue
        }

        // Actualización
        simbolo.Valor = valor
        v.Entorno[id] = valor
        fmt.Printf("Asignación: %s = %v (tipo: %s)\n", id, valor, valorInferido)

        // Generar ASM (opcional)
        err := traducciones.GenerarCodigoDeclaracionSinTipo(id, simbolo.TipoDato, valor, &v.OutputASM)
        if err != nil {
            v.Tabla.Errores.NewSemanticError(ctx.GetStart(), err.Error())
        }
    }

    return nil
}
```

- EvalVisitor.VisitSliceDef: Procesa la definición de slices.

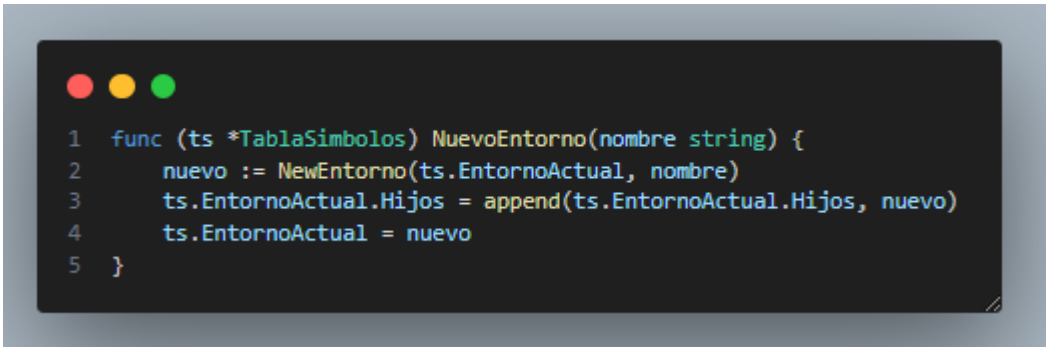
```go
func (v *EvalVisitor) VisitSliceDef(ctx *parser.SliceDefContext) interface{} {
    id := ctx.IDENTIFICADOR().GetText()
    literal := ctx.SliceLiteral()

    // 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
}
```

Metodos encargados de traducciones

Archivo funcionesEmbebidas.go

```go
package traducciones

import (
    "fmt"
    "strings"
)

var contadorEtiqueta int = 0
var DataBuilder strings.Builder
var TextBuilder strings.Builder
var mensajesUnicos = make(map[string]string)
var FuncionesBuilder strings.Builder
var contadorSuma int = 0
var contadorResta int = 0
var contadorDivision int = 0
var contadorMulti int = 0
var contadorMod int = 0

func GenerarCodigoPrint(msg string, addNewline bool) {
    if addNewline {
        msg += "\n"
    }
    if etiqueta, existe := mensajesUnicos[msg]; existe {
        // Ya fue agregado, simplemente usar etiqueta
        TextBuilder.WriteString(fmt.Sprintf(`mov x0, #1
adr x1, %s
mov x2, #%d
mov w8, #64
svc #0

`, etiqueta, len(msg)))
        return
    }

    etiqueta := fmt.Sprintf("msg_%d", contadorEtiqueta)
    contadorEtiqueta++
    mensajesUnicos[msg] = etiqueta

    DataBuilder.WriteString(fmt.Sprintf("%s: .ascii \"%s\"\n", etiqueta, escape(msg)))

    TextBuilder.WriteString(fmt.Sprintf(`mov x0, #1
adr x1, %s
mov x2, #%d
mov w8, #64
svc #0

`, etiqueta, len(msg)))
}

func escape(input string) string {
    input = strings.ReplaceAll(input, "\n", `\n`)
    input = strings.ReplaceAll(input, `"`, `\"`)
    return input
}

func ResetearCodigoASM() {
    contadorEtiqueta = 0
    DataBuilder.Reset()
    TextBuilder.Reset()
    mensajesUnicos = make(map[string]string)
    FuncionesBuilder.Reset()
    variablesReservadas = make(map[string]bool)
    contadorSuma = 0
}
```

En el archivo decAsig.go encontraran

Variables globales

- contadorFloat, contadorString: Contadores para generar nombres únicos de etiquetas para valores float y string en el ensamblador.
- variablesReservadas: Mapa para llevar control de las variables ya reservadas en la sección .data del ensamblador y evitar duplicados.

Funciones

- ProcesarDeclaracionMultiple: Procesa la declaración de varias variables a la vez, verifica compatibilidad de tipos, genera código ensamblador y actualiza la tabla de símbolos.
- generarCodigoDeclaracion: Según el tipo de valor (int, float, bool, string, slice), llama a la función correspondiente para generar el código ensamblador de la declaración.
- GenerarCodigoDeclaracionSinTipo: Similar a la anterior, pero espera que el tipo ya esté definido y verifica que el valor coincida con ese tipo.
- inferirTipo: Determina el tipo de un valor en tiempo de ejecución (int, float, string, bool, slice, etc.).
- tiposCompatibles: Verifica si dos tipos (el declarado y el real) son compatibles para asignación, incluyendo casos como float/int y slices anidados.
- extraerTipoInterno: Extrae el tipo interno de un slice, útil para comparar tipos de slices anidados.
- generarCodigoInt, generarCodigoFloat, generarCodigoString, generarCodigoBool: Generan el código ensamblador necesario para inicializar variables de tipo int, float, string y bool respectivamente.
- reservarVariableEnData: Reserva espacio en la sección .data del ensamblador para una variable, según su tipo, y marca la variable como reservada.

Estructuras de datos utilizadas

- Mapas: Uso extensivo para la tabla de símbolos (map[string]*Simbolo), entornos, 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.