Universidad de San Carlos de Guatemala
LABORATORIO ORGANIZACION DE LENGUAJES Y COMPILADORES 2

# Manual Técnico

Nombre: José Eduardo Galdámez González
Carnet: 202109732

# Manual Técnico:

Bienvenidos al Manual Técnico del Compilador T-Swift, una guía exhaustiva que detalla el proceso de desarrollo y funcionamiento de nuestro compilador para el lenguaje T-Swift. En este documento, abordaremos los aspectos esenciales del proyecto, destacando los enfoques y las competencias clave que permitieron la creación exitosa de esta poderosa herramienta de desarrollo de software.

I. Desarrollo del Compilador T-Swift

Nuestro proyecto se centró en la creación de un compilador completo para el lenguaje T-Swift, que se inspira en la sintaxis de Swift pero con un enfoque simplificado. A lo largo de este manual, se detallará el proceso de desarrollo, que se llevó a cabo siguiendo una serie de etapas críticas:

- Análisis Léxico, Sintáctico y Semántico: Utilizamos herramientas clave como ANTLR4 para realizar el análisis léxico y sintáctico del código fuente T-Swift. Esto implicó definir las reglas gramaticales del lenguaje y construir un árbol de sintaxis abstracta para representar la estructura del programa.
- Generación de Código Ejecutable: Implementamos la generación de código ejecutable a partir del árbol de sintaxis abstracta. Esto permitió a los programadores compilar programas escritos en T-Swift en ejecutables funcionales.

## II. Competencias Adquiridas

Durante el proceso de desarrollo del compilador T-Swift, los miembros del equipo adquirieron competencias clave en el ámbito de la compilación y la traducción de lenguajes de programación. Algunas de las competencias más destacadas incluyen:

- Uso de Herramientas Léxicas y Sintácticas: Aprendimos a utilizar herramientas como ANTLR4 para el reconocimiento y análisis del lenguaje, lo que facilitó la definición de la gramática de T-Swift y la generación de analizadores léxicos y sintácticos.
- Traducción Orientada por la Sintaxis: Implementamos reglas semánticas haciendo uso de atributos sintetizados y heredados, lo que permitió la traducción efectiva del código fuente en T-Swift a código ejecutable.
- Familiarización con Estructuras de Datos y Herramientas de Interprete: A lo largo del proyecto, nos familiarizamos con las herramientas y estructuras de datos necesarias para la creación de un compilador e intérprete eficientes.

## III. Estructura del Manual

Este manual está organizado de manera lógica y secuencial para proporcionar a los desarrolladores y usuarios una comprensión profunda del funcionamiento interno del compilador T-Swift. Cada sección aborda aspectos específicos del desarrollo y la implementación del compilador, desde el análisis léxico hasta la generación de código.

# Patrón Visitor:

En el proyecto de desarrollo del compilador T-Swift, se utilizó el patrón Visitor de manera fundamental para llevar a cabo el análisis semántico y la generación de código a partir del árbol de sintaxis abstracta (AST) generado durante el análisis sintáctico. A continuación, se explica qué es el patrón Visitor y cómo se aplicó de manera concisa e informativa en el proyecto, lo que puede servir de guía para otros desarrolladores:

**¿Qué es el patrón Visitor?**

El patrón Visitor es un patrón de diseño de software que permite separar las operaciones a realizar en una estructura de objetos compleja de la propia estructura. En el contexto de un compilador, el patrón Visitor se utiliza para recorrer un árbol de sintaxis abstracta y realizar acciones específicas en los nodos del árbol sin necesidad de modificar directamente las clases de los nodos.

```
var out = ""
var errores=NewErrorList()
var simbolos=NewSymbolList()
var cont int
var entorno string

type VisitorEvalue struct {
    parser.BaseSwiftLanVisitor
    globalScope    *Scope
    currentScope   *Scope
    returnValue    interface{}
    returnVoid     bool
    returnBreak    bool
    returnContinue bool
    funcsname string
}
```

```
func OutData() string {
    return out
}
func OutErrors() *ErrorList {
    return errores
}
func OutSimbolos() *SymbolList {
    return simbolos
}

func NewVisitorEvalue() *VisitorEvalue {
    globalScope := NewScope()
    return &VisitorEvalue{
        globalScope:  globalScope,
        currentScope: globalScope,
    }
}
```

```go
func (e *VisitorEvalue) Visit(tree antlr.ParseTree) interface{} {
    fmt.Printf("Visitando: %v\n", reflect.TypeOf(tree))
    switch t := tree.(type) {
    case *antlr.ErrorNodeImpl:
        cont=cont+1
        fmt.Printf("Error: - %v\n", t.GetText())
        errores.AddError(cont,"Error Sintactico:"+t.GetText(),entorno,t.GetSymbol().GetLine(),t.GetSymbol().GetCo
        return nil
    default:
        return tree.Accept(e)
    }
}

func (e *VisitorEvalue) VisitInicio(ctx *parser.InicioContext) interface{} {
    fmt.Printf("Calculating Program: %s\n", ctx.GetText())
    errores.ResetErrors()
    simbolos.ResetSymbols()
    cont=0
    out = ""
    entorno="Global"
    return e.VisitChildren(ctx)
}
```

```go
func (e *VisitorEvalue) VisitChildren(node antlr.RuleNode) interface{} {
    for _, n := range node.GetChildren() {
        e.Visit(n.(antlr.ParseTree))
    }
    return VOID
}

func (e *VisitorEvalue) VisitSentencias(ctx *parser.SentenciasContext) interface{} {
    fmt.Printf("Entro - Sentencias\n")
    for _, StamentsCtx := range ctx.AllStatement() {

        e.Visit(StamentsCtx)
        if e.returnValue != nil{
            fmt.Println(e.returnValue)
            return e.returnValue
        } else if e.returnVoid {
            return RETURNVOID
        } else if e.returnBreak {
            return BREAK
        } else if e.returnContinue {
            return CONTINUE
        }
```

```go
func (e *VisitorEvalue) VisitStatement(ctx *parser.StatementContext) interface{} {
    fmt.Printf("Entro VisitStatement\n")
    childCount := ctx.GetChildCount()

    for i := 0; i < childCount; i++ {
        child := ctx.GetChild(i).(antlr.ParseTree)
        e.Visit(child)
    }
    return VOID

}
```

Aquí hay una explicación de cómo se utiliza el patrón Visitor en este código y cómo ha sido beneficioso:

1. **Definición de la clase VisitorEvalue:** Se define una estructura VisitorEvalue que implementa la interfaz parser.SwiftLanVisitor. Esta estructura se utiliza para recorrer el AST generado por ANTLR4 y realizar acciones específicas durante la visita de los nodos del árbol.
2. **Método Visit(tree antlr.ParseTree) interface{}:** Este método se llama cuando se visita un nodo en el árbol. Se utiliza un switch para determinar el tipo de nodo y ejecutar el comportamiento correspondiente. Por ejemplo, cuando se encuentra un antlr.ErrorNodeImpl, se registra un error sintáctico.
3. **Método VisitInicio(ctx *parser.InicioContext) interface{}:** Este método se llama al comienzo del análisis del programa. Se restablecen los errores y los símbolos, y se inicializan algunas variables importantes.
4. **MétodoVisitSentencias(ctx*parser.SentenciasContext ) interface{}:** Este método se utiliza para visitar las sentencias dentro del programa. Itera a través de las sentencias y llama al método Visit correspondiente para cada una. También maneja los casos de retorno, break y continue.
5. **Método VisitStatement(ctx*parser.StatementContext) interface{}:** Este método se utiliza para visitar una declaración dentro de las sentencias. Recorre los hijos de la declaración y llama a Visit para cada uno.

# Estructura de la Gramática

La gramática que rige el Compilador T-Swift se ha diseñado cuidadosamente para permitir la escritura y análisis de código en el lenguaje T-Swift. A continuación, se describen los principales componentes de la gramática:

**Sentencias**
El compilador T-Swift es capaz de manejar una variedad de sentencias, incluyendo asignaciones, llamadas a funciones, bucles, condicionales y más. Cada tipo de sentencia se define de manera precisa y se adapta a la sintaxis del lenguaje.

**Declaración de Variables, Vectores, Matrices, Funciones y Structs**
El compilador es versátil en términos de declaración de variables, vectores y matrices, así como en la definición de funciones y estructuras (structs). Los programadores pueden utilizar las palabras clave "let" y "var" para definir variables, y también pueden trabajar con estructuras de datos complejas como matrices y vectores.

**Llamadas a Funciones**
El compilador admite la definición y llamada de funciones, lo que permite a los desarrolladores modularizar su código de manera efectiva. Las funciones pueden recibir parámetros y devolver valores, lo que facilita la implementación de lógica personalizada.

**Ciclos y Condicionales**
Se han implementado bucles "for" y "while" para facilitar la iteración y la toma de decisiones en el código. Además, el compilador admite condicionales "if" y "switch", lo que permite un flujo de control de programa flexible.

**Expresiones y Operadores**
El lenguaje T-Swift admite una amplia gama de expresiones y operadores, incluyendo operadores aritméticos, lógicos y de comparación. Estas expresiones se pueden utilizar para realizar cálculos y tomar decisiones dentro del código.

**Palabras Reservadas**
El compilador reconoce y gestiona las palabras reservadas del lenguaje T-Swift, como "if", "for", "while" y muchas otras, garantizando un análisis léxico preciso.

**Uso de la Gramática**
Este manual proporciona ejemplos y explicaciones detalladas sobre cómo utilizar cada componente de la gramática en el código T-Swift. Los desarrolladores pueden consultar esta referencia para comprender la sintaxis y la semántica del lenguaje de manera completa.

# GRAMÁTICA ANTLR4

```
grammar SwiftLan;

inicio: sentencias;

sentencias: (statement)*;

//Sentencias
statement:
        asignacion
        | reasignacion
        | vectorAsign
        | funcstmt
        | fPrint
        | ifstmt
        | callFuncstmt
        | forstmt
        | switchstmt
        | whilestmt
        | retStmt
        | breakstmt
        | continuestmt
        | appendVec
        | removeVec
        | removeLastVec
        | vecReasig
        | guardstmt
        | incremento
        | decremento
        | matrizAsign
        | reasigMatriz
        | defStruct
        | structObj
        | callFuncStruct
        | reasigVarStruct
;
```

```
//Manejo de Variables
reasignacion: tiposId '=' expression # FuncionReasign;
incremento: tiposId '+''=' expression # FuncionIncremento;
decremento: tiposId '-''=' expression # FuncionDecremento;
tipoInit: Var |
Let;
tiposAsign:
        IntDecla
        | FloatDecla
        | StringDecla
        | BoolDecla
        | CharDecla
        | IdMayus
;

//Llamada de funciones
callFuncstmt: Id '(' (exprListCallFunc)? ')' # FuncionCallFunc
| Id '(' (exprVector)? ')' # FuncionCallFunc2
;
callFuncStruct: tiposId '.' tiposId'('')' # FuncionCallFuncStrcut;
reasigVarStruct: SELF '.' tiposId tipoIgual expression  # FuncionSelfReasig;
tipoIgual:'='
|'+''='
|'-''='
;
```

```
//Reservadas
Var:'var';
Let:'let';
Print: 'print';
IntDecla: 'Int';
FloatDecla: 'Float';
BoolDecla: 'Bool';
StringDecla: 'String';
CharDecla: 'Character';
IF:'if';
FUNC:'func';
INOUT:'inout';
STRUCT:'struct';
RETURN:'return';
ELSE:'else';
```

```
FOR: 'for';
WHILE: 'while';
SWITCH: 'switch';
APEND:'append';
CASE: 'case';
DEFAULT: 'default';
BREAK:'break';
GUARD:'guard';
COUNT:'count';
REMOVE:'remove';
REMOVELAST:'removeLast';
AT:'at';
RANGE: '...';
CONTINUE:'continue';
REPEATING:'repeating';
MUTANT:'mutating';
```

```
//Declaracion varaibles, vectores, matrices,funciones y structs.
asignacion: tipoInit tiposId '=' structAsig # funcionAsigStruct
        | op=(Let|Var) tiposId ':' tiposAsign '=' expression # funcionAsigTipoExp
        | op=(Let|Var) tiposId  ':' tiposAsign '?' # funcionAsigTipoNil
        | op=(Let|Var) tiposId  '=' expression  # funcionAsigExp
;

matrizAsign: Var Id ':' '[' '[' tiposAsign ']' ']' '=' '[' exprListMatrixDecla ']'                    # FuncionAsignarMatrizNormal
|       Var Id ':' '[' '['']'  tiposAsign ']' ']' ']' '=' '[' ( '[' exprListMatrixDecla ']' )* ']'     # FuncionAsignarMatriz3D
|       Var Id ':' '[' '[' '['  tiposAsign ']' ']' ']' '=' '[' '['']'  tiposAsign ']' ']' ']' '(' REPEATING ':' '['
'[' tiposAsign ']' ']' '(' REPEATING ':' '[' tiposAsign ']' ',' '(' REPEATING ':' expression ',' COUNT ':'
expression ')' ','  COUNT ':' expression')' ',' COUNT ':' expression ')'  # FuncionAsignarM3D;

defStruct: STRUCT IdMayus '{' (atributosLista| atributosLista2)* (funcStructs)* '}' # FuncionDefStruct;

vectorAsign: Var tiposId ':' '[' tiposAsign ']' '=' '[' (exprVector)? ']' # FuncionVectorAsig
        | Var tiposId ':' '[' tiposAsign ']' '=' tiposId # FuncionVectorAsigVar
        | Var tiposId '=' '[' IdMayus ']' '(' ')' # FuncionVectorAsigVarStruct
        | Var tiposId '=' '[' structAsig (',' structAsig)* ']' # FuncionVectorAsigVarObjs;

funcstmt: FUNC Id '(' (exprListFunc|exprListFuncBajo)? ')' '->' tiposAsign '{' sentencias '}' # FuncionDeclaFunc
        | FUNC Id '(' (exprListFunc|exprListFuncBajo)? ')' '{' sentencias '}' # FuncionDeclaFunc2;
```

```
funcStructs: FUNC tiposId '('')' '{' sentencias '}'     # FuncionCrearFunc
| MUTANT FUNC tiposId '('')' '{' sentencias '}'                  # FuncionCrearFuncMut
;

//Lista expresiones
exprListStruct: tiposId ':' expr_struct ( ',' tiposId ':' expr_struct )* # listAtibStruct;
exprListFunc: dataFuncTipo (','dataFuncTipo)*   # listaFuncConTipo;
dataFuncTipo:tiposId tiposId ':'(INOUT)? expr_llave? tiposAsign expr_llave2? # FuncionDataFuncTipo;
exprListFuncBajo: dataFuncBajo (',' dataFuncBajo)*      # listaFuncConBarra;
dataFuncBajo:'_' tiposId ':'(INOUT)? expr_llave? tiposAsign expr_llave2? # FuncionDataFuncBajo;
expr_llave:('[')*;
expr_llave2:(']')*;
exprListMatrixDecla: '[' exprVector ']' (',' '[' exprVector ']')*       # exprListMatrix;
exprListCallFunc: tiposId ':' expression ( ',' tiposId ':' expression )* ;
exprVector: expression ( ',' expression )*;
expr_struct:(expression |structAsig) # retornoExpStruct
;
```

```
//Funciones matrices y vectores
reasigMatriz: Id '[' expression ']' '[' expression ']' '=' expression           # FuncionReasignMatriz
                      |       Id '[' expression ']' '[' expression ']' '[' expression ']' '=' expression             # FuncionReasignMatriz3D;

removeVec:tiposId '.' REMOVE '(' AT ':' expression')' # FuncionRemoveVec;

vecReasig: tiposId'[' expression ']' '=' expression # FuncionVecReasig;

removeLastVec:tiposId'.' REMOVELAST '('')' # FuncionRemoveLastVec;

appendVec: tiposId '.' APEND '('expression')' # FuncionAppendVector
| tiposId '.' APEND '('structAsig')' # FuncionAppendVectorStr;

//Struct
structAsig: IdMayus '(' (exprListStruct) ')'                                        # defStructExpression;
structObj: tiposId ('.' tiposId)+ '=' expression                        # FuncionReasigObj;
atributosLista2: op=(Let|Var) tiposId  ':' IdMayus      # FuncionAtributosStruct;
atributosLista: op=(Let|Var) tiposId (':' tiposAsign) # FuncionAtributosListTipo
| op=(Let|Var) tiposId ('=' expression)         # FuncionAtributosListExp;
```

```
expression: '!' expression                                                          # funcionNot
        | '-' expression                                                                    # funcionUnariaExp
        | <assoc = right> expression '^' expression                         # funcionPowExp
        | expression op = ('*' | '/' | '%') expression                  # expressionMultDivMod
        | expression op = ('+' | '-') expression                            # expressionSumRes
        | expression op = ('>=' | '<=' | '>' | '<') expression  # funcionCompExp
        | expression op = ('==' | '!=') expression                          # funcionEqExp
        | expression '&&' expression                                                # funcionAndExp
        | expression '||' expression                                                # funcionOrExp
        | expression '?' expression ':' expression                          # funcionTernaryExp
        | Nil                                                                                       # nilExpression
        | Number                                                                                    # numberExpression
        | BoolVal                                                                                   # boolExpression
        | Id                                                                                        # idExpression
        | String                                                                                    # stringExpression
        | '(' expression ')'                                                        # expressionExpression
        | callFuncstmt                                                                      # exprCalFunc
        | tiposId '.' COUNT                                                                 # countExpression
        | tiposId '.' 'IsEmpty'                                                     # emptyVecExpression
        | tiposId '[' expression ']'                                                # vecCallExpression
        | tiposId '[' expression ']' '[' expression ']'                     # matrizCallExpression
        | tiposId '[' expression ']' '[' expression ']' '[' expression ']'      # matriz3DCallExpression
        | tiposId ('.' tiposId)+                                                    # callVarStructExpression
        | tiposAsign '(' expression ')'                                             # funcionesEmbeExpression
        | '&'tiposId                                                                # callArray
        | '&'tiposId '[' expression ']' '[' expression ']'              # callMatriz
        | SELF '.' tiposId                                                          # callSelfExp
        | tiposId '[' expression ']''.' tiposId                                     # vecStrCallExpression
;
```

# Conclusion:

El desarrollo del Compilador T-Swift ha sido un viaje emocionante y desafiante que nos ha permitido explorar en profundidad los aspectos clave de la compilación de lenguajes de programación. A lo largo de este proyecto, hemos logrado varios hitos importantes y hemos creado una poderosa herramienta que puede traducir código T-Swift en ejecutables funcionales. Aquí hay algunas conclusiones clave:

## Logros Destacados

- Compilador Funcional: Hemos logrado construir un compilador completamente funcional para el lenguaje T-Swift. Este compilador es capaz de realizar análisis léxicos, sintácticos y semánticos, así como de generar código ejecutable a partir del código fuente T-Swift.
- Manejo de Estructuras de Datos Complejas: Nuestro compilador puede manejar variables, vectores y matrices, así como estructuras de datos personalizadas (structs). Esto permite a los programadores crear aplicaciones complejas y escalables en T-Swift.
- Soporte para Funciones: Implementamos funciones y procedimientos que permiten modularizar el código y reutilizarlo de manera efectiva. Esto mejora la organización y legibilidad del código.
- Flexibilidad en Expresiones y Operadores: Hemos incorporado una amplia variedad de expresiones y operadores, lo que facilita la implementación de cálculos y lógica personalizada en el código T-Swift.
- Ciclos y Condicionales: Los bucles "for" y "while" junto con las estructuras condicionales "if" y "switch" ofrecen un control de flujo robusto para nuestros usuarios.