TRABAJO FINAL

CURSO DE FUNDAMENTOS TEORÍA DE COMPILADORES

Carrera de Ciencias de la Computación

Sección: CC61

Alumnos:

- Gabriel Alonso Reyna Alvarado
- Ian Joaquin Sanchez Alva
- Ibrahim Imanol Jordi Arquinigo Jacinto
- Alejandro Olaf López Flores

Septiembre 2024

Índice

1. Problema y motivación

1.1. Problema

Este trabajo consiste en la creación de un lenguaje de programación personalizado utilizando **ANTLR** para generar los analizadores léxicos y sintácticos, y planear implementar el lenguaje en **C++** en un sentido más natural y para **hispanohablantes**. El reto principal es definir una gramática robusta que pueda procesar las construcciones del lenguaje, generar el código correspondiente y hacerlo eficiente en un entorno C++. Ademas usaremos **LLVM** como backend para poder generar el **código IR de nuestro lenguaje**.

1.2. Motivación

1.2.1. Automatización del análisis léxico y sintáctico

La construcción manual de analizadores léxicos y sintácticos para un nuevo lenguaje de programación es un proceso complejo y propenso a errores. ANTLR proporciona una solución automatizada que permite definir la gramática del lenguaje de manera estructurada, generando automáticamente el código necesario para procesar las entradas de manera precisa. Esto no solo ahorra tiempo, sino que también mejora la consistencia y precisión del análisis, especialmente en lenguajes con sintaxis intrincada.

1.2.2. Flexibilidad en la evolución del lenguaje

Los lenguajes de programación suelen pasar por varias iteraciones y ajustes a medida que evolucionan. ANTLR facilita este proceso al permitir la modificación de la gramática de manera sencilla y rápida. Cada vez que se introducen nuevos elementos o reglas en el lenguaje, ANTLR regenera el analizador léxico y sintáctico sin necesidad de rehacer completamente la base del compilador. Esto aporta flexibilidad al desarrollo y garantiza que los cambios puedan ser implementados eficientemente.

1.2.3. Optimización mediante integración con C++

C++ es conocido por su capacidad de manejar memoria y recursos con gran precisión, lo que lo convierte en una opción ideal para crear compiladores y lenguajes de programación orientados al alto rendimiento. Al integrar ANTLR con C++, se aprovecha la eficiencia y el control que ofrece el lenguaje, optimizando el desempeño del código generado. Esto es especialmente relevante en aplicaciones de sistemas donde la eficiencia y el control de bajo nivel son esenciales.

1.2.4.Reducción de la complejidad en la definición de la gramática

Definir una gramática para un lenguaje de programación personalizado puede ser una tarea monumental, especialmente si el lenguaje incluye características avanzadas como expresiones anidadas o macros. ANTLR facilita este proceso al proporcionar una plataforma robusta para la definición y validación de gramáticas, reduciendo la complejidad y permitiendo al desarrollador centrarse en las funcionalidades esenciales del lenguaje en lugar de los detalles técnicos del análisis.

2. Objetivos

2.1. Automatizar la creación del análisis léxico y sintáctico del lenguaje

El objetivo es utilizar ANTLR para generar automáticamente los analizadores léxicos y sintácticos del lenguaje, eliminando la necesidad de desarrollar manualmente estas partes fundamentales de un compilador o intérprete. Esto permitirá un procesamiento eficiente y preciso del código fuente.

2.2. Facilitar la evolución y modificación del lenguaje

Definir una gramática flexible y escalable que pueda ser modificada con facilidad es fundamental para permitir la evolución del lenguaje. ANTLR facilitará la actualización del análisis sintáctico a medida que se añaden nuevas características o se ajusten las reglas gramaticales.

2.3. Optimizar el rendimiento del lenguaje mediante la integración con C++

El objetivo es aprovechar las capacidades de C++ para optimizar el rendimiento del código generado por ANTLR. Esto implica garantizar que los analizadores generados sean eficientes y se integren perfectamente con el entorno de C++ para maximizar el control y la gestión de recursos en la fase de ejecución.

2.4. Reducir la complejidad en la definición y manejo de la gramática

Otro objetivo clave es simplificar la definición de la gramática del nuevo lenguaje utilizando ANTLR, asegurando que el proceso de crear un conjunto de reglas gramaticales sea claro y manejable. Esto permitirá reducir los errores y aumentar la productividad en el desarrollo del compilador o intérprete.

3. Gramática en ANTLR4

PARSER Y LEXER:

```
grammar Expr;

// Regla de inicio
program : (principal | ioStatement | statement)* EOF ;

// ----------- GRAMÁTICA DE STATEMENTS -----------

statement
    : varDeclaracionStmt # VarDeStmt
    | varAsignacionStmt  # VarAsStmt
    | 'retorno' exp ';' # ReturnStmt
    | 'salida' '(' exp ')' ';' # OutputStmt
    | 'si' '(' exp ')' '{' statement* '}' ('sino' '{'
statement* '}')? # IfElseStmt
    | 'mientras' '(' exp ')' '{' statement* '}' #
WhileStmt
    ;


varDeclaracionStmt
    : tipo ID '=' exp ';'
    ;

varAsignacionStmt
    : ID '=' exp ';'
    ;

ioStatement
    : 'entrada' '(' ID ')' ';' # InputStatement
    | 'salida' '(' exp ')' ';' # OutputStatement
    ;

tipo
```

```
    : 'entero'  # Integer
    | 'flotante' # Float
    | 'cadena' # String
    ;

exp
    : ID                        # ID
    | NUM                       # Number
    | STRING                    # StringLiteral
    | '(' exp ')'               # Parenthesis
    | '-' exp                   # Negation
    | exp '*' exp               # Multiplication
    | exp '/' exp               # Division
    | exp '+' exp               # Addition
    | exp '-' exp               # Subtraction
    | exp '&&' exp              # LogicalAnd
    | exp '||' exp              # LogicalOr
    | exp '==' exp              # EqualityCheck
    | exp '!=' exp              # InequalityCheck
    | exp '>' exp               # GreaterThan
    | exp '<' exp               # LessThan
    ;

principal
    : 'entero' 'principal' '(' ')' '{' mainBody '}'
    ;

mainBody
    : statement* # StatementList
    ;

// ----------- REGLAS COMUNES -----------
ID      : [a-zA-Z_][a-zA-Z0-9_]* ;
NUM     : [-]?[0-9]+('.'[0-9]+)? ;  // Acepta tanto
enteros como flotantes
STRING  : '"' (~["\r\n])* '"' ;
WS      : [ \t\r\n]+ -> skip ;
```

```
COMMENT : '//' ~[\r\n]*->skip ;
```

DRIVER.H:

```
#ifndef DRIVER_H
#define DRIVER_H

#include "ExprBaseVisitor.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include <string>
#include <map>

using namespace antlr4;
using namespace llvm;
using namespace std;

class Driver : public ExprBaseVisitor {
private:
    LLVMContext &C;
    unique_ptr<Module> M;
    unique_ptr<IRBuilder<>> builder;
    map<string, Value *> variables; // Mapa de variables
declaradas

    IntegerType *i32Ty;

    // Crear una constante entera
    Value *ConstI32(int val) {
        return ConstantInt::get(i32Ty, val);
    }

    // Crear una constante de cadena
    Value *ConstString(const string &str) {
        return builder->CreateGlobalString(str);
```

```
    }

    // Obtener o crear la función printf
    Function *getOrCreatePrintf() {
        Function *printfFn = M->getFunction("printf");
        if (!printfFn) {
            FunctionType *printfTy = FunctionType::get(
                Type::getInt32Ty(C),
                PointerType::get(Type::getInt8Ty(C), 0),
true);
            printfFn = Function::Create(printfTy,
Function::ExternalLinkage, "printf", M.get());
        }
        return printfFn;
    }

public:
    Driver(const string &sourceFileName, LLVMContext &C)
        : C(C), M(make_unique<Module>(sourceFileName,
C)), builder(make_unique<IRBuilder<>>(C)) {
        i32Ty = IntegerType::getInt32Ty(C);
    }

    Module *getModule() {
        return M.get();
    }

    // Función principal
    virtual std::any
visitPrincipal(ExprParser::PrincipalContext *ctx)
override {
        FunctionType *mainFnTy = FunctionType::get(i32Ty,
false);
        Function *mainFn = Function::Create(mainFnTy,
Function::ExternalLinkage, "language", M.get());
        BasicBlock *entryBB = BasicBlock::Create(C,
"language.0", mainFn);
```

```
        builder->SetInsertPoint(entryBB);

        visit(ctx->mainBody());

        builder->CreateRet(ConstI32(0));
        return nullptr;
    }

    // Declaración de variables
    virtual std::any
visitVarDeclaracionStmt(ExprParser::VarDeclaracionStmtCon
text *ctx) override {
        string varName = ctx->ID()->getText();
        Value *initVal = std::any_cast<Value
*>(visit(ctx->exp()));
        AllocaInst * alloc = builder->CreateAlloca(i32Ty,
nullptr, varName.c_str());
        variables[varName] = alloc;
        builder->CreateStore(initVal,
variables[varName]);
        return nullptr;
    }

    // Asignación de variables
    virtual std::any
visitVarAsignacionStmt(ExprParser::VarAsignacionStmtConte
xt *ctx) override {
        string varName = ctx->ID()->getText();
        Value *newVal = std::any_cast<Value
*>(visit(ctx->exp()));
        if (!variables.count(varName)) {
            cerr << "Error: Variable no declarada: " <<
varName << endl;
            exit(1);
        }
        builder->CreateStore(newVal, variables[varName]);
        return nullptr;
```

```
    }

    // Manejo de identificadores (variables)
    virtual std::any visitID(ExprParser::IDContext *ctx)
override {
        string varName = ctx->getText();
        if (!variables.count(varName)) {
            cerr << "Error: Variable no declarada: " <<
varName << endl;
            exit(1);
        }
        AllocaInst *varPtr =
cast<AllocaInst>(variables[varName]);

        Value *varValue = builder->CreateLoad(i32Ty,
varPtr, varName.c_str());

        return varValue;
    }

    virtual std::any
visitParenthesis(ExprParser::ParenthesisContext *ctx)
override {
        return std::any_cast<Value *>(visit(ctx->exp()));
    }

    // Operaciones aritméticas
    virtual std::any
visitAddition(ExprParser::AdditionContext *ctx) override
{
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateAdd(lhs, rhs, "addtmp");
    }
```

```
    virtual std::any
visitSubtraction(ExprParser::SubtractionContext *ctx)
override {
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateSub(lhs, rhs, "subtmp");
    }

    virtual std::any
visitMultiplication(ExprParser::MultiplicationContext
*ctx) override {
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateMul(lhs, rhs, "multmp");
    }

    virtual std::any
visitDivision(ExprParser::DivisionContext *ctx) override
{
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateSDiv(lhs, rhs, "divtmp");
    }

    // Salida de valores
    virtual std::any
visitOutputStmt(ExprParser::OutputStmtContext *ctx)
override {
        Value *output = std::any_cast<Value
*>(visit(ctx->exp()));
```

```
        Function *printfFn = getOrCreatePrintf();
        Value *formatStr;

        if (output->getType()->isIntegerTy()) {
            formatStr = ConstString("%d\n");
        } else if (output->getType()->isPointerTy()) {
            formatStr = ConstString("%s\n");
        } else {
            cerr << "Error: Tipo no soportado en salida."
<< endl;
            exit(1);
        }

        builder->CreateCall(printfFn, {formatStr,
output});
        return nullptr;
    }

// Comparaciones lógicas
    virtual std::any
visitGreaterThan(ExprParser::GreaterThanContext *ctx)
override {
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateICmpSGT(lhs, rhs, "gttmp");
// Mayor que
    }

    virtual std::any
visitLessThan(ExprParser::LessThanContext *ctx) override
{
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
```

```
        return builder->CreateICmpSLT(lhs, rhs, "lttmp");
// Menor que
    }

    virtual std::any
visitEqualityCheck(ExprParser::EqualityCheckContext *ctx)
override {
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateICmpEQ(lhs, rhs, "eqtmp");
// Igualdad
    }

    virtual std::any
visitInequalityCheck(ExprParser::InequalityCheckContext
*ctx) override {
        Value *lhs = std::any_cast<Value
*>(visit(ctx->exp(0)));
        Value *rhs = std::any_cast<Value
*>(visit(ctx->exp(1)));
        return builder->CreateICmpNE(lhs, rhs, "netmp");
// Desigualdad
    }
// Declaración de condiciones (if-else)
    virtual std::any
visitIfElseStmt(ExprParser::IfElseStmtContext *ctx)
override {
        Value *cond = std::any_cast<Value
*>(visit(ctx->exp()));
        cond = builder->CreateICmpNE(cond, ConstI32(0),
"ifcond");

        Function *fn =
builder->GetInsertBlock()->getParent();
```

```
        BasicBlock *thenBB = BasicBlock::Create(C,
"then", fn);
        BasicBlock *elseBB = BasicBlock::Create(C,
"else");
        BasicBlock *mergeBB = BasicBlock::Create(C,
"merge");

        builder->CreateCondBr(cond, thenBB, elseBB);

        builder->SetInsertPoint(thenBB);
        visit(ctx->statement(0));
        builder->CreateBr(mergeBB);

        elseBB->insertInto(fn);
        builder->SetInsertPoint(elseBB);

        if (ctx->statement(1)) {
            visit(ctx->statement(1));
        }
        builder->CreateBr(mergeBB);

        mergeBB->insertInto(fn);
        builder->SetInsertPoint(mergeBB);
        return nullptr;
    }

    // Manejo de números
    virtual std::any
visitNumber(ExprParser::NumberContext *ctx) override {
        return ConstI32(stoi(ctx->getText()));
    }

    // Manejo de cadenas literales
    virtual std::any
visitStringLiteral(ExprParser::StringLiteralContext *ctx)
override {
        string str = ctx->STRING()->getText();
```

```
        str = str.substr(1, str.length() - 2); // Remover
comillas
        return ConstString(str);
    }
};

#endif // DRIVER_H
```

MAIN.CPP:

```
#include <iostream>
#include <fstream>
#include <memory>

#include "Driver.h"
#include "ExprLexer.h"
#include "ExprParser.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Verifier.h"
#include "llvm/Support/raw_ostream.h"

using namespace antlr4;
using namespace llvm;
using namespace std;

void addMainFunction(Module *M) {
    LLVMContext &C = M->getContext();
    FunctionType *mainTy =
FunctionType::get(Type::getInt32Ty(C), {}, false);
    Function *mainFn = Function::Create(mainTy,
Function::ExternalLinkage, "main", M);
    BasicBlock *mainBB = BasicBlock::Create(C, "entry",
mainFn);
    IRBuilder<> builder(mainBB);
```

```
    // Llamar a la función "language"
    Function *languageFn = M->getFunction("language");
    if (!languageFn) {
        errs() << "Error: 'language' function not found
in module.\n";
        return;
    }
    builder.CreateCall(languageFn);

builder.CreateRet(ConstantInt::get(Type::getInt32Ty(C),
0));
}

int main(int argc, const char *argv[]) {
    // Leer archivo de entrada o stdin
    ifstream inputFile;
    string sourceFileName = "stdin";
    if (argc > 1) {
        sourceFileName = argv[1];
        inputFile.open(sourceFileName);
        if (!inputFile.is_open()) {
            cerr << "Error: Cannot open file " <<
sourceFileName << "\n";
            return 1;
        }
    }
    istream &inputStream = (argc > 1) ? inputFile : cin;

    ANTLRInputStream input(inputStream);
    ExprLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    ExprParser parser(&tokens);

    // Contexto LLVM
    LLVMContext context;
    Driver driver(sourceFileName, context);
```

```
    // Generar IR
    try {
        unique_ptr<Module> module(driver.getModule());
        driver.visit(parser.program());

        addMainFunction(module.get());

        // Imprimir IR generado
        module->print(outs(), nullptr);

        // Verificar módulo
        if (verifyModule(*module, &errs())) {
            errs() << "Error: module verification
failed.\n";
            return 1;
        }
    } catch (const exception &e) {
        cerr << "Error during code generation: " <<
e.what() << "\n";
        return 1;
    }

    return 0;
}
```

Ejemplo (ej2.expr):

```
entero principal() {
    entero a = 9;
    entero b = 11 + (5 * 2 / (2*1));

    a = 5 + b;
    salida("a es igual: ");
    salida(a);
```

```
    salida("b es igual: ");
    salida(b -  -1);
    si (10 > 1){
        salida("10 es mayor a 1");
    } sino {
        salida("10 no es mayor a 1");
    }
}
```

Generación del IR code (ej2.ll):

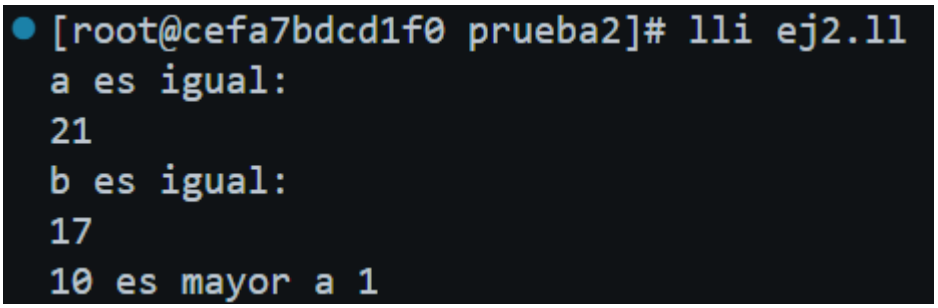
- Se genera el código con el comando: `build/prog ej2.expr > ej2.ll`
- Y luego este comando: `lli ej2.ll # para poder ver los resultados abajo`

```
; ModuleID = 'ej2.expr'
source_filename = "ej2.expr"

@0 = private unnamed_addr constant [13 x i8] c"a es
igual: \00", align 1
@1 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
align 1
@2 = private unnamed_addr constant [4 x i8] c"%d\0A\00",
align 1
@3 = private unnamed_addr constant [13 x i8] c"b es
igual: \00", align 1
@4 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
align 1
@5 = private unnamed_addr constant [4 x i8] c"%d\0A\00",
align 1
@6 = private unnamed_addr constant [16 x i8] c"10 es
mayor a 1\00", align 1
@7 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
align 1
```

```
@8 = private unnamed_addr constant [19 x i8] c"10 no es
mayor a 1\00", align 1
@9 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
align 1

define i32 @language() {
language.0:
  %a = alloca i32, align 4
  store i32 9, ptr %a, align 4
  %b = alloca i32, align 4
  store i32 16, ptr %b, align 4
  %b1 = load i32, ptr %b, align 4
  %addtmp = add i32 5, %b1
  store i32 %addtmp, ptr %a, align 4
  %0 = call i32 (ptr, ...) @printf(ptr @1, ptr @0)
  %a2 = load i32, ptr %a, align 4
  %1 = call i32 (ptr, ...) @printf(ptr @2, i32 %a2)
  %2 = call i32 (ptr, ...) @printf(ptr @4, ptr @3)
  %b3 = load i32, ptr %b, align 4
  %subtmp = sub i32 %b3, -1
  %3 = call i32 (ptr, ...) @printf(ptr @5, i32 %subtmp)
  br i1 true, label %then, label %else

then:                                             ; preds
= %language.0
  %4 = call i32 (ptr, ...) @printf(ptr @7, ptr @6)
  br label %merge

else:                                             ; preds
= %language.0
  %5 = call i32 (ptr, ...) @printf(ptr @9, ptr @8)
  br label %merge

merge:                                            ; preds
= %else, %then
  ret i32 0
}
```

```
declare i32 @printf(ptr, ...)

define i32 @main() {
entry:
  %0 = call i32 @language()
  ret i32 0
}
```

Ejemplo con interpretador:



Funcionalidades:

- **Manejo de variables**
- **Manejo de if y else en español**
- **Manejo de cálculos básicos de una calculadora con operaciones básicas (suma, resta, división y multiplicación).**
- **Lectura de variables**
- **Salida de variables y de texto (ejemplo: salida(“hola”), salida(a_variable))**
- **números negativos**
- **función main principal**

Link del repositorio:
https://github.com/forestgump22/TP-Compiladores/tree/main/TF-COMPILADORES

4. Referencias Bibliográficas:

- GitHub. (2024). *TP-Compiladores* [Repositorio de código]. GitHub. https://github.com/forestgump22/TP-Compiladores/tree/main/TF-COMPILADORES

- Guerrero, M. (s. f.). *Lenguaje latino*. https://www.lenguajelatino.org/
- Antlr4. (s. f.). *antlr4/doc/index.md at master · antlr/antlr4*. GitHub. https://github.com/antlr/antlr4/blob/master/doc/index.md
- Jeffery, C. L. (2021). *Build your own programming language*. O'Reilly Online Learning. https://learning.oreilly.com/library/view/build-your-own/9781800204805/