Project Report

Of

Compiler design Laboratory

(CSE605)

Bachelor of Technology (CSE)

By

Nirav Lad (22000743)

Harshil Brahmani (22000744)

Third Year, Semester 6

Course In-charge: Prof. Vaibhavi Patel

NAVRACHANA UNIVERSITY

Accredited with Grade 'A' by NAAC

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester

(2024-2025)

Introduction

Language is named as: Hinglish

This project implements a compiler for a custom programming language using Flex (for lexical analysis) and Bison (for parsing and Three-Address Code (TAC) generation). The compiler processes input programs written in the custom language, performs lexical and syntactic analysis, and generates TAC as output. The language supports assignments, arithmetic operations, conditionals, loops, and output statements. This document describes the language's syntax, implementation details, a comprehensive test case, and instructions for running the compiler.

Language Specification

The custom language is designed to be simple and uses Hindi-inspired keywords for accessibility. It supports fundamental programming constructs.

Keywords

- shuru: Marks the program start (like main in C).
- bolo: Outputs a variable or expression value (like print).
- agar: Represents an if condition (like if).
- nahi to: Represents the else clause (like else).
- jabtak: Represents a while loop (like while).

Operators

- **Arithmetic**: + (addition), - (subtraction), * (multiplication), / (division).
- **Relational**: == (equal), != (not equal), < (less than), > (greater than), <= (less than or equal), >= (greater than or equal).
- **Logical**: && (AND), || (OR), ! (NOT).
- **Assignment**: =.

Syntax Rules

- **Program Structure**: Starts with shuru, followed by statements.
- **Statements**:
  - **Assignment**: ID = Expression;, where ID is an identifier.
  - **Output**: bolo Expression;, prints the expression value.
  - **Conditional**: agar (Expression) { Statements } nahi to { Statements }, for if-else.
  - **Loop**: jabtak (Expression) { Statements }, for while loops.
- **Identifiers**: Alphanumeric strings starting with a letter or underscore (e.g., sum, i).
- **Numbers**: Integer literals (e.g., 123).

- **Expressions**: Combine identifiers, numbers, and operators, with parentheses for grouping.

Example Program

```
shuru
sum = 0;
i = 1;
jabtak (i <= 3) {
   sum = sum + i;
   i = i + 1;
}
bolo sum;
```

This program computes the sum of numbers from 1 to 3 and prints it.

Implementation Details

The compiler uses Flex for lexical analysis and Bison for parsing and TAC generation.

Lexical Analysis (lex.l)

The lexer (lex.l) tokenizes the input into:

- **Keywords**: shuru, bolo, agar, nahi to, jabtak.
- **Operators**: Arithmetic, relational, logical, and assignment.
- **Identifiers**: Alphanumeric strings.
- **Numbers**: Integers.
- **Symbols**: ;, (,), {, }.
  Whitespace is ignored, and unrecognized characters trigger an error. Tokens are returned with values stored in yylval.str.

Syntactic Analysis and TAC Generation (yacc.y)

The parser (yacc.y) uses a context-free grammar to parse tokens and generate TAC. It includes:

- **Grammar Rules**: Define program, statement, and expression structures.
- **Semantic Actions**: Generate TAC using temporary variables (t1, t2, etc.) and labels (L1, L2, etc.).
- **Temporary Variables**: Created via newTemp() for intermediate results.
- **Labels**: Created via newLabel() for control flow.
  The parser handles assignments, output, conditionals, loops, and expressions, writing TAC to output.txt.

Input and Output Handling

The input is read from input.txt, and TAC is written to output.txt. The main function in yacc.y manages file operations and invokes yyparse().

Three-Address Code (TAC)

TAC instructions have at most three operands, e.g.:

- **Assignment**: t1 = 0, sum = t1.
- **Arithmetic**: t5 = sum + i.
- **Conditional Jump**: if t4 goto L2.
- **Unconditional Jump**: goto L1.
- **Output**: print sum.

Test Case: Comprehensive Program

This test program demonstrates all implemented functionalities: assignments, arithmetic, logical and relational operators, conditionals, loops, and output.

Test Program (input.txt)

```
shuru

x = 20;

y = 5;

z = x * y + 10 / 2;

agar (z > 50 && x != y || ! (y == 5)) {

   result = z - x;

   bolo result;

} nahi to {

   result = z + x;

   bolo result;

}

counter = 1;

sum = 0;
```

```
jabtak (counter <= 4) {
    sum = sum + counter * 2;
    counter = counter + 1;
}
bolo sum;
```

Expected TAC (output.txt)

```
t1 = 20
x = t1
t2 = 5
y = t2
t3 = x * y
t4 = 10
t5 = 2
t6 = t4 / t5
t7 = t3 + t6
z = t7
t8 = 50
t9 = z > t8
t10 = x != y
t11 = t9 && t10
t12 = 5
t13 = y == t12
t14 = !t13
t15 = t11 || t14
```

```
if t15 goto L1
goto L2
L1:
t16 = z - x
result = t16
print result
goto L3
L2:
t17 = z + x
result = t17
print result
L3:
t18 = 1
counter = t18
t19 = 0
sum = t19
L4:
t20 = 4
t21 = counter <= t20
if t21 goto L5
goto L6
L5:
t22 = 2
t23 = counter * t22
```

t24 = sum + t23

sum = t24

t25 = 1

t26 = counter + t25

counter = t26

goto L4

L6:

print sum

How to Run the Compiler

1. Install Flex and Bison.
2. Compile the lexer and parser:
3. flex lex.l
4. bison -d yacc.y
5. gcc lex.yy.c yacc.tab.c -o compiler
6. Place the test program in input.txt.
7. Run the compiler:
8. ./compiler
9. Check the TAC in output.txt.

Conclusion

This project implements a compiler for a custom language using Flex and Bison, generating TAC for programs with assignments, arithmetic, conditionals, loops, and output. The test case demonstrates all functionalities, and the documentation provides a comprehensive guide for understanding and extending the compiler.