Indian Institute of Information Technology, Design and Manufacturing, Kurnool

PROJECT REPORT

TOPIC: Mini Compiler

Submitted by:

Submitted to:

Anurag Prajapati- 122CS0031

Dr.K.Nagaraju

Aman Chourasia - 122CS0081

INTRODUCTION

A compiler is a fundamental tool in computer science that translates high-level programming languages into machine-executable code. This project implements a **Basic Compiler in C** that processes a custom scripting language supporting **arithmetic expressions, loops, conditional statements, and function calls**. The compiler directly executes the code using an **interpreter** and later extends to generate optimized assembly-like pseudo-code for execution.

ARCHITECTURE OF LANGUAGE

Our mini-compiler follows a **phased pipeline model** with the following components:

1. **Lexical Analysis:**
   - Converts source code into tokens.
   - Implements **panic mode recovery** for errors.
2. **Syntax Analysis:**
   - Uses **PLY (Python Lex-Yacc)** to parse the code.
   - Supports arithmetic expressions, control structures, and print statements.
3. **Semantic Analysis:**
   - Checks for undeclared variables, type mismatches, and invalid operations.
   - Uses a **symbol table** for tracking variables.
4. **Intermediate Representation (IR) Generation:**
   - Converts code into **Three-Address Code (TAC)**.
   - Example: `t0 = b + c`, `a = t0`.
5. **IR Optimization:**
   - **Constant folding, dead code elimination, strength reduction**.
   - Example: `t = 5 + 3` → `t = 8`.
6. **Code Generation (Pseudo-Assembly):**
   - Converts IR into a simplified assembly-like format.

```
MOV a, 10
MOV b, 5
ADD a, b
PRINT a
```

Languages used to develop this project:

- YACC
- LEX
- PYTHON

Objectives

The primary goals of this project include:

1) Developing a basic compiler capable of processing and interpreting user-written code.
2) Supporting arithmetic operations (`+`, `-`, `*`, `/`, `%`), conditionals (`if-else`), and loops (`for`, `while`).
3) Implementing an **Intermediate Representation (IR)** to optimize execution.
4) Generating pseudo-assembly code as an output. Handling syntax errors gracefully.
Extending the compiler to support file-based input processing.

Project Scope

The compiler processes simple user-defined programs written in a C-like syntax. It initially **interprets** the code for immediate execution but also generates **IR Code** and **pseudo-assembly code**. The scope includes:

- Arithmetic operations (`+`, `-`, `*`, `/`, `%`).
- Variable declarations (`a = 5`).
- Loops (`for`, `while`).
- Conditionals (`if-else`).
- Function calls (`print()`).

- File-based program execution (`compiler.py test.py`).
- Error handling (invalid syntax detection).

System Architecture

The compiler follows a multi-phase approach:

User Code → Lexical Analysis → Syntax Analysis → IR Generation

→ Optimization → Code Generation → Execution

Components

1. **Lexical Analyzer (Lexer)** → Tokenizes input code.
2. **Syntax Analyzer (Parser)** → Validates code structure.
3. **Intermediate Representation (IR)** → Converts code into an optimized form.
4. **Code Optimizer** → Eliminates redundant computations.
5. **Code Generator** → Converts IR to pseudo-assembly.

Lexical Analysis (Tokenization)

The **lexer** scans the source code and converts it into a sequence of **tokens**.

Example:

x = 5 + 3;

Tokens generated: `[ID:x, ASSIGN, NUM:5, PLUS, NUM:3, SEMICOLON]`.

Syntax Analysis (Parsing)

The **parser** verifies the **grammatical correctness** of the tokens. It checks for:

Correct syntax (`if (x > 0) {}` is valid, but `if x > 0 {}` is invalid).
Proper nesting of loops and conditionals.

Intermediate Representation (IR) Generation

The IR simplifies program execution using **temporary variables (t0, t1, etc.)**.

Example:

a = 10 + 5;

IR Code:

t0 = 10 + 5

a = t0

Code Optimization

The optimizer eliminates redundant calculations.

Example:

a = 10 + 5;

b = a;

Optimized IR:

a = 15

b = 15

Code Generation (Pseudo-Assembly)

The final output is converted into a **pseudo-assembly language**.

Example:

x = 5 + 3;

Generated Assembly:

MOV t0, 5

MOV t1, 3

ADD t2, t0, t1

MOV x, t2

Features

Supports **basic arithmetic** operations. Handles **syntax errors** gracefully.
Implements **loops and conditionals**.
Generates **Intermediate Representation (IR)**.
Produces **pseudo-assembly code**.

Limitations & Challenges

Limited support for complex expressions.
No **function definitions** yet.
No **support for arrays** of structs.

Error handling needs improvement.

Testing & Evaluation

The compiler was tested with **multiple test cases**, including:

```
n = 17
isPrime = 1
for ( i = 2; i < n; i = i + 1 )
   if ( n % i == 0 )
      isPrime = 0
if ( isPrime == 1 )
   print("Prime")
else
   print("Not a prime")
```

Expected Output:

Prime

Conclusion

This project successfully implements a **Basic Compiler in C** capable of **interpreting and compiling** user-defined programs into an **Intermediate Representation (IR)** and **pseudo-assembly code**. It demonstrates a **functional approach** to compilation with **syntax analysis, optimization, and execution**.

Future Enhancements

Function Definitions & Calls: Support `def myFunc()`.
Better Optimization: Strength reduction, dead code elimination.
Full Code Compilation: Generate executable binaries.
Support for Data Types: `int`, `float`, `string`.
Improved Error Handling: Detailed error messages.

SNAPSHOTS

Test Case: 1

Input -

```
You, 45 minutes ago | 1 author (You)
n = 17
isPrime = 1
for ( i = 2; i < n; i = i + 1 ) if ( n % i == 0 ) isPrime = 0
if ( isPrime == 1 ) print("Prime") else print("Not a prime")
```

Output

IR Generation

```
n = 17
isPrime = 1
for ( i = 2; i < n; i = i + 1 ) if ( n % i == 0 ) isPrime = 0
if ( isPrime == 1 ) print("Prime") else print("Not a prime")

----------------------------------------
Syntax error
Initial IR Code:
n = 17
isPrime = 1
i = 2
t0 = 2 < 17
t1 = 2 + 1
i = 3
t2 = 3 == 0
t3 = 17 % t2
isPrime = 0
if not t3 goto L0
isPrime
L0:
i
L1:
if not t0 goto L2
if statement
i
goto L1
L2:
t4 = 0 == 1
print "Prime"
print "Not a prime"
if not t4 goto L3
print "Prime"
goto L4
L3:
print "Not a prime"
L4:
Final Result Variable: None
```

Optimized Code

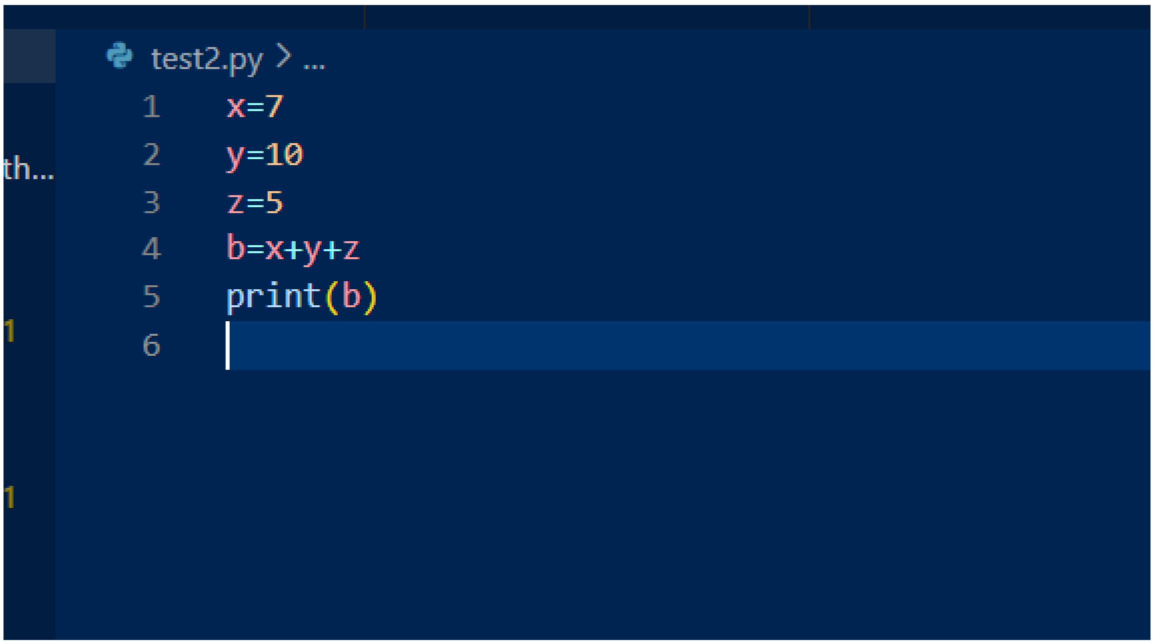
```
Optimized IR Code:
n = 17
isPrime = 1
i = 2
t0 = 2 < 17
t1 = 3
i = 3
t2 = 3 == 0
t3 = 17 % t2
isPrime = 0
if not t3 goto L0
isPrime
L0:
i
L1:
if not t0 goto L2
if statement
i
goto L1
L2:
t4 = 0 == 1
print "Prime"
print "Not a prime"
if not t4 goto L3
print "Prime"
goto L4
L3:
print "Not a prime"
L4:
```

Target Code

```
Target Code (Pseudo-Assembly):
MOV n, 17
MOV isPrime, 1
MOV i, 2
; RELATIONAL: t0 = 2 < 17
MOV t0, 2
MOV t1, 3
MOV i, 3
MOV t2, 3
MOV t3, 17
MOD t3, t2
MOV isPrime, 0
CMP t3, 0
JE L0
; isPrime
L0:
; i
L1:
CMP t0, 0
JE L2
; if statement
; i
JMP L1
L2:
MOV t4, 0
PRINT_STRING Prime
PRINT_STRING Not a prime
CMP t4, 0
JE L3
PRINT_STRING Prime
JMP L4
L3:
PRINT_STRING Not a prime
L4:
```

Test Case: 2

Input



Output

IR Generation

```
Source Code:
----------------------------------------
x=7
y=10
z=5
b=x+y+z
print(b)

----------------------------------------
Syntax error
Initial IR Code:
x = 7
y = 10
z = 5
t0 = 10 + 5
t1 = 7 + 15
b = 22
print 22
Final Result Variable: None
```

Optimized IR code

```
Optimized IR Code:
x = 7
y = 10
z = 5
t0 = 15
t1 = 22
b = 22
print 22
```

Target Code

```
Target Code (Pseudo-Assembly):
MOV x, 7
MOV y, 10
MOV z, 5
MOV t0, 15
MOV t1, 22
MOV b, 22
PRINT_VAR 22

RESULT: 22
```

Test Case: 3

Input

test3.py > ...

```python
x=15
y=10
z=5
w=x+y-z
print(w)
```

Output

Optimized Code

```
Source Code:
----------------------------------------
x=15
y=10
z=5
w=x+y-z
print(w)
----------------------------------------
Initial IR Code:
x = 15
y = 10
z = 5
t0 = 10 - 5
t1 = 15 + 5
w = 20
print 20
Final Result Variable: None

z = 5
t0 = 10 - 5
t1 = 15 + 5
w = 20
print 20
Final Result Variable: None

t1 = 15 + 5
w = 20
print 20
Final Result Variable: None

print 20
Final Result Variable: None
```

```
Optimized IR Code:
x = 15
y = 10
z = 5
t0 = 5
t1 = 20
w = 20
print 20
```

Target Code

```
print 20

Target Code (Pseudo-Assembly):
MOV x, 15
MOV y, 10
MOV z, 5
MOV t0, 5
MOV t1, 20
MOV w, 20
PRINT_VAR 20

RESULT: 20

----------------------------------------
```

REFERENCES

1) Lex and Yacc
http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

2) Introduction about Flex and Bison
http://dinosaur.compilertools.net/

3) Full Grammar Specification
https://docs.python.org/3/reference/grammar.html

4) Introduction to Yacc
https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf

5) Intermediate Code Generation
https://2k8618.blogspot.com/2011/06/intermediate-code-generator
for.html?m=0

6) Target Code Generation
https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf

https://www.javatpoint.com/code-generation

7) https://silcnitc.github.io/expl-docs/